quarta edição

SISTEMAS DISTRIBUÍDOS

CONCEITOS E PROJETO

George Coulouris
Jean Dollimore
Tim Kindberg

Bookman®

Hidden page

TECNOLOGIA DA INFORMAÇÃO

Engenharia e desenvolvimento de software

AMBLER, S. W.
Modelagem Ágil: Práticas Eficazes para Programação eXtrema e o Processo Unificado

BECK, K.
Programação eXtrema Explicada: Acolha as Mudanças

BRAUDE, E.
Projeto de Software: da Programação à Arquitetura, Uma Abordagem Baseada em Java

BURNETTE, E.
Eclipse IDE: Guia de Bolso

COCKBURN, A.
Escrevendo Casos de Uso Eficazes: Um Guia Prático para Desenvolvedores de Softwares

COULORIS, DOLLIMORE E KINDBERG
Sistemas Distribuídos: Conceitos e Projeto, 4.ed.

FOWLER, M.
Refatoração: Aperfeiçoando o Projeto de Código Existente

FOWLER, M.
UML Essencial: Um Breve Guia para a Linguagem-Padrão de Modelagem de Objetos, 3.ed.

FOWLER, M.
Padrões de Arquitetura de Aplicações Corporativas

GAMMA, HELM, JOHNSON & VLISSIDES
Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos

GUSTAFSON, D.
Engenharia de Software (COLEÇÃO SCHAUM)

HORSTMANN, C.
Padrões e Projeto Orientados a Objetos, 2.ed.

LARMAN, C.
Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientados a Objetos e ao Processo Unificado, 3.ed.

MARINESCU, F.
Padrões de Projeto EJB: Padrões Avançados, Processos e Idiomas

METSKER, S. J.
Padrões de Projeto em Java

SCOTT, K.
O Processo Unificado Explicado

SHALLOWAY & TROTT
Explicando Padrões de Projeto: Uma Nova Perspectiva em Projeto Orientado a Objeto

Bookman® Editora
Av. Jerônimo de Ornelas, 670
90040-340 Porto Alegre, RS, Brasil
Fone (51) 3027-7000 Fax (51) 3027-7070
e-mail: bookman@artmed.com.br

SISTEMAS DISTRIBUÍDOS
CONCEITOS E PROJETO

Obra originalmente publicada sob o título
Distributed Systems - Concepts and Design, 4th Edition

ISBN 0321263545

Capa: *Gustavo Demarchi,* arte sobre capa original

Leitura final: *Rachel Garcia Valdez*

Supervisão editorial: *Denise Weber Nowaczyk*

Editoração eletrônica: *Laser House*

SÃO PAULO
Av. Angélica, 1.091 – Higienópolis
01227-100 – São Paulo – SP
Fone: (11) 3665-1100 Fax: (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Prefácio

Esta quarta edição de nosso livro-texto aparece no momento em que a Internet e a web são sistemas amadurecidos, suportando uma ampla variedade de aplicações distribuídas em uma escala bem maior do que poderíamos ter antecipado quando nossa terceira edição foi publicada há quase cinco anos.

O objetivo deste livro é fornecer um entendimento sobre os princípios nos quais são baseados a Internet e outros sistemas distribuídos, sua arquitetura, algoritmos e projeto. Começamos com dois capítulos de visão conceitual que descrevem em linhas gerais as características dos sistemas distribuídos e os desafios que devem ser enfrentados em seu projeto: escalabilidade, heterogeneidade, segurança e tratamento de falhas sendo os mais significativos. Esses capítulos também desenvolvem modelos abstratos para entender a interação, falha e segurança de processos. Depois deles existem capítulos básicos dedicados ao estudo de interligação em rede, comunicação entre processos, invocação remota e *middleware*, suporte do sistema operacional e atribuição de nomes.

Abordamos em seguida os tópicos bem-estabelecidos de segurança, replicação de dados, comunicação em grupo, sistemas de arquivos distribuídos, transações distribuídas, CORBA, memória compartilhada distribuída e sistemas multimídia, junto com vários assuntos novos: serviços web, XML, grade (*grid*), serviços *peer-to-peer*, sistemas móveis e ubíquos. Os algoritmos associados a todos esses tópicos são abordados à medida que surgem e também em capítulos separados, dedicados à temporização, coordenação e acordo.

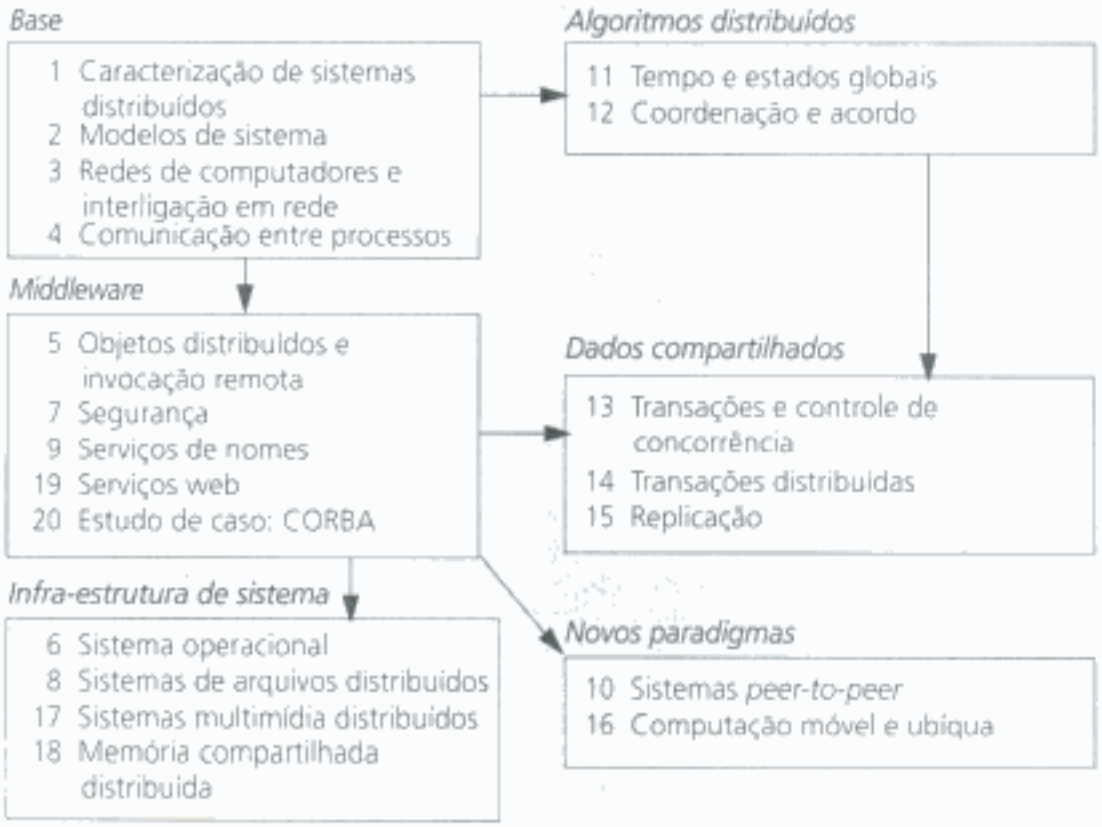
Objetivos e público-alvo

O livro se destina a cursos de graduação e cursos introdutórios de pós-graduação. Ele pode ser igualmente usado de forma autodidata. Adotamos a estratégia *top-down* (de cima para baixo), tratando dos problemas a serem resolvidos no projeto de sistemas distribuídos e descrevendo as estratégias bem-sucedidas na forma de modelos abstratos, algoritmos e estudos de caso detalhados sobre os sistemas mais utilizados. Abordamos a área com profundidade e amplitude suficientes para permitir aos leitores prosseguirem com seus estudos usando a maioria dos artigos de pesquisa presentes na literatura sobre sistemas distribuídos.

Nosso objetivo é tornar o assunto acessível a estudantes que tenham um conhecimento básico de programação orientada a objetos, sistemas operacionais e arquitetura de computadores. O livro inclui uma abordagem dos aspectos das redes de computadores, relevantes aos sistemas distribuídos, inclusive as tecnologias subjacentes da Internet, das redes de longa distância, locais e sem fio. Por todo o livro são apresentados algoritmos e interfaces em Java ou, em alguns casos, em C ANSI. Por brevidade e clareza da apresentação, também é usada uma forma de pseudocódigo, derivado de Java/C.

Organização do livro

O diagrama a seguir mostra os capítulos integrantes sob seis áreas de assunto principais. Seu objetivo é fornecer um guia para a estrutura do livro e indicar as rotas de navegação recomendadas para os instrutores que queiram fornecer (ou leitores que queiram obter) um entendimento sobre as várias subáreas do projeto de sistemas distribuídos:



Referências

A existência da World Wide Web mudou a maneira pela qual um livro como este pode ser vinculado ao material de origem, incluindo artigos de pesquisa, especificações técnicas e padrões. Muitos dos documentos de origem agora estão disponíveis na web; alguns estão disponíveis somente nela. Por motivos de brevidade e facilidade de leitura, empregamos uma forma especial de referência ao material da web, que se assemelha um pouco a um URL: citações como [www.omg.org] e [www.rsasecurity.com I] se referem à documentação que está disponível somente na web. Elas podem ser pesquisadas na lista de referências existente no final do livro, mas os URLs completos são dados apenas em uma versão *on-line* dessa lista, no site do livro www.cdk4.net/refs, onde assumem a forma de *links* e podem ser acessados diretamente com um simples clique de mouse. As duas versões da lista de referências incluem uma explicação mais detalhada desse esquema.

Alterações relativas à terceira edição

Antes de começarmos a escrever esta nova edição, fizemos um levantamento junto aos professores que utilizaram a terceira edição. A partir dos resultados, identificamos o novo material exigido e as alterações a serem feitas. Isso nos levou a escrever três capítulos totalmente novos e a fazer numerosas inserções por todo o livro. Todos os capítulos foram alterados para refletir as novas informações que se tornaram disponíveis sobre os sistemas descritos. Entretanto, para ajudar os professores que usaram a terceira edição, deixamos a estrutura dos capítulos existentes praticamente inalterada. Os capítulos

Capítulos novos:		
10	Sistemas *peer-to-peer*	A partir do capítulo 10, os capítulos receberam numeração nova nesta edição.
16	Computação móvel e ubíqua	
19	Serviços web	
Capítulos nos quais foi acrescentado material, mas sem alteração estrutural:		
1	Caracterização dos sistemas distribuídos	Seção 1.3.1: atualizada para introduzir serviços web
2	Modelos de sistema	Seção 2.2.2: atualizada para introduzir sistemas *peer-to-peer*
3	Redes de computadores e interligação em rede	Muitas atualizações Nova Seção 3.5.3: Estudo de caso: Bluetooth
4	Comunicação entre processos	Nova Seção 4.3.3: XML
7	Segurança	Várias atualizações Nova Seção 7.6.4: deficiências do WiFi
9	Serviços de nome	Seção 9.1.1: seção sobre URIs atualizada
20	Estudo de caso: CORBA	Seção 20.2.1: atualizada para Java 2 v. 1.4 Seção 20.2.6: integração com serviços web
Os capítulos restantes sofreram pequenas modificações.		

novos e aqueles que contêm alterações substanciais estão listados na tabela acima. O estudo de caso do Mach foi removido e está disponível no site do livro, junto com vários estudos de caso menores que foram retirados da segunda e da terceira edições.

Agradecimentos

Estamos muito gratos aos seguintes professores que participaram de nosso levantamento: Kay Robbins, Kohei Honda, Stefan Leue e Ian Wakeman.

Gostaríamos de agradecer às seguintes pessoas que revisaram os capítulos novos ou ajudaram substancialmente: John Barton, Arne Glenstrup, Roy Logie, Friedemann Mattern, Christian Mortensen, Anthony Rowstron, Bo Sanden, Dave Scott, Ben Smyth, Mirjana Spasojevic, Salman Taherian, Andrew Twigg, Jim Waldo, Eiko Yoneki, Kan Zhang e Ben Zhao.

O Departamento de Ciência da Computação, Queen Mary College, Universidade de Londres, que hospeda o site que acompanha a terceira edição do livro e concordou em hospedar o site da quarta edição. Agradecemos ao departamento por seu apoio e a Keith Clarke e à equipe de sistemas pela ajuda na configuração e na manutenção desses sites.

Finalmente, agradecemos a Simon Plumtree, Bridget Allen, Mary Lince e Owen Knight, da Pearson Education/Addison-Wesley, pelo apoio fundamental por todo o árduo processo de produção do livro.

Site web

Mantemos um site com uma ampla variedade de material destinado a ajudar os professores e leitores.* Esse site pode ser acessado por meio dos seguintes URLs:

www.cdk4.net www.pearsoned.co.uk/coulouris

O site inclui:

Guia do instrutor:

- Arte-final completa do livro, disponível como arquivos PowerPoint.

* Esses sites e os materiais complementares ali disponíveis estão em inglês.

- Soluções dos exercícios (protegidas por senha).
- Dicas de ensino capítulo a capítulo.
- Sugestões de projetos de laboratório.

Lista de referências: a lista de referências que pode ser encontrada no final do livro está reproduzida no site. A versão web da lista de referências inclui *links* para o material que está disponível on-line.

Lista de errata: uma lista dos erros conhecidos no livro, com suas correções. Tais erros serão corrigidos nas novas impressões e uma lista de errata separada será fornecida para cada impressão.*

Material suplementar: mantemos um conjunto de material suplementar para cada capítulo. Isso consiste no código-fonte dos programas presentes no livro e no material de leitura relevante que foi apresentado nas edições anteriores do livro, mas removido por razões de espaço. As referências a esse material suplementar aparecem no livro com *links* como www.cdk4.net/ipc.

***Links* para sites de cursos que utilizam o livro**: o site da terceira edição contém *links* para 15 cursos que utilizam nosso livro, os quais tornam disponíveis inúmeras anotações de conferência úteis, slides, exercícios e projetos de laboratório. Esperamos obter permissão dos professores desses cursos para colocar essas referências no novo site. Solicitamos aos outros professores para que nos informem sobre seus cursos com sites na web, para inclusão na lista.

George Coulouris
Jean Dollimore
***Tim** Kindberg*
<authors@cdk4.net>

* A errata disponível no momento da impressão desta obra já está corrigida nesta edição brasileira (dezembro de 2006).

Sumário

1 Caracterização de Sistemas Distribuídos

Um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens. Essa definição leva às seguintes características dos sistemas distribuídos: concorrência de componentes, falta de um relógio global e falhas de componentes independentes.

Daremos três exemplos de sistemas distribuídos:

- a Internet;
- uma intranet, que é uma parte da Internet gerenciada por uma organização;
- computação móvel e ubíqua (ou pervasiva).

O compartilhamento de recursos é um forte motivo para a construção de sistemas distribuídos. Os recursos podem ser gerenciados por servidores e acessados por clientes, ou podem ser encapsulados como objetos e acessados por outros objetos clientes. A web será discutida como um exemplo de compartilhamento de recursos e serão apresentadas suas principais características.

Os desafios advindos da construção de sistemas distribuídos são a heterogeneidade de seus componentes, ser um sistema aberto, o que permite que componentes sejam adicionados ou substituídos, a segurança, a escalabilidade – a capacidade de funcionar bem quando o número de usuários aumenta – o tratamento de falhas, a concorrência de componentes e a transparência.

1.1 Introdução

As redes de computadores estão por toda parte. A Internet é uma delas, assim como as muitas redes das quais ela é composta. Redes de telefones móveis, redes corporativas, redes de fábrica, redes em campus, redes domésticas, redes dentro de veículos, todas elas, tanto separadamente como em conjunto, compartilham as características básicas que as tornam assuntos relevantes para estudo sob o título *sistemas distribuídos*. Neste livro, queremos explicar as características dos computadores interligados em rede que afetam os projetistas e desenvolvedores de sistema e apresentar os principais conceitos e técnicas que foram criadas para ajudar nas tarefas de projeto e implementação de sistemas que os têm por base.

Definimos um sistema distribuído como sendo aquele no qual os componentes de hardware ou software, localizados em computadores interligados em rede, se comunicam e coordenam suas ações apenas enviando mensagens entre si. Essa definição simples abrange toda a gama de sistemas nos quais computadores interligados em rede podem ser distribuídos de maneira útil.

Os computadores conectados por meio de uma rede podem estar separados por qualquer distância. Eles podem estar em continentes separados, no mesmo prédio ou na mesma sala. Nossa definição de sistemas distribuídos tem as seguintes conseqüências importantes:

Concorrência: em uma rede de computadores, a execução concorrente de programas é a norma. Posso fazer meu trabalho em meu computador, enquanto você faz o seu em sua máquina, compartilhando recursos como páginas web ou arquivos, quando necessário. A capacidade do sistema de manipular recursos compartilhados pode ser ampliada pela adição de mais recursos (por exemplo, computadores) na rede. Vamos descrever como essa capacidade extra pode ser distribuída em muitos pontos de maneira útil. A coordenação de programas em execução concorrente e que compartilham recursos também é um assunto importante e recorrente.

Inexistência de relógio global: quando os programas precisam cooperar, eles coordenam suas ações trocando mensagens. A coordenação freqüentemente depende de uma noção compartilhada do tempo em que as ações dos programas ocorrem. Entretanto, verifica-se que existem limites para a precisão com a qual os computadores podem sincronizar seus relógios em uma rede – não existe uma noção global única do tempo correto. Essa é uma conseqüência direta do fato de que a *única* comunicação se dá por meio do envio de mensagens em uma rede. Exemplos desses problemas de sincronização e suas soluções serão descritos no Capítulo 11.

Falhas independentes: todos os sistemas de computador podem falhar e é responsabilidade dos projetistas de sistema pensar nas conseqüências das possíveis falhas. Nos sistemas distribuídos, as falhas são diferentes. Falhas na rede resultam no isolamento dos computadores que estão conectados a ela, mas isso não significa que eles param de funcionar. Na verdade, os programas neles existentes talvez não consigam detectar se a rede falhou ou se tornou demasiadamente lenta. Analogamente, a falha de um computador ou o término inesperado de um programa em algum lugar no sistema (um *colapso no sistema*) não é imediatamente percebida pelos outros componentes com os quais ele se comunica. Cada componente do sistema pode falhar independentemente, deixando os outros ainda em funcionamento. As conseqüências dessa característica dos sistemas distribuídos serão um tema recorrente por todo o livro.

A motivação para construir e usar sistemas distribuídos é proveniente do desejo de compartilhar recursos. O termo "recurso" é bastante abstrato, mas caracteriza bem o conjunto de coisas que podem ser compartilhadas de maneira útil em um sistema de computadores interligados em rede. Ele abrange desde componentes de hardware, como discos e impressoras, até entidades definidas pelo software, como arquivos, bancos de dados e objetos de dados de todos os tipos. Isso inclui o fluxo de quadros de vídeo proveniente de uma câmera de vídeo digital ou a conexão de áudio que uma chamada de telefone móvel representa.

O objetivo deste capítulo é transmitir uma visão clara da natureza dos sistemas distribuídos e dos desafios que devem ser enfrentados para garantir que eles sejam bem-sucedidos. A Seção 1.2 fornece alguns exemplos importantes de sistemas distribuídos, os componentes a partir dos quais eles são construídos e seus objetivos. A Seção 1.3 explora o projeto de sistemas com compartilhamento de

recursos no contexto da World Wide Web. A Seção 1.4 descreve os principais desafios enfrentados pelos projetistas de sistemas distribuídos: heterogeneidade, sistemas abertos, segurança, escalabilidade, tratamento de falhas, concorrência e a necessidade de transparência.

1.2 Exemplos de sistemas distribuídos

Nossos exemplos são baseados em redes de computadores que nos são familiares e amplamente difundidas: a Internet, intranets e a emergente tecnologia das redes baseadas em dispositivos móveis. Eles foram planejados de modo a exemplificar a ampla gama de serviços e aplicações suportadas pelas redes de computadores e para iniciar a discussão sobre os problemas técnicos que fundamentam a sua implementação.

1.2.1 A Internet

A Internet é um conjunto de redes de computadores, de muitos tipos diferentes, interligadas. A Figura 1.1 ilustra uma parte típica da Internet. Os programas que estão em execução nos computadores conectados a ela interagem enviando mensagens através de um meio de comunicação comum. O projeto e a construção dos mecanismos de comunicação da Internet (os protocolos Internet) são uma realização técnica importante, permitindo que um programa em execução em qualquer lugar envie mensagens para programas em qualquer outro lugar.

A Internet é um sistema distribuído muito grande. Ela permite que os usuários, onde quer que estejam, façam uso de serviços como a World Wide Web, e-mail e transferência de arquivos. Na verdade, às vezes, a web é confundida como sendo a Internet. O conjunto de serviços da Internet é aberto – ele pode ser ampliado pela adição de computadores servidores e novos tipos de serviços. A Figura 1.1 mostra um conjunto de intranets – sub-redes operadas por empresas e outras organizações. Os provedores de serviços de Internet (ISPs, do inglês *Internet Service Providers*) são empresas que fornecem, através de modem de linha discada ou outros tipos de conexões, o acesso a usuários individuais, ou organizações, à Internet. Esse acesso permite que os usuários utilizem os diferentes serviços disponibilizados pela Internet. Os provedores fornecem ainda alguns serviços locais, como e-mail e hospedagem de páginas web. As intranets são interligadas por meio de *backbones*. Um *backbone* é um

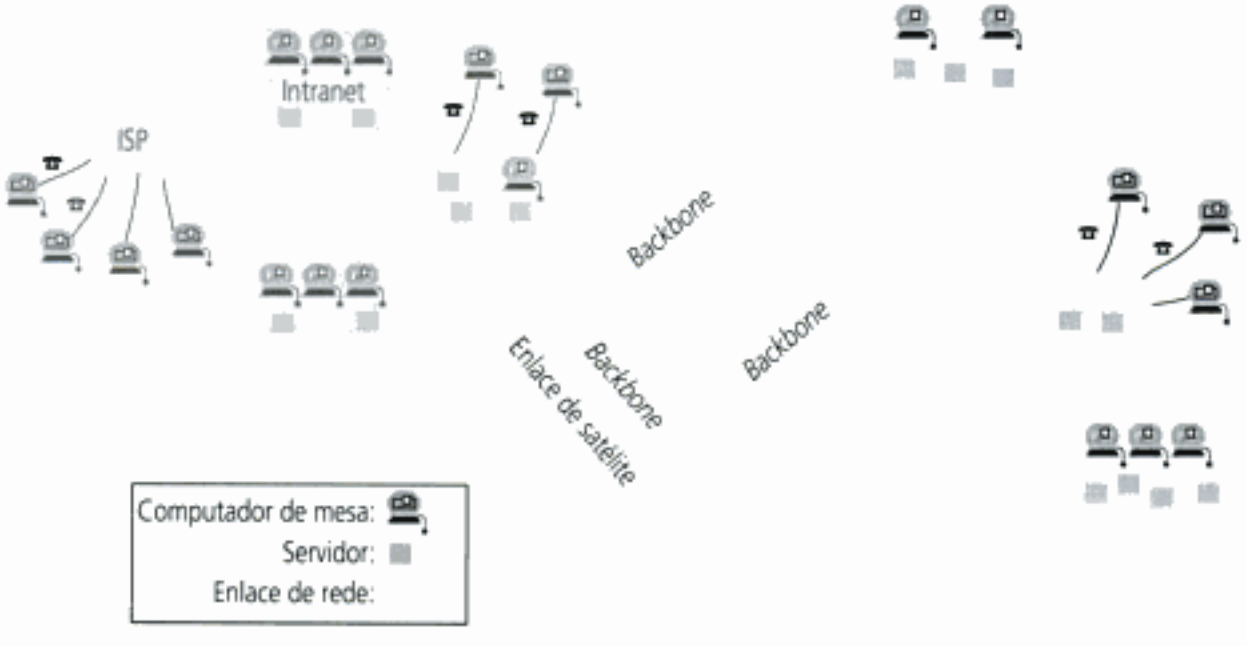


Figura 1.1 Uma parte típica da Internet.

enlace de rede com uma alta capacidade de transmissão, empregando conexões via satélite, cabos de *fibra óptica* ou outros meios físicos de transmissão que possuam uma grande largura de banda.

Uma série de serviços de multimídia estão disponíveis na internet, permitindo aos usuários acessarem dados de áudio e vídeo, o que inclui música, rádio, canais de TV e a manter conferências via telefone e vídeo. Atualmente, a capacidade da Internet para tratar os requisitos especiais da comunicação de dados multimídia é bastante limitada, pois ela não fornece os recursos necessários para reservar capacidade de rede para que fluxos de dados individuais sejam enviados e recebidos de forma adequada a dar uma boa qualidade de serviço. O Capítulo 17 discute as necessidades dos sistemas multimídia distribuídos.

A implementação da Internet e dos serviços que ela suporta tem acarretado o desenvolvimento de soluções práticas para muitos problemas dos sistemas distribuídos (incluindo a maior parte daqueles definidos na Seção 1.4). Vamos destacar essas soluções por todo o livro, apontando sua abrangência e suas limitações, onde for apropriado.

1.2.2 Intranets

Uma intranet é uma parte da Internet administrada separadamente, cujo limite pode ser configurado para impor planos de segurança locais. A Figura 1.2 mostra uma intranet típica. Ela é composta de várias redes locais (*Local Area Networks* – LANs) interligadas por conexões de *backbone*. A configuração de rede de uma intranet em particular é de responsabilidade da organização que a administra e pode variar muito – desde uma LAN em um único site até um conjunto de LANs interconectadas pertencentes às filiais de uma empresa ou outra organização, em diferentes países.

Uma intranet é conectada à Internet por intermédio de um roteador, o qual permite aos usuários de dentro dessa intranet utilizarem serviços que são providos em outro lugar, como acesso a servidores web ou de correio eletrônico. Ela também permite que os usuários de outras intranets acessem os serviços que fornece. Muitas organizações precisam proteger seus próprios serviços contra uso não autorizado, possivelmente por usuários externos mal-intencionados. Por exemplo, uma empresa não deseja que informações sigilosas estejam acessíveis para usuários de organizações concorrentes, e um hospital não deseja que dados sigilosos dos pacientes sejam revelados. As empresas também querem se proteger de programas prejudiciais, como os vírus que entram e atacam os computadores que estão na intranet, possivelmente, destruindo dados valiosos.

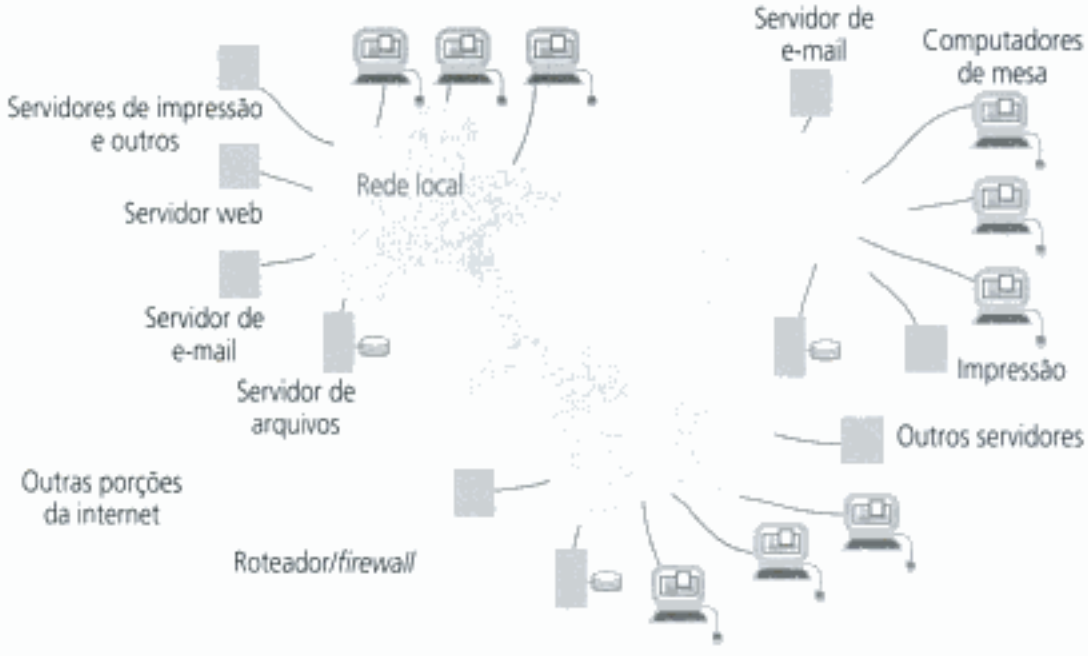


Figura 1.2 Uma intranet típica.

A função de um *firewall* é proteger uma intranet, impedindo a entrada ou saída de mensagens não autorizadas. Um *firewall* é basicamente composto de mecanismos que realizam a filtragem de mensagens recebidas e enviadas, de acordo, por exemplo, com sua origem ou destino. Um *firewall* poderia, por exemplo, na intranet que protege, permitir apenas a passagem das mensagens relacionadas a correio eletrônico e ao acesso ao servidor web.

Algumas organizações nem querem conectar suas redes internas na Internet. Por exemplo, a polícia e outras agências de segurança e do poder executivo provavelmente podem ter pelo menos algumas redes internas isoladas do mundo exterior. Algumas organizações militares desconectam suas redes internas da Internet em tempos de guerra. Mas, mesmo essas organizações desejam tirar proveito da enorme variedade de aplicações e de softwares que empregam protocolos de comunicação da Internet. A solução normalmente adotada por tais organizações é implementar uma intranet como a descrita anteriormente, mas sem conexões com a Internet. Tal intranet pode dispensar o uso de um *firewall*; na realidade, dizendo de outra maneira, ela já possui um *firewall* bastante eficaz: a ausência de qualquer conexão física com a Internet.

Os principais problemas que surgem no projeto de componentes para uso em intranets são:

- Serviços de arquivo são necessários para permitir que os usuários compartilhem dados; o seu projeto será discutido no Capítulo 8.
- Os *firewalls* tendem a impedir o acesso legítimo aos serviços – quando o compartilhamento de recursos entre usuários internos e externos é necessário, os *firewalls* devem ser complementados com o uso de outros mecanismos de segurança; isso será discutido no Capítulo 7.
- O custo da instalação de software e suporte é uma questão importante. Esses custos podem ser reduzidos com o uso de arquiteturas de sistema, como computadores de rede e clientes "leves", descritos no Capítulo 2.

1.2.3 Computação móvel e ubíqua

Os avanços tecnológicos na miniaturização de dispositivos e interligação em rede sem fio têm levado cada vez mais à integração de equipamentos de computação pequenos e portáteis com sistemas distribuídos. Esses equipamentos incluem:

- Computadores notebook
- Aparelhos portáteis, incluindo assistentes digitais pessoais (PDAs), telefones móveis, *pagers*, câmeras de vídeo e digitais
- Aparelhos acoplados ao corpo, como relógios de pulso inteligentes com funcionalidade semelhante à de um PDA
- Dispositivos incorporados em aparelhos, como máquinas de lavar, aparelhos de som de alta fidelidade, carros, geladeiras, etc.

A portabilidade de muitos desses dispositivos, junto com sua capacidade de se conectar convenientemente com redes em diferentes lugares, tornam a *computação móvel* possível. A computação móvel, também chamada de *computação nômade* [Kleinrock 1997], é a execução de tarefas de computação, enquanto o usuário está se deslocando de um lugar a outro ou visitando lugares diferentes de seu ambiente usual. Na computação móvel, os usuários que estão longe de suas intranets "de base" (a intranet do trabalho ou de sua residência) podem acessar recursos por intermédio dos equipamentos que carregam consigo. Eles podem continuar a acessar a Internet, os recursos em sua intranet de base e, cada vez mais, existem condições para que os usuários utilizem recursos como impressoras, que estão convenientemente próximos, enquanto transitam. Esta última possibilidade também é conhecida como *computação com reconhecimento de localização* ou *com reconhecimento de contexto*.

A *computação ubíqua* [Weiser 1993], também denominada de *computação pervasiva*, é a utilização de vários dispositivos computacionais pequenos e baratos, que estão presentes nos ambientes físicos dos usuários, incluindo suas casas, escritórios e até na rua. O termo "pervasivo" se destina a sugerir que pequenos equipamentos de computação finalmente se tornarão tão entranhados nos objetos diários que mal serão notados. Isto é, seu comportamento computacional será transparente e

intimamente vinculado à sua função física. Por sua vez, o termo "ubíquo" dá a noção que o acesso a serviços de computação está onipresente, isto é, disponível em qualquer lugar.

A presença de computadores em toda parte só se torna útil quando eles podem se comunicar uns com os outros. Por exemplo, seria conveniente para um usuário controlar sua máquina de lavar e seu aparelho de som de alta fidelidade a partir de um único dispositivo universal de controle remoto em sua casa. Da mesma forma, seria cômodo a máquina de lavar enviar uma mensagem quando a lavagem tiver terminado.

A computação ubíqua e a computação móvel se sobrepõem, pois, em princípio, o usuário móvel pode utilizar computadores que estejam em qualquer lugar. Mas, de modo geral, elas são distintas. A computação ubíqua pode ajudar os usuários enquanto eles permanecem em um determinado ambiente, como sua casa ou um hospital. Isoladamente, a computação móvel também oferece vantagens, basta imaginar a tarefa de enviar uma impressão a partir de um notebook sem nenhum tipo de conexão física (cabo).

A Figura 1.3 mostra um usuário que é visitante em uma organização anfitriã e que pode acessar serviços locais a intranet dessa organização, a Internet ou a sua intranet doméstica.

O usuário da Figura 1.3 acessa três formas diferentes de conexão sem fio. Seu notebook tem uma maneira de se conectar com a rede local sem fio da organização anfitriã. Essa rede fornece cobertura de algumas centenas de metros (um andar de um edifício, digamos) e se conecta ao restante da intranet anfitriã por meio de um *gateway*. O usuário também tem um telefone móvel (celular), que possui conexão à Internet permitindo acesso a páginas de informações simples, as quais apresenta em sua pequena tela. Finalmente, o usuário está portando uma câmera digital, que se comunica por intermédio de uma rede sem fio pessoal (com alcance de cerca de 10m) com um outro equipamento, como por exemplo, uma impressora.

Com uma infra-estrutura conveniente, o usuário pode executar algumas tarefas no site anfitrião, usando os equipamentos que carrega consigo. Enquanto está no site anfitrião, o usuário pode buscar os preços de ações mais recentes a partir de um servidor web, usando seu celular. Durante uma reunião com seus anfitriões, o usuário pode lhes mostrar uma fotografia, enviando-a a partir da câmera digital, diretamente para uma impressora adequadamente ativada na sala de reuniões. Isso exige apenas a comunicação sem fio entre a câmera e a impressora. Em princípio, eles podem enviar um documento de seus notebooks para a mesma impressora, utilizando a rede local sem fio e conexões Ethernet cabeadas com a impressora.

A computação móvel e a computação ubíqua são uma área de pesquisa ativa e serão o assunto do Capítulo 16.

Figura 1.3 Equipamentos portáteis em um sistema distribuído.

1.3 Compartilhamento de recursos e a web

Os usuários estão tão acostumados às vantagens do compartilhamento de recursos que podem ignorar seu significado facilmente. Rotineiramente, compartilhamos recursos de hardware (como impressoras), recursos de dados (como arquivos) e recursos com funcionalidade mais específica (como os mecanismos de busca).

Do ponto de vista do hardware, compartilhamos equipamentos como impressoras e discos para reduzir os custos. Mas a maior importância para os usuários é o compartilhamento dos recursos em um nível de abstração mais alto, como informações necessárias às suas aplicações, ao seu trabalho diário e em suas atividades sociais. Por exemplo, os usuários se preocupam em compartilhar informações através de um banco de dados ou de um conjunto de páginas web – e não com discos ou processadores em que eles estão armazenados. Analogamente, os usuários pensam em termos de recursos compartilhados, como um mecanismo de busca ou um conversor de moeda corrente, sem considerar o servidor (ou servidores) que os fornecem.

Na prática, os padrões de compartilhamento de recursos variam amplamente na abrangência e no quanto os usuários trabalham em conjunto. Em um extremo, temos um mecanismo de busca na web que fornece um recurso para usuários de todo o mundo, usuários estes que nunca entram em contato diretamente. No outro extremo, no *trabalho cooperativo apoiado por computador*, um grupo de usuários colabora diretamente compartilhando recursos, como documentos, em um pequeno grupo fechado. O padrão de compartilhamento e a distribuição geográfica dos usuários determinam quais mecanismos o sistema deve fornecer para coordenar as ações dos usuários.

O termo *serviço* é usado para designar uma parte distinta de um sistema de computador que gerencia um conjunto de recursos relacionados e apresenta sua funcionalidade para usuários e aplicativos. Por exemplo, acessamos arquivos compartilhados por intermédio de um serviço de sistema de arquivos; enviamos documentos para impressoras por meio de um serviço de impressão; adquirimos bens por meio de um serviço de pagamento eletrônico. O único acesso que temos ao serviço é por intermédio do conjunto de operações que ele exporta. Por exemplo, um serviço de sistema de arquivos fornece operações de *leitura*, *escrita* e *exclusão* dos arquivos.

Em parte, o fato dos serviços restringirem o acesso ao recurso a um conjunto de operações bem definidas é uma prática padrão na engenharia de software. Mas isso também reflete a organização física dos sistemas distribuídos. Em um sistema distribuído, os recursos são fisicamente encapsulados dentro dos computadores e só podem ser acessados a partir de outros computadores por intermédio de mecanismos de comunicação. Para se obter um compartilhamento eficiente, cada recurso deve ser gerenciado por um programa que ofereça uma interface de comunicação, permitindo ao recurso ser acessado e atualizado de forma confiável e consistente.

O termo *servidor* provavelmente é conhecido da maioria dos leitores. Ele se refere a um programa em execução (um *processo*) em um computador interligado em rede, que aceita pedidos de programas em execução em outros computadores para efetuar um serviço e responder apropriadamente. Os processos que realizam os pedidos são referidos como *clientes*. Os pedidos são enviados em mensagens dos clientes para um servidor e as respostas são enviadas do servidor para os clientes. Quando o cliente envia um pedido para que uma operação seja efetuada, dizemos que o cliente *requisita uma operação* no servidor. Uma interação completa entre um cliente e um servidor, desde quando o cliente envia seu pedido até o momento em que recebe a resposta do servidor, é chamada de *requisição remota*.

O mesmo processo pode ser tanto cliente como servidor, pois, às vezes, os servidores solicitam operações em outros servidores. Os termos cliente e servidor só se aplicam às funções desempenhadas em um único pedido. Os clientes são ativos e os servidores são passivos; os servidores funcionam continuamente, enquanto os clientes duram apenas enquanto os aplicativos dos quais fazem parte estão ativos.

Note que, por padrão, os termos cliente e servidor se referem a *processos* e não aos computadores em que são executados, embora no jargão comum esses termos também se refiram aos computadores em si. Outra distinção, que discutiremos no Capítulo 5, é que, em um sistema distribuído escrito em uma linguagem orientada a objetos, os recursos podem ser encapsulados como objetos e acessados por objetos clientes, nesse caso falamos em um *objeto cliente* invocando um método em um *objeto servidor*.

Muitos sistemas distribuídos (mas certamente não todos) podem ser totalmente construídos na forma de clientes e de servidores interagindo. A World Wide Web, o e-mail e impressoras interligadas em rede são exemplos que se encaixam nesse modelo. Discutiremos alternativas para os sistemas cliente-servidor no Capítulo 2.

Um navegador web em execução é um exemplo de cliente. O navegador web se comunica com um servidor web para solicitar páginas web. Agora, examinaremos a web com mais detalhes.

1.3.1 A World Wide Web

A World Wide Web [www.w3.org], Berners-Lee 1991] é um sistema em evolução para a publicação e para o acesso a recursos e serviços pela Internet. Por meio de navegadores web (*browsers*) comumente disponíveis, os usuários recuperam e vêem documentos de muitos tipos, ouvem fluxos de áudio, assistem a fluxos de vídeo e interagem com um vasto conjunto de serviços.

A web nasceu no centro europeu de pesquisa nuclear (CERN), na Suíça, em 1989, como um veículo para troca de documentos entre uma comunidade de físicos conectados pela Internet [Berners-Lee 1999]. Uma característica importante da web é que ela fornece uma estrutura de *hipertexto* entre os documentos que armazena, refletindo a necessidade dos usuários de organizar seus conhecimentos. Isso significa que os documentos contêm *links* (ou *hyperlinks*) – referências para outros documentos e recursos que também estão armazenados na web.

É fundamental para a experiência do usuário da web que, ao encontrar determinada imagem ou texto dentro de um documento, isso seja freqüentemente acompanhado de *links* para documentos e outros recursos relacionados. A estrutura de *links* pode ser arbitrariamente complexa e o conjunto de recursos que podem ser adicionados é ilimitado – a "teia" (web) de *links* é realmente mundial. Bush [1945] imaginou estruturas de hipertexto há mais de 50 anos; foi com o desenvolvimento da Internet que essa idéia pode se manifestar em escala mundial.

A web é um sistema *aberto*: ela pode ser ampliada e implementada de novas maneiras, sem perturbar a funcionalidade existente (veja a Seção 1.4.2). A operação da web é baseada em padrões de comunicação e de documentos livremente publicados. Por exemplo, existem muitos tipos de navegadores, em muitos casos, implementados em várias plataformas; e existem muitas implementações de servidores web. Qualquer navegador pode recuperar recursos de qualquer servidor desde que ambos utilizem os mesmos padrões em suas implementações, em outros termos, não importa como eles são implementados mas sim o que eles implementam. Os usuários têm acesso a navegadores na maioria dos equipamentos que utilizam, desde telefones móveis até computadores de mesa.

A web é aberta no que diz respeito aos tipos de recursos que nela podem ser publicados e compartilhados. Em sua forma mais simples, um recurso da web é uma página, ou algum outro tipo de *conteúdo*, que possa ser armazenado em um arquivo e apresentado para o usuário, como arquivos de programa, arquivos de mídia e documentos em PostScript ou Portable Document Format (pdf). Se alguém inventar, digamos, um novo formato de armazenamento de imagem, então as imagens que tenham esse formato poderão ser publicadas na web imediatamente. Os usuários necessitam de meios para ver imagens nesse novo formato, mas os navegadores são projetados de maneira a acomodar nova funcionalidade de apresentação de conteúdo, na forma de aplicativos "auxiliares" e "plug-ins".

A web já foi além desses recursos de dados simples, e hoje abrange serviços como a aquisição eletrônica de bens. Ela tem evoluído sem mudar sua arquitetura básica. A web é baseada em três componentes tecnológicos padrão principais:

- HTML (*HyperText Markup Language*) que é uma linguagem para especificar o conteúdo e o leiaute de páginas de forma que elas possam ser exibidas pelos navegadores web.
- URLs (*Uniform Resource Locators*), que identificam os documentos e outros recursos armazenados como parte da web. O Capítulo 9 discutirá outras formas de identificadores de recursos da web.
- Uma arquitetura de sistema cliente-servidor, com regras padrão para interação (o protocolo HTTP – *HyperText Transfer Protocol*), por meio das quais os navegadores e outros clientes buscam documentos e outros recursos dos servidores web. A Figura 1.4 mostra alguns nave-

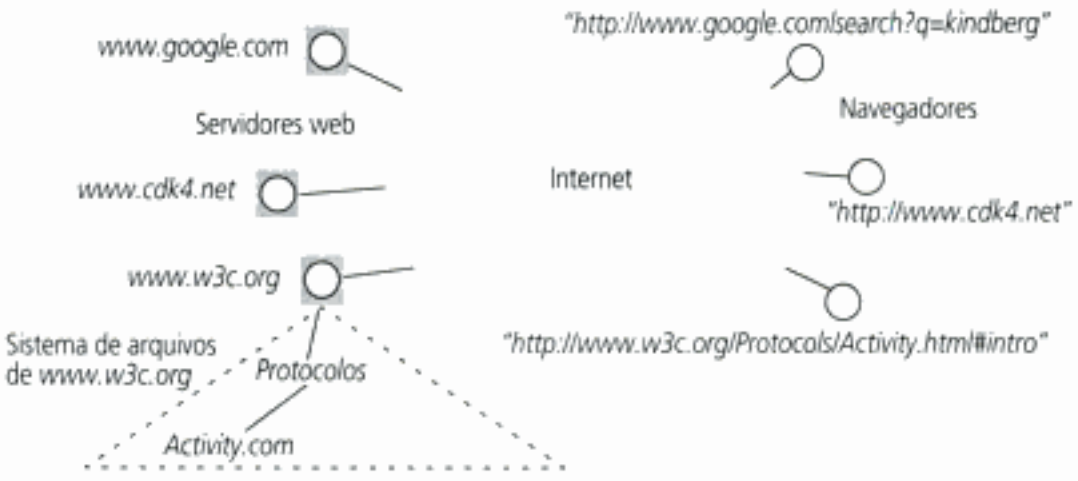


Figura 1.4 **Servidores e navegadores web.**

gadores realizando pedidos para servidores web. É uma característica importante o fato dos usuários poderem localizar e gerenciar seus próprios servidores web em qualquer parte da Internet.

Discutiremos agora cada um desses componentes e, ao fazermos isso, explicaremos o funcionamento dos navegadores e servidores web quando um usuário busca páginas web e clica nos *links* nelas existentes.

HTML ◊ A *HyperText Markup Language* [www.w3.org II] é usada para especificar o texto e as imagens que compõem o conteúdo de uma página web e para especificar como eles são dispostos e formatados para apresentação ao usuário. Uma página web contém itens estruturados como cabeçalhos, parágrafos, tabelas e imagens. A HTML também é usada para especificar *links* e os recursos que estão associados a eles.

Os usuários produzem código HTML manualmente, através de um editor de textos padrão, ou utilizando um editor *wysiwyg* (*what you see is what you get*) com reconhecimento de HTML, que gera código HTML a partir de um leiaute criado graficamente. Um texto em HTML típico aparece a seguir:

```
<IMG SRC = "http://www.cdk4.net/webExample/Images/earth.jpg">                1
<P>                                                                          2
Welcome to Earth! Visitors may also be interested in taking a look at the    3
<A HREF = "http://www.cdk4.net/webExample/moon.html">Moon</A>.               4
</P>                                                                         5
```

Esse texto em HTML é armazenado em um arquivo localizado em um lugar onde um servidor web pode lê-lo – digamos que seja o arquivo nomeado como *earth.html* gravado no disco local do próprio servidor web. Um navegador recupera o conteúdo desse arquivo a partir de um servidor web – neste caso específico, um servidor em um computador chamado *www.cdk4.net*. O navegador lê o conteúdo retornado pelo servidor e o apresenta como um texto formatado com as imagens que o compõe, dispostos em uma página web na forma em que estamos familiarizados a ver. Apenas o navegador – e não o servidor – interpreta o texto em HTML. Mas o servidor informa ao navegador sobre o tipo de conteúdo que está retornando, para distingui-lo de, digamos, um documento em PostScript. O servidor pode deduzir o tipo de conteúdo a partir da extensão de nome de arquivo .html.

Note que as diretivas HTML, conhecidas como *tags*, são incluídas entre sinais de menor e maior, como em *<P>*. A linha 1 do exemplo identifica um arquivo contendo uma imagem de apresentação. Seu URL é *http://www.cdk4.net/webExample/Images/earth.jpg*. As linhas 2 e 5 possuem as diretivas para iniciar e terminar um parágrafo, respectivamente. As linhas 3 e 4 contêm o texto a ser exibido na página web, no formato padrão de parágrafo.

A linha 4 especifica um *link* na página web. Ele contém a palavra *Moon*, circundada por duas diretivas HTML relacionadas *<A HREF...>* e *</A>*. O texto entre essas diretivas é o que aparece no

link, quando ele é apresentado na página web. Por padrão, a maioria dos navegadores é configurada de modo a mostrar o texto de *links* sublinhado; portanto, o que o usuário verá nesse parágrafo será:

Welcome to Earth! Visitors may also be interested in taking a look at the Moon.

O navegador grava a associação entre o texto exibido do *link* e o URL contido na diretiva *<A HREF...>* – neste caso:

http://www.cdk4.net/webExample/moon.html

Quando o usuário clica no texto, o navegador recupera o recurso identificado pelo URL correspondente e o apresenta para o usuário. No exemplo, o recurso está em um arquivo HTML que especifica uma página web sobre a lua.

URLs ◊ O objetivo de um URL (*Uniform Resource Locator*) [www.w3.org III] é identificar um recurso. Na verdade, o termo usado em documentos de arquitetura da web é URI (*Uniform Resource Identifier*), mas neste livro o termo mais conhecido, URL, será usado quando não houver dúvida possível. Os navegadores examinam os URLs para acessar os recursos correspondentes. Os URLs são fornecidos por um usuário no momento em que ele o digita no navegador web, porém, as situações mais comuns são aquelas em que o navegador pesquisa o URL correspondente quando o usuário clica em um *link*, quando seleciona um URL de sua lista de *bookmarks*, ou quando o navegador busca um recurso incorporado em uma página web, como uma imagem.

Todo URL, em sua forma completa e absoluta, tem dois componentes de nível superior:

esquema : identificador-específico-do-esquema

O primeiro componente, o "esquema", declara qual é o tipo desse URL. Os URLs são obrigados a identificar uma variedade de recursos. Por exemplo, *mailto:joe@anISP.net* identifica o endereço de e-mail de um usuário; *ftp://ftp.downloadIt.com/software/aProg.exe* identifica um arquivo que deve ser recuperado com o protocolo FTP (*File Transfer Protocol*), em vez do protocolo mais comumente usado, HTTP. Outros exemplos de esquemas são "nntp" (usado para especificar um grupo de notícias da Usenet) e "mid" (usado para identificar uma mensagem de e-mail).

A web não tem restrições com relação aos tipos de recursos que pode usar para acesso, graças aos designadores de esquema presentes nos URLs. Se alguém inventar um novo tipo de recurso, por exemplo, *widget* – talvez com seu próprio esquema de endereçamento para localizar elementos em uma janela gráfica e seu próprio protocolo para acessá-los –, então o mundo poderá começar a usar URLs da forma *widget:...* . É claro que os navegadores devem ter a capacidade de usar o novo protocolo *widget*, mas isso pode ser feito pela adição de um *plug-in*.

Os URLs com um designador de esquema do tipo HTTP são mais comuns para acessar recursos web. Um URL HTTP tem duas tarefas principais a executar: identificar qual servidor web mantém o recurso e qual dos recursos está sendo solicitado a esse servidor. A Figura 1.4 mostra três navegadores fazendo pedidos de recursos, gerenciados por três servidores web. O navegador que está mais acima está fazendo uma consulta em um mecanismo de busca. O navegador do meio solicita a página padrão de um outro site web. O navegador que está mais abaixo solicita uma página web especificada por completo, incluindo um nome de caminho relativo para o servidor. Os arquivos de determinado servidor web são mantidos em uma ou mais subárvores (diretórios) do sistema de arquivos desse servidor e cada recurso é identificado por um nome e um caminho relativo (*pathname*) nesse servidor.

Em geral, os URLs HTTP são da seguinte forma:

http:// nomedoservidor [:porta] [/nomedeCaminho] [?consulta][#fragmento]

– onde os itens entre colchetes são opcionais. Um URL HTTP completo sempre começa com o string http://, seguido de um nome de servidor, expresso como um nome DNS (*Domain Name System*) (veja a Seção 9.2). O nome DNS do servidor é, opcionalmente, seguido do número da porta em que o servidor recebe os pedidos (veja o Capítulo 4) – que, por padrão, é a porta 80. Em seguida, aparece um nome de caminho opcional do recurso no servidor. Se ele estiver ausente, então a página web padrão do servidor será solicitada. Finalmente, o URL termina, também opcionalmente, com um componente de consulta – por exemplo, quando um usuário envia entradas de um formulário, como a página de

consulta de um mecanismo de busca – e/ou um identificador de fragmento, que identifica um componente de um determinado recurso.

Considere os URLs a seguir:

http://www.cdk4.net
http://www.w3.org/Protocols/Activity.htm# intro
http://www.google.com/search?q=kindberg

Eles podem ser subdivididos, como segue:

Nome DNS do servidor	*Nome de caminho*	*Consulta*	*Fragmento*
www.cdk4.net	(padrão)	(nenhuma)	(nenhum)
www.w3.org	Protocols/Activity.html	(nenhuma)	intro
www.google.com	search	q=kindberg	(nenhum)

O primeiro URL designa a página padrão fornecida por *www.cdk4.net*. O seguinte identifica um fragmento de um arquivo HTML cujo nome de caminho é *Protocols/Activity.HTML*, relativo ao servidor *www.w3.org*. O identificador do fragmento (especificado após o caractere # no URL) é *intro* e um navegador procurará esse identificador de fragmento dentro do texto HTML, após ter feito o *download* do arquivo inteiro. O terceiro URL especifica uma consulta para um mecanismo de busca. O caminho identifica um programa chamado search e o string após o caractere ? codifica um string de consulta fornecido como argumento para esse programa.

Publicando um recurso: Embora a web tenha um modelo claramente definido para acessar um recurso a partir de seu URL, os métodos exatos para publicação de recursos dependem da implementação do servidor. O método mais simples de publicação de um recurso na web é colocar o arquivo correspondente em um diretório que o servidor web possa acessar. Sabendo o nome do servidor, *S*, e um nome de caminho para o arquivo, *C*, que o servidor possa reconhecer, o usuário constrói o URL como *http://S/C*. O usuário coloca esse URL em um *link* de um documento já existente ou informa esse URL para outros usuários de diversas formas, como por exemplo, por e-mail.

Existem certas convenções de nome de caminho que os servidores reconhecem. Por exemplo, um nome de caminho começando com *~joe* está, por convenção, em um subdiretório *public_html* do diretório *home* do usuário *joe*. Analogamente, um nome de caminho que termina em um nome de diretório, em vez de terminar em um arquivo simples, convencionalmente se refere a um arquivo nesse diretório, chamado *index.html*.

Huang *et al.* [2000] fornece um modelo para inserir conteúdo na web com mínima intervenção humana. Isso é particularmente relevante quando os usuários precisam extrair conteúdo de uma variedade de equipamentos, como câmeras, para publicação em páginas web.

HTTP ◊ O protocolo *HyperText Transfer Protocol* [www.w3.org IV] define as maneiras pelas quais os navegadores e outros tipos de cliente interagem com os servidores web. O Capítulo 4 examinará o protocolo HTTP com mais detalhes, mas destacaremos aqui suas principais características (restringindo nossa discussão à recuperação de referências em arquivos):

Interações requisição-resposta: o protocolo HTTP é do tipo requisição-resposta. O cliente envia uma mensagem de requisição para o servidor, contendo o URL do recurso solicitado. O servidor pesquisa o nome de caminho e, se ele existir, envia de volta para o cliente o conteúdo do arquivo em uma mensagem de resposta. Caso contrário, ele envia de volta uma resposta de erro, como a conhecida 404 Not Found.

Tipos de conteúdo: os navegadores não são necessariamente capazes de manipular todo tipo de conteúdo. Quando um navegador faz uma requisição, ele inclui uma lista dos tipos de conteúdo que prefere – por exemplo, em princípio, ele poderia exibir imagens no formato GIF, mas não no formato JPEG. O servidor poderá levar isso em conta ao retornar conteúdo para o navegador. O servidor inclui o tipo de conteúdo na mensagem de resposta para que o navegador saiba como

processá-lo. Os strings que denotam o tipo de conteúdo são chamados de tipos MIME e estão padronizados no RFC 1521 [Freed e Borenstein 1996]. Por exemplo, se o conteúdo for de tipo text/html, então um navegador interpretará o texto como HTML e o exibirá; se o conteúdo for de tipo image/GIF,o navegador o representará como uma imagem no formato GIF; se for do tipo application/zip, então são dados compactados no formato zip e o navegador ativará um aplicativo auxiliar externo para descompactá-lo. O conjunto de ações a serem executadas pelo navegador para determinado tipo de conteúdo pode ser configurado e os leitores devem verificar essas configurações em seus próprios navegadores.

Um recurso por requisição: os clientes especificam um recurso por requisição HTTP. Se uma página web contém, digamos, nove imagens, o navegador emitirá um total de 10 requisições separadas para obter o conteúdo inteiro da página. Normalmente, os navegadores fazem vários pedidos concorrentes, para reduzir o atraso global para o usuário.

Controle de acesso simplificado: por padrão, qualquer usuário com conectividade de rede para um servidor web pode acessar qualquer um de seus recursos publicados. Se for necessário restringir o acesso a determinados recursos, isso pode ser feito configurando o servidor de modo a emitir um pedido de identificação para qualquer cliente que o solicite. Então, o usuário correspondente precisa provar que tem direito de acessar o recurso, por exemplo, digitando uma senha.

Páginas dinâmicas ◊ Até aqui, descrevemos como os usuários podem publicar páginas web e outros tipos de conteúdos na web. Entretanto, grande parte da experiência dos usuários na web é a interação com serviços, em vez da simples recuperação de informações. Por exemplo, ao adquirir um item em uma loja *on-line*, o usuário freqüentemente preenche um *formulário* para fornecer seus detalhes pessoais ou para especificar exatamente o que deseja adquirir. Um formulário web é uma página contendo instruções para o usuário e elementos de janela para entrada de dados, como campos de texto e caixas de seleção. Quando o usuário envia o formulário (normalmente, clicando sobre um botão no próprio formulário ou pressionando a tecla *return*), o navegador envia um pedido HTTP para um servidor web, contendo os valores inseridos pelo usuário.

Como o resultado do pedido depende da entrada do usuário, o servidor precisa *processar* a entrada do usuário. Portanto, o URL, ou seu componente inicial, designa um *programa* no servidor e não um arquivo. Se os dados de entrada fornecidos pelo usuário forem razoavelmente curtos, então, normalmente eles serão enviados como o componente de consulta do URL, após um caractere ?, caso contrário, eles serão enviados a parte. Por exemplo, um pedido contendo o URL a seguir ativa um programa chamado search no endereço www.google.com e especifica o string de consulta kindberg: *http://www.google.com/search?q=kindberg*.

Esse programa search produz texto em HTML na saída e o usuário verá uma listagem das páginas que contêm a palavra kindberg. (O leitor pode inserir uma consulta em seu mecanismo de busca predileto e observar o URL exibido pelo navegador, quando o resultado for retornado.) O servidor retorna o texto em HTML gerado pelo programa, exatamente como se tivesse sido recuperado de um arquivo. Em outras palavras, a diferença entre o conteúdo estático recuperado a partir de um arquivo e o conteúdo gerado dinamicamente é transparente para o navegador.

Um programa que os servidores web executam para gerar conteúdo para seus clientes é freqüentemente denominado como programa CGI (*Common Gateway Interface*). Um programa CGI pode ter qualquer funcionalidade específica do aplicativo, desde que possa analisar os argumentos fornecidos pelo cliente e produzir conteúdo do tipo solicitado (normalmente, texto HTML). Freqüentemente, o programa consultará ou atualizará um banco de dados no processamento do pedido.

Código carregado por *download*: um programa CGI é executado no servidor. Às vezes, os projetistas de serviços web exigem que algum código relacionado ao serviço seja executado pelo navegador no computador do usuário. Por exemplo, freqüentemente, junto com um formulário web, é feito o *download* de um código escrito em Javascript [www.netscape.com] para proporcionar uma interação de melhor qualidade com o usuário, em vez daquela suportada pelos elementos de janela padrão do protocolo HTML. Uma página aprimorada com Javascript pode dar ao usuário retorno imediato sobre entradas inválidas (em vez de obrigar o usuário a verificar os valores no servidor, o que seria muito

mais demorado). O código Javascript pode ser usado para atualizar partes do conteúdo de uma página web, sem a busca de uma nova versão inteira da página e sem sua reapresentação.

A linguagem Javascript tem funcionalidade bastante limitada. Em comparação, um *applet* é um aplicativo cujo *download* o navegador faz automaticamente e executa quando busca uma página web correspondente. Os *applets* podem acessar a rede e fornecer interfaces personalizadas com o usuário, usando recursos da linguagem Java [java.sun.com, Flanagan 2002]. Por exemplo, às vezes, os aplicativos de bate-papo são implementados como *applets* executados nos navegadores dos usuários, junto com um programa servidor. Nesse caso, os *applets* enviam o texto de um usuário para o servidor, o qual, por sua vez, o redistribui para os demais *applets* para serem apresentados aos outros usuários. Discutiremos os *applets* com mais detalhes na Seção 2.2.3.

Serviços web (*web services*) ◊ Até aqui, discutimos a web do ponto de vista de um usuário operando um navegador. Mas outros programas, além dos navegadores, também podem ser clientes web; na verdade, o acesso por meio de programas aos recursos da web é muito comum.

Entretanto, sozinhos, os padrões HTML e HTTP são insuficientes para realizar interações por meio de programas. Primeiramente, há uma necessidade cada vez maior na web de trocar dados de tipos estruturados, mas o protocolo HTML não possui essa capacidade, ele é limitado a realizar a navegação em informações. O protocolo HTML tem um conjunto estático de estruturas, como parágrafos, e elas são restritas nas maneiras como os dados devem ser apresentados para os usuários. A linguagem XML (*Extensible Markup Language*) (ver a Seção 4.3.3) foi projetada como um modo de representar dados em formas padronizadas, estruturadas e específicas de um determinado aplicativo. Por exemplo, a XML pode ser usada para descrever os recursos de dispositivos e para descrever informações pessoais mantidas sobre os usuários. A XML é uma metalinguagem para descrever dados, a qual torna os dados portáveis entre aplicativos.

Segundo, o protocolo HTTP não fornece estrutura alguma para estabelecer as operações específicas do serviço que podem ser solicitadas a recursos web, nem os argumentos e respostas de erro das operações. Por exemplo, na loja virtual do endereço amazon.com, as operações do serviço web incluem uma ação para pedir um livro e outra ação para verificar o estado atual de um pedido. O reverso dessa flexibilidade pode ser a falta de robustez no modo como o software funciona. O Capítulo 19 examinará a estrutura dos serviços web com profundidade, a qual permite aos projetistas desses serviços especificar para os programadores exatamente como os clientes devem acessá-los.

Discussão sobre a web ◊ O sucesso fenomenal da web baseia-se na relativa facilidade com que muitas fontes individuais e organizacionais podem publicar recursos, na conveniência de sua estrutura de hipertexto para organizar muitos tipos de informação e o fato de sua arquitetura ser um sistema aberto. Os padrões nos quais sua arquitetura está baseada são simples e foram amplamente publicados em um estágio precoce. Eles têm permitido a integração de muitos tipos novos de recursos e serviços.

O sucesso da web esconde alguns problemas de projeto. Primeiramente, seu modelo de hipertexto é deficiente sob alguns aspectos. Se um recurso é excluído ou movido, os assim chamados *links* "pendentes" para esse recurso ainda podem permanecer, causando frustração para os usuários. E há o conhecido problema de usuários "perdidos no hiperespaço". Freqüentemente, os usuários ficam confusos, seguindo muitos *links* distintos, referenciando páginas de um conjunto de fontes discrepantes e de confiabilidade duvidosa, em alguns casos.

Os mecanismos de busca são uma alternativa para localizar informações na web, mas eles são sabidamente imperfeitos na produção do que o usuário pretende especificamente. Um modo de tratar desse problema, exemplificado no *Resource Description Framework* [www.w3.org V], é produzir vocabulários, sintaxe e semântica padrões para expressar metadados sobre as coisas de nosso mundo e encapsular esses metadados nos recursos web correspondentes. Em vez de pesquisar palavras que ocorrem em páginas web, os programas de busca podem então, em princípio, com base na correspondência semântica, realizar pesquisas nos metadados para compilar listas de *links* relacionados. Coletivamente, esse emaranhado de recursos de metadados vinculados é o que é conhecido como *web semântica*.

Como uma arquitetura de sistema, a web enfrenta problemas de escalabilidade. Os servidores web mais populares podem ter muitos acessos (*hits*) por segundo e, como resultado, a resposta para os usuários pode ser lenta. O Capítulo 2 descreverá o uso de cache em navegadores e de servidores *proxies* para melhorar o tempo de resposta e a divisão da carga de processamento por um grupo de

computadores. A arquitetura cliente-servidor web implica em que ela não tem nenhum meio eficiente de manter os usuários atualizados com as versões mais recentes das páginas. Os usuários precisam explicitamente fazer uma operação de atualizar em seus navegadores para terem certeza de que possuem as informações mais recentes. Nesse caso, os navegadores são obrigados a se comunicar com os servidores web para verificar se a cópia local que possuem de um recurso ainda é válida ou se os servidores possuem uma versão mais recente.

Finalmente, uma página web nem sempre possui uma interface satisfatória com o usuário. Os elementos de janela da interface definidos pela HTML são limitados e os projetistas freqüentemente incluem nas páginas web *applets* e/ou muitas imagens para fazê-las parecer e funcionar de modo mais aceitável. Com isso, há um conseqüente aumento no tempo de *download*.

1.4 Desafios

Os exemplos da Seção 1.2 ilustram a abrangência dos sistemas distribuídos e os problemas que surgem em seu projeto. Embora os sistemas distribuídos sejam encontrados em toda parte, seu projeto é muito simples e ainda há muito espaço para o desenvolvimento de serviços e de aplicativos mais ambiciosos. Muitos dos desafios discutidos nesta seção já foram resolvidos, mas os futuros projetistas precisam conhecê-los e tomar o cuidado de levá-los em consideração.

1.4.1 Heterogeneidade

A Internet permite aos usuários acessarem serviços e executarem aplicativos por meio de um conjunto heterogêneo de computadores e redes. A heterogeneidade (isto é, variedade e diferença) se aplica aos seguintes aspectos:

- redes
- hardware de computador
- sistemas operacionais
- linguagens de programação
- implementações de diferentes desenvolvedores

Embora a Internet seja composta de muitos tipos de rede (ilustrado na Figura 1.1), suas diferenças são mascaradas pelo fato de que todos os computadores ligados a elas utilizam protocolos Internet para se comunicar. Por exemplo, um computador que possui uma placa Ethernet tem uma implementação dos protocolos Internet enquanto um computador em um tipo diferente de rede tem uma implementação dos protocolos Internet para essa rede. O Capítulo 3 explicará como os protocolos Internet são implementados em uma variedade de redes diferentes.

Os tipos de dados, como os inteiros, podem ser representados de diversas maneiras em diferentes tipos de hardware; por exemplo, existem duas alternativas para a ordem em que os bytes de valores inteiros são armazenados: uma iniciando a partir do byte mais significativo e outra a partir do byte menos significativo. Essas diferenças na representação devem ser consideradas, caso mensagens devam ser trocadas entre programas sendo executados em diferentes hardwares.

Embora os sistemas operacionais de todos os computadores na Internet precisem incluir uma implementação dos protocolos Internet, nem todos fornecem necessariamente a mesma interface de programação de aplicativos para esses protocolos. Por exemplo, as chamadas para troca de mensagens no UNIX são diferentes das chamadas no Windows.

Diferentes linguagens de programação usam diferentes representações para caracteres e estruturas de dados, como arrays e registros. Essas diferenças devem ser consideradas, caso programas escritos em diferentes linguagens precisem se comunicar.

Os programas escritos por diferentes desenvolvedores não podem se comunicar, a menos que eles utilizem padrões comuns; por exemplo, para realizar a comunicação via rede e usar uma mesma re-

presentação de tipos de dados primitivos e estruturas de dados nas mensagens. Para que isso aconteça, padrões precisam ser estabelecidos e adotados. Assim é o caso dos protocolos Internet.

Middleware ◊ O termo *middleware* se aplica a uma camada de software que fornece uma abstração de programação, assim como o mascaramento da heterogeneidade das redes, do hardware, de sistemas operacionais e linguagens de programação subjacentes. O CORBA (*Common Object Request Broker*), que será descrito nos Capítulos 4, 5 e 20, é um exemplo. Alguns *middlewares*, como o Java RMI (*Remote Method Invocation*) (veja o Capítulo 5), suportam apenas uma linguagem de programação. A maioria é implementada sobre os protocolos Internet, os quais escondem a diferença existente entre redes subjacentes. Todo *middleware*, em si, trata das diferenças em nível dos sistemas operacionais e do hardware – o modo como isso é feito será o tópico principal do Capítulo 4.

Além de resolver os problemas de heterogeneidade, o *middleware* fornece um modelo computacional uniforme para ser usado pelos programadores de serviços e de aplicativos distribuídos. Os modelos possíveis incluem a invocação remota de objetos, a notificação remota de eventos, o acesso remoto a banco de dados e o processamento de transação distribuído. Por exemplo, o CORBA fornece invocação remota de objetos, a qual permite que um objeto, em um programa sendo executado em um computador, invoque um método de um objeto em um programa executado em outro computador. Sua implementação oculta o fato de que as mensagens passam por uma rede para enviar o pedido de invocação e sua resposta.

Heterogeneidade e migração de código ◊ O termo *migração de código*, ou ainda, *código móvel*, é usado para se referir ao código que pode ser enviado de um computador para outro e ser executado no destino – os *applets* Java são um exemplo. Um código destinado a execução em um computador não é necessariamente adequado para outro computador, pois, normalmente, os programas executáveis são específicos a um conjunto de instruções e a um sistema operacional. Por exemplo, arquivos executáveis enviados como anexos de e-mail por usuários de Windows/x86 não funcionarão em um computador x86 executando Linux ou em um computador Macintosh executando Mac OS X.

A estratégia de *máquina virtual* oferece uma maneira de tornar um código executável em qualquer tipo de processador (hardware) e sistema operacional. Nessa estratégia, o compilador de uma linguagem em particular gera código para uma máquina virtual, em vez de código para um processador e sistema operacional específicos; por exemplo, o compilador Java produz código para a máquina virtual Java (*Java Virtual Machine* – JVM). Para permitir a execução de programas Java, é necessário ter-se uma versão da JVM implementada para o processador e sistema operacional da máquina alvo. Entretanto, a solução Java não é aplicável de forma generalizada aos programas escritos em outras linguagens.

1.4.2 Sistemas abertos

Diz-se que um sistema computacional é aberto quando ele pode ser estendido e reimplementado de várias maneiras. O fato de um sistema distribuído ser ou não um sistema aberto é determinado principalmente pelo grau com que novos serviços podem ser adicionados e disponibilizados para uso por uma variedade de programas clientes.

A característica de sistema aberto é obtida a partir do momento em que a especificação e a documentação das principais interfaces de software dos componentes de um sistema estão disponíveis para os desenvolvedores de software. Em uma palavra, as principais interfaces são *publicadas*. Esse processo é similar àquele realizado por organizações de padronização, porém, freqüentemente, ignora os procedimentos oficiais, os quais normalmente são pesados e lentos.

Entretanto, a publicação de interfaces é apenas o ponto de partida para adicionar e estender serviços em um sistema distribuído. O maior desafio para os projetistas é encarar a complexidade de sistemas distribuídos compostos por muitos componentes e elaborados por diferentes pessoas.

Os projetistas dos protocolos Internet elaboraram uma série de documentos, chamados *Requests For Comments* ou RFCs, cada um dos quais identificado por um número. No início dos anos 80, as especificações dos protocolos de comunicação Internet foram publicadas nessa série, acompanhadas de especificações de aplicativos com eles executados, como a transferência de arquivos, e-mail e telnet. Essa prática continua e forma a base da documentação técnica da Internet. Essa série inclui

discussões, assim como as especificações dos protocolos (pode-se obter cópias dos RFCs no endereço [www.ietf.org]). Dessa forma, com a publicação dos protocolos de comunicação Internet, permitiu-se a construção de uma variedade de sistemas e aplicativos para a Internet, incluindo a web. Os RFCs não são os únicos meios de publicação. Por exemplo, o CORBA é publicado por intermédio de uma série de documentos técnicos, incluindo uma especificação completa das interfaces de seus serviços (veja [www.omg.org]).

Os sistemas projetados a partir de padrões públicos são chamados de *sistemas distribuídos abertos*, para reforçar o fato de que eles são extensíveis. Eles podem ser ampliados em nível de hardware, pela adição de computadores em uma rede, e em nível de software, pela introdução de novos serviços ou pela reimplementação dos antigos, permitindo aos programas aplicativos compartilharem recursos. Uma vantagem adicional, freqüentemente mencionada para sistemas abertos, é sua independência de fornecedores individuais.

Resumindo:

- Os sistemas abertos são caracterizados pelo fato de suas principais interfaces serem publicadas.
- Os sistemas distribuídos abertos são baseados na estipulação de um mecanismo de comunicação uniforme e em interfaces publicadas para acesso aos recursos compartilhados.
- Os sistemas distribuídos abertos podem ser construídos a partir de hardware e software heterogêneo, possivelmente de diferentes fornecedores. Para que um sistema funcione corretamente, a compatibilidade de cada componente com o padrão publicado deve ser cuidadosamente testada e verificada.

1.4.3 Segurança

Muitos recursos de informação que se tornam disponíveis e são mantidos em sistemas distribuídos têm um alto valor intrínseco para seus usuários. Portanto, sua segurança é de considerável importância. A segurança de recursos de informação tem três componentes: confidencialidade (proteção contra exposição para pessoas não autorizadas), integridade (proteção contra alteração ou dano) e disponibilidade (proteção contra interferência com os meios de acesso aos recursos).

A Seção 1.1 mostrou que, embora a Internet permita que um programa em um computador se comunique com um programa em outro computador, independentemente de sua localização, existem riscos de segurança associados ao livre acesso a todos os recursos em uma intranet. Embora um *firewall* possa ser usado para formar uma barreira em torno de uma intranet, restringindo o tráfego que pode entrar ou sair, isso não garante o uso apropriado dos recursos pelos usuários de dentro da intranet, nem o uso apropriado dos recursos na Internet, que não são protegidos por *firewalls*.

Em um sistema distribuído, os clientes enviam pedidos para acessar dados gerenciados por servidores, o que envolve o envio de informações em mensagens por uma rede. Por exemplo:

1. Um médico poderia solicitar acesso aos dados dos pacientes de um hospital ou enviar mais informações sobre esses pacientes.
2. No comércio eletrônico e nos serviços bancários, os usuários enviam seus números de cartão de crédito pela Internet.

Nesses dois exemplos, o desafio é enviar informações sigilosas em uma ou mais mensagens, por uma rede, de maneira segura. Mas a segurança não é apenas uma questão de ocultar o conteúdo de mensagens – ela também envolve saber com certeza a identidade do usuário, ou outro agente, em nome de quem uma mensagem foi enviada. No primeiro exemplo, o servidor precisa saber se o usuário é realmente um médico e, no segundo exemplo, o usuário precisa ter certeza da identidade da loja ou do banco com o qual está tratando. O segundo desafio aqui é identificar corretamente um usuário ou outro agente remoto. Esses dois desafios podem ser resolvidos com o uso de técnicas de criptografia desenvolvidas para esse propósito. Elas são amplamente usadas na Internet e serão discutidas no Capítulo 7.

Entretanto, dois desafios de segurança, descritos a seguir, ainda não foram totalmente resolvidos:

Ataque de negação de serviço (*Denial of Service*): ocorre quando um usuário interrompe um serviço por algum motivo. Isso pode ser conseguido bombardeando o serviço com um número tão

grande de pedidos sem sentido, que os usuários sérios não são capazes de usá-lo. Isso é chamado de *ataque de negação de serviço*. De tempos em tempos, surgem ataques de negação de serviços contra alguns serviços e servidores web bem conhecidos. Atualmente, tais ataques são controlados pela tentativa de capturar e punir os responsáveis após o evento, mas essa não é uma solução geral para o problema. Medidas para se opor a isso, baseadas em melhorias no gerenciamento das redes, estão sendo desenvolvidas e isso será um assunto do Capítulo 3.

Segurança de código móvel: um código móvel precisa ser manipulado com cuidado. Considere alguém que receba um programa executável como um anexo de correio eletrônico: os possíveis efeitos da execução do programa são imprevisíveis; por exemplo, pode parecer que ele apenas exiba uma figura interessante, mas, na realidade, está acessando recursos locais ou, talvez, fazendo parte de um ataque de negação de serviço. Algumas medidas para tornar seguro um código móvel serão esboçadas no Capítulo 7.

1.4.4 Escalabilidade

Os sistemas distribuídos funcionam de forma efetiva e eficaz em muitas escalas diferentes, variando desde uma pequena intranet até a Internet. Um sistema é descrito como *escalável* se permanece eficiente quando há um aumento significativo no número de recursos e no número de usuários. A Internet é um exemplo de um sistema distribuído no qual o número de computadores e serviços vem aumentando substancialmente. A Figura 1.5 mostra o aumento no número de computadores na Internet durante 24 anos, até 2003, e a Figura 1.6 mostra o número cada vez maior de computadores e servidores web durante os 10 anos de história da web (veja [zakon.org]).

O projeto de sistemas distribuídos escaláveis apresenta os seguintes desafios:

Controlar o custo dos recursos físicos: à medida que a demanda por um recurso aumenta, deve ser possível, a um custo razoável, ampliar o sistema para atendê-la. Por exemplo, a freqüência com que os arquivos são acessados em uma intranet provavelmente vai crescer à medida que o número de usuários e de computadores aumentar. Deve ser possível adicionar servidores de arquivos de forma a evitar o gargalo de desempenho que haveria caso um único servidor de arquivos tivesse que tratar todos os pedidos de acesso a arquivos. Em geral, para que um sistema com n usuários seja escalável, a quantidade de recursos físicos exigida para suportá-los deve ser no máximo $O(n)$ – isto é, proporcional a n. Por exemplo, se um único servidor de arquivos pode suportar 20 usuários, então dois servidores deverão suportar 40 usuários. Embora isso pareça um objetivo óbvio, como mostraremos no Capítulo 8, não é necessariamente fácil de atingi-lo.

Controlar a perda de desempenho: considere o gerenciamento de um conjunto de dados, cujo tamanho é proporcional ao número de usuários ou recursos presentes no sistema; por exemplo, a tabela de correspondência entre os nomes de domínio dos computadores e seus endereços IP mantidos pelo *Domain Name System* (DNS), que é usado principalmente para pesquisar nomes, como www.amazon.com. Os algoritmos que utilizam estruturas hierárquicas têm melhor escalabilidade do que aqueles que usam estruturas lineares. Mas, mesmo com as estruturas hierárquicas, um aumento no tamanho resultará em alguma perda de desempenho: o tempo que leva para acessar dados hierarquicamente estruturados é $O(log\ n)$, onde n é o tamanho do conjunto de dados. Para que um sistema seja escalável, a perda de desempenho máxima não deve ser maior do que isso.

Data	*Computadores*	*Servidores web*
Dez. de 1979	188	0
Julho de 1989	130.000	0
Julho de 1999	56.218.000	5.560.866
Jan. de 2003	171.638.297	35.424.956

Figura 1.5 Computadores (com endereços IP registrados) na Internet.

Data	*Computadores*	*Servidores web*	*Porcentagem*
Julho de 1993	1.776.000	130	0,008
Julho de 1995	6.642.000	23.500	0,4
Julho de 1997	19.540.000	1.203.096	6
Julho de 1999	56.218.000	6.598.697	12
Julho de 2001	125.888.197	31.299.592	25
Julho de 2003		42.298.371	

Um servidor web pode estar hospedado em vários sites.

Figura 1.6 Computadores *versus* servidores web na Internet.

Impedir que os recursos de software se esgotem: um exemplo de falta de escalabilidade é mostrado pelos números usados como endereços IP (endereços de computador na Internet). No final dos anos 70, decidiu-se usar 32 bits para esse propósito, mas, conforme será explicado no Capítulo 3, a quantidade de endereços IP disponíveis está se esgotando. Por isso, uma nova versão do protocolo, com endereços de IP em 128 bits, está sendo adotada e isso exigirá modificações em muitos componentes de software. Para sermos justos com os primeiros projetistas da Internet, não há uma solução correta para esse problema. É difícil prever, com anos de antecedência, a demanda que será imposta sobre um sistema. Além disso, superestimar o crescimento futuro pode ser pior do que adaptar para uma mudança quando formos obrigados a isso – por exemplo, endereços IP maiores ocupam espaço extra no armazenamento de mensagens e no computador.

Evitar gargalos de desempenho: em geral, os algoritmos devem ser descentralizados para evitar a existência de gargalos de desempenho. Ilustramos esse ponto com referência ao predecessor do *Domain Name System*, no qual a tabela de correspondência entre endereços IP e nomes era mantida em um único arquivo central, cujo *download* podia ser feito em qualquer computador que precisasse dela. Isso funcionava bem quando havia apenas algumas centenas de computadores na Internet, mas logo se tornou um sério gargalo de desempenho e administrativo. Com os milhares de computadores na Internet, imagine o tamanho e o acesso a esse arquivo central. O *Domain Name System* eliminou esse gargalo particionando a tabela de correspondência de nomes entre diversos servidores localizados em toda a Internet e administrados de forma local – veja os Capítulos 3 e 9.

Alguns recursos compartilhados são acessados com muita freqüência; por exemplo, muitos usuários podem acessar uma mesma página web, causando uma queda no desempenho. No Capítulo 2, veremos que o uso de cache e replicação melhora o desempenho de recursos que são pesadamente utilizados.

De preferência, o software de sistema e de aplicativo não deve mudar quando a escala do sistema aumentar, mas isso é difícil de conseguir. O problema da escala é um tema central no desenvolvimento de sistemas distribuídos. As técnicas que têm obtido sucesso serão discutidas extensivamente neste livro. Elas incluem o uso de dados replicados (Capítulo 15), a técnica associada de uso de cache (Capítulos 2 e 8) e a distribuição de vários servidores para manipular as tarefas comumente executadas, permitindo que várias tarefas semelhantes sejam executadas concorrentemente.

1.4.5 Tratamento de falhas

Às vezes, os sistemas de computador falham. Quando ocorrem falhas no hardware ou no software, os programas podem produzir resultados incorretos ou podem parar antes de terem concluído a computação pretendida. No Capítulo 2, discutiremos e classificaremos uma variedade de tipos de falhas possíveis, que podem ocorrer nos processos e nas redes que compõem um sistema distribuído.

As falhas em um sistema distribuído são parciais – isto é, alguns componentes falham, enquanto outros continuam funcionando. Portanto, o tratamento de falhas é particularmente difícil. As técnicas a seguir para tratamento de falhas serão discutidas por todo o livro:

Detecção de falhas: algumas falhas podem ser detectadas. Por exemplo, somas de verificação podem ser usadas para detectar dados danificados em uma mensagem ou em um arquivo. O Capítulo 2 explicará que é difícil, ou mesmo impossível, detectar algumas outras falhas, como um servidor remoto danificado na Internet. O desafio é como gerenciar a ocorrência de falhas que não podem ser detectadas, mas que podem ser suspeitas.

Mascaramento de falhas: algumas falhas detectadas podem ser ocultas ou se tornar menos sérias. Dois exemplos de ocultação de falhas:

1. mensagens podem ser retransmitidas quando não chegam;
2. dados de arquivos podem ser gravados em dois discos, para que, se um estiver danificado, o outro ainda possa estar correto.

Simplesmente eliminar uma mensagem danificada é um exemplo de como tornar uma falha menos séria – ela pode ser retransmitida. O leitor provavelmente perceberá que as técnicas descritas para o mascaramento de falhas podem não funcionar nos piores casos; por exemplo, os dados no segundo disco também podem estar danificados ou a mensagem pode não chegar em um tempo razoável e, contudo, ser retransmitida freqüentemente.

Tolerância a falhas: a maioria dos serviços na Internet apresenta falhas – não seria prático para eles tentar detectar e mascarar tudo que possa ocorrer em uma rede grande assim, com tantos componentes. Seus clientes podem ser projetados de forma a tolerar falhas, o que geralmente envolve a tolerância também por parte dos usuários. Por exemplo, quando um navegador não consegue contatar um servidor web, ele não faz o usuário esperar indefinidamente, enquanto continua tentando – ele informa o usuário sobre o problema, deixando-o livre para tentar novamente. Os serviços que toleram falhas serão discutidos no parágrafo sobre redundância, a seguir.

Recuperação de falhas: a recuperação envolve projetar software de modo que o estado dos dados permanentes possa ser recuperado ou retrocedido, após a falha de um servidor. Em geral, as computações realizadas por alguns programas ficarão incompletas quando ocorrer uma falha e os dados permanentes que eles atualizam (arquivos em disco) podem não estar em um estado consistente. A recuperação de falhas será estudada no Capítulo 14.

Redundância: os serviços podem se tornar tolerantes a falhas com o uso de componentes redundantes. Considere os exemplos a seguir:

1. Sempre deve haver pelo menos duas rotas diferentes entre dois roteadores quaisquer na Internet.
2. No *Domain Name System*, toda tabela de correspondência de nomes é replicada em pelo menos dois servidores diferentes.
3. Um banco de dados pode ser replicado em vários servidores, para garantir que os dados permaneçam acessíveis após a falha de qualquer servidor. Os servidores podem ser projetados de forma a detectar falhas em seus pares; quando uma falha é detectada em um servidor, os clientes são redirecionados para os servidores restantes.

O projeto de técnicas eficazes para manter réplicas atualizadas de dados que mudam rapidamente, sem perda de desempenho excessiva, é um desafio. Várias estratégias para isso serão discutidas no Capítulo 15.

Os sistemas distribuídos fornecem um alto grau de disponibilidade perante falhas de hardware. A *disponibilidade* de um sistema é a medida da proporção de tempo em que ele está pronto para uso. Quando um dos componentes de um sistema distribuído falha, apenas o trabalho que estava usando o componente defeituoso é afetado. Um usuário pode passar para outro computador, caso aquele que estava usando falhe; um processo servidor pode ser iniciado em outro computador.

1.4.6 Concorrência

Tanto os serviços como os aplicativos fornecem recursos que podem ser compartilhados pelos clientes em um sistema distribuído. Portanto, existe a possibilidade de que vários clientes tentem acessar um recurso compartilhado ao mesmo tempo. Por exemplo, uma estrutura de dados que registre lances de um leilão pode ser acessada com muita freqüência, quando o prazo final se aproximar.

O processo que gerencia um recurso compartilhado poderia aceitar e tratar um pedido de cliente por vez. Mas essa estratégia limita o desempenho do tratamento de pedidos. Portanto, os serviços e aplicativos geralmente permitem que vários pedidos de cliente sejam processados concorrentemente. Para tornar isso mais concreto, suponha que cada recurso seja encapsulado como um objeto e que as chamadas sejam executadas em diferentes fluxos de execução, processos ou *threads*, concorrentes. Nessa situação é possível que vários fluxos de execução estejam simultaneamente dentro de um objeto e, eventualmente, suas operações podem entrar em conflito e produzir resultados inconsistentes. Por exemplo, se dois lances em um leilão forem Smith: $122 e Jones: $111 e as operações correspondentes fossem entrelaçadas sem nenhum controle, eles poderiam ser armazenados como Smith: $111 e Jones: $122.

A moral dessa história é que qualquer objeto que represente um recurso compartilhado em um sistema distribuído deve ser responsável por garantir que ele opere corretamente em um ambiente concorrente. Isso se aplica não apenas aos servidores, mas também aos objetos nos aplicativos. Portanto, todo programador que implemente um objeto que não foi destinado para uso em um sistema distribuído, deve fazer o que for necessário para garantir que em um ambiente concorrente ele não assuma resultados inconsistentes.

Para que um objeto mantenha coerência em um ambiente concorrente, suas operações devem ser sincronizadas de tal maneira que seus dados permaneçam consistentes. Isso pode ser obtido através de técnicas padrão, como semáforos, que são disponíveis na maioria dos sistemas operacionais. Esse assunto e sua extensão para coleções de objetos compartilhados distribuídos, serão discutidos nos Capítulos 6 e 13.

1.4.7 Transparência

A transparência é definida como sendo a ocultação, para um usuário final ou para um programador de aplicativos, da separação dos componentes em um sistema distribuído de modo que o sistema seja percebido como um todo, em vez de uma coleção de componentes independentes. As implicações da transparência têm grande influência sobre o projeto do software de sistema.

O ANSA *Reference Manual* [ANSA 1989] e o RM-ODP (*Reference Model for Open Distributed Processing*) da *International Organization for Standardization* [ISO 1992] identificam oito formas de transparência. Parafraseamos as definições originais da ANSA, substituindo transparência de migração por transparência de mobilidade, cujo escopo é mais abrangente:

Transparência de acesso permite que recursos locais e remotos sejam acessados com o uso de operações idênticas.

Transparência de localização permite que os recursos sejam acessados sem conhecimento de sua localização física ou na rede (por exemplo, qual prédio ou endereço IP).

Transparência de concorrência permite que vários processos operem concorrentemente, usando recursos compartilhados sem interferência entre eles.

Transparência de replicação permite que várias instâncias dos recursos sejam usadas para aumentar a confiabilidade e o desempenho, sem conhecimento das réplicas por parte dos usuários ou dos programadores de aplicativos.

Transparência de falhas permite a ocultação de falhas, possibilitando que usuários e programas aplicativos concluam suas tarefas, a despeito da falha de componentes de hardware ou software.

Transparência de mobilidade permite a movimentação de recursos e clientes dentro de um sistema, sem afetar a operação de usuários ou programas.

Transparência de desempenho permite que o sistema seja reconfigurado para melhorar o desempenho à medida que as cargas variam.

Transparência de escalabilidade permite que o sistema e os aplicativos se expandam em escala, sem alterar a estrutura do sistema ou os algoritmos de aplicativo.

As duas transparências mais importantes são a de acesso e a de localização; sua presença ou ausência afeta mais fortemente a utilização de recursos distribuídos. Às vezes, elas são referidas em conjunto como *transparência de rede*.

Como exemplo da transparência de acesso, considere o uso de uma interface gráfica em um sistema de arquivo que organiza diretórios e subdiretórios em pastas, o que o usuário enxerga é a mesma coisa independente dos arquivos serem contidos em uma pasta local ou remota. Outro exemplo é uma interface de programação para arquivos que usa as mesmas operações para acessar tanto arquivos locais como remotos (veja o Capítulo 8). Como exemplo de falta de transparência de acesso, considere um sistema distribuído que não permite acessar arquivos em um computador remoto, a não ser que você utilize o programa ftp para isso.

Os nomes de recurso na web, isto é, os URLs, são transparentes à localização, pois a parte do URL que identifica o nome de um servidor web se refere a um nome de computador em um domínio, em vez de seu endereço IP. Entretanto, os URLs não são transparentes à mobilidade, pois se a página web de alguém mudar para o seu novo local de trabalho, em um domínio diferente, todos as referências (*links*) nas outras páginas ainda apontarão para a página original.

Em geral, identificadores como os URLs, que incluem os nomes de domínio dos computadores, impedem a transparência de replicação. Embora o DNS permita que um nome de domínio se refira a vários computadores, ele escolhe apenas um deles ao pesquisar um nome. Como um esquema de replicação geralmente precisa acessar todos os computadores participantes, ele precisará acessar cada uma das entradas de DNS pelo nome.

Para ilustrar a transparência de rede, considere o uso de um endereço de correio eletrônico, como *Fred.Flintstone@stoneit.com*. O endereço consiste em um nome de usuário e um nome de domínio. Note que, embora os programas de correio aceitem nomes de usuário para usuários locais, eles anexam o nome de domínio local. O envio de correspondência para tal usuário não envolve o conhecimento de sua localização física ou na rede. Nem o procedimento de envio de uma mensagem de correio depende da localização do destinatário. Assim, o correio eletrônico, dentro da Internet, oferece tanto transparência de localização como de acesso (isto é, transparência de rede).

A transparência a falhas também pode ser vista no contexto do correio eletrônico, que finalmente é entregue, mesmo quando os servidores ou os enlaces de comunicação falham. As falhas são mascaradas pela tentativa de retransmitir as mensagens, até que elas sejam enviadas com êxito, mesmo que isso demore vários dias. Geralmente, o *middleware* converte as falhas de redes e processos em exceções em nível de programação (veja uma explicação no Capítulo 5).

Para ilustrar a transparência de mobilidade, considere o caso dos telefones móveis. Suponha que quem chama e quem é chamado estejam viajando de trem em diferentes partes de um país. Consideramos o telefone de quem chama como cliente e o telefone de quem foi chamado, como recurso. Os dois usuários de telefone que fazem a ligação não estão cientes da mobilidade dos telefones (o cliente e o recurso).

O uso dos diferentes tipos de transparência oculta e transforma em anônimos os recursos que não têm relevância direta para a execução de uma tarefa por parte de usuários e de programadores de aplicativos. Por exemplo, geralmente é desejável que recursos de hardware semelhantes sejam alocados de maneira permutável para executar uma tarefa – a identidade do processador usado para executar um processo geralmente fica oculta do usuário e permanece anônima. Conforme mencionado na Seção 1.2.3, nem sempre isso pode ser atingido. Por exemplo, um viajante que liga um notebook na rede local de cada escritório visitado, faz uso de serviços locais, como o envio de correio eletrônico, usando diferentes servidores em cada local. Mesmo dentro de um prédio, é normal preparar um documento para que ele seja impresso em uma determinada impressora configurada no sistema, normalmente, a que está mais próxima do usuário.

1.5 Resumo

Os sistemas distribuídos estão em toda parte. A Internet permite que usuários de todo o mundo acessem seus serviços onde quer que possam estar. Cada organização gerencia uma intranet, a qual fornece serviços locais e serviços Internet para usuários locais e remotos. Sistemas distribuídos de pequena escala podem ser construídos a partir de computadores móveis e outros dispositivos computacionais portáteis interligados através de redes sem fio.

O compartilhamento de recursos é o principal fator de motivação para a construção de sistemas distribuídos. Recursos como impressoras, arquivos, páginas web ou registros de banco de dados são gerenciados por servidores de tipo apropriado, por exemplo, servidores web gerenciam páginas web. Os recursos são acessados por clientes específicos, por exemplo; os clientes dos servidores web geralmente são chamados de navegadores.

A construção de sistemas distribuídos gera muitos desafios:

Heterogeneidade: eles devem ser construídos a partir de uma variedade de redes, sistemas operacionais, hardware e linguagens de programação diferentes. Os protocolos de comunicação da Internet mascaram a diferença existente nas redes e o *middleware* pode cuidar das outras diferenças.

Sistemas abertos: os sistemas distribuídos devem ser extensíveis – o primeiro passo é publicar as interfaces dos componentes, mas a integração de componentes escritos por diferentes programadores é um desafio real.

Segurança: a criptografia pode ser usada para proporcionar proteção adequada para os recursos compartilhados e para manter informações sigilosas em segredo, quando são transmitidas em mensagens por uma rede. Os ataques de negação de serviço ainda são um problema.

Escalabilidade: um sistema distribuído é considerado escalável se o custo da adição de um usuário for um valor constante, em termos dos recursos que devem ser adicionados. Os algoritmos usados para acessar dados compartilhados devem evitar gargalos de desempenho e os dados devem ser estruturados hierarquicamente para se obter os melhores tempos de acesso. Os dados acessados freqüentemente podem ser replicados.

Tratamento de falhas: qualquer processo, computador ou rede pode falhar, independentemente dos outros. Portanto, cada componente precisa conhecer as maneiras possíveis pelas quais os componentes de que depende podem falhar e ser projetado de forma a tratar de cada uma dessas falhas apropriadamente.

Concorrência: a presença de múltiplos usuários em um sistema distribuído é uma fonte de pedidos concorrentes para seus recursos. Em um ambiente concorrente, cada recurso deve ser projetado para manter a consistência nos estados de seus dados.

Transparência: o objetivo é tornar certos aspectos da distribuição invisíveis para o programador de aplicativos, para que este se preocupe apenas com o projeto de seu aplicativo em particular. Por exemplo, ele não precisa estar preocupado com sua localização ou com os detalhes sobre como suas operações serão acessadas por outros componentes, nem se será replicado ou migrado. As falhas de rede e de processos podem ser apresentadas aos programadores de aplicativos na forma de exceções – mas elas devem ser tratadas.

Exercícios

1.1 Cite cinco tipos de recurso de hardware e recursos de dados ou software que possam ser compartilhados com sucesso. Dê exemplos práticos de seu compartilhamento em sistemas distribuídos.
páginas 16, 20–22

1.2 Como os relógios de dois computadores ligados por uma rede local podem ser sincronizados sem referência a uma fonte de hora externa? Quais fatores limitam a precisão do procedimento que você descreveu? Como os relógios de um grande número de computadores conectados pela Internet poderiam ser sincronizados? Discuta a precisão desse procedimento. *página 16*

1.3 Um usuário chega a uma estação de trem que nunca havia visitado, portando um PDA capaz de interligação em rede sem fio. Sugira como o usuário poderia receber informações sobre os serviços locais e comodidades dessa estação, sem digitar o nome ou os atributos da estação. Quais desafios técnicos devem ser superados? *página 19–20*

1.4 Quais são as vantagens e desvantagens das tecnologias básicas HTML, URLs e HTTP para navegação em informações? Alguma dessas tecnologias é conveniente como base para a computação cliente-servidor em geral? *página 22*

1.5 Use a web como exemplo para ilustrar o conceito de compartilhamento de recursos, cliente e servidor.

Os recursos na web e outros serviços são nomeados por URLs. O que denotam as iniciais URL? Dê exemplos de três diferentes tipos de recursos da web que podem ser nomeados por URLs. *página 20–21*

1.6 Cite um exemplo de URL HTTP.

Liste os principais componentes de um URL HTTP, dizendo como seus limites são denotados e ilustrando cada um, a partir de seu exemplo.

Até que ponto um URL HTTP tem transparência de localização? *página 20–21*

1.7. Um programa servidor escrito em uma linguagem (por exemplo, C++) fornece a implementação de um objeto BLOB destinado a ser acessado por clientes que podem ser escritos em uma linguagem diferente (por exemplo, Java). Os computadores cliente e servidor podem ter hardware diferente, mas estão ligados a uma rede. Descreva os problemas causados por cada um dos cinco aspectos da heterogeneidade que precisam ser resolvidos para tornar possível que um objeto cliente invoque um método no objeto servidor. *página 28–29*

1.8 Um sistema distribuído aberto permite que novos serviços de compartilhamento de recursos, como o objeto BLOB do Exercício 1.7, sejam adicionados e acessados por uma variedade de programas clientes. Discuta, no contexto desse exemplo, até que ponto a necessidade de sistema aberto difere da necessidade de heterogeneidade. *página 29*

1.9 Suponha que as operações do objeto BLOB sejam separadas em duas categorias – operações públicas, que estão disponíveis para todos os usuários, e operações protegidas, que estão disponíveis apenas para certos usuários nomeados. Mencione todos os problemas envolvidos para se garantir que apenas os usuários nomeados possam usar uma operação protegida. Supondo que o acesso a uma operação protegida forneça informações que não devem ser reveladas a todos os usuários, quais outros problemas surgem? *página 29–30*

1.10 O serviço INFO gerencia um conjunto de recursos potencialmente muito grande, cada um dos quais podendo ser acessado por usuários de toda a Internet por intermédio de uma chave (um nome de string). Discuta uma estratégia para o projeto dos nomes dos recursos que dê uma perda de desempenho mínima à medida que o número de recursos no serviço aumenta. Sugira a maneira como o serviço INFO pode ser implementado de modo a evitar gargalos de desempenho quando o número de usuários se tornar muito grande. *página 30–31*

1.11 Liste os três principais componentes de software que podem falhar quando um processo cliente invocar um método em um objeto servidor. Dê um exemplo de falha para cada caso. Sugira a maneira como os componentes podem ser feitos para tolerar as falhas uns dos outros. *página 32–33*

1.12 Um processo servidor mantém um objeto de informações compartilhadas, como o objeto BLOB do Exercício 1.7. Dê argumentos a favor e contra o fato dos pedidos de cliente serem executados concorrentemente pelo servidor. No caso de eles serem executados concorrentemente, dê um exemplo de uma possível "interferência" que possa ocorrer entre operações solicitadas por diferentes clientes. Sugira o modo como tal interferência pode ser evitada. *página 34*

1.13 Um serviço é implementado por vários servidores. Explique por que recursos poderiam ser transferidos entre eles. Seria satisfatório que os clientes fizessem uma difusão seletiva (*multicast*) de todos os pedidos para o grupo de servidores como uma maneira de se obter transparência de mobilidade? *página 34*

2 Modelos de Sistema

Um modelo de arquitetura de um sistema distribuído envolve o posicionamento de suas partes e os relacionamentos entre elas. Exemplos incluem o modelo cliente-servidor e o modelo *peer-to-peer*. O modelo cliente-servidor pode ser modificado das seguintes maneiras:

- pelo particionamento dos dados ou pela replicação de servidores colaborativos;
- pela colocação dos dados em cache por meio de servidores *proxies* e clientes;
- pelo uso de código e de agentes móveis;
- pelo requisito de adicionar e remover dispositivos móveis de maneira conveniente.

Os modelos fundamentais envolvem uma descrição mais formal das propriedades comuns a todos os modelos de arquitetura.

Não existe a noção de relógio global em um sistema distribuído; portanto, os relógios de diferentes computadores não fornecem necessariamente a mesma hora. Toda comunicação entre processos é obtida por meio de troca de mensagens. A comunicação por troca mensagens em uma rede de computadores pode ser afetada por atrasos, pode sofrer uma variedade de falhas e é vulnerável a ataques contra a segurança. Esses problemas são tratados por três modelos:

- O modelo de interação que trata do desempenho e da dificuldade de estabelecer limites de tempo em um sistema distribuído, por exemplo, para entrega de mensagens.
- O modelo de falha que visa fornecer uma especificação precisa das falhas que podem ser exibidas por processos e canais de comunicação. Ele define a noção de comunicação confiável e da correção dos processos.
- O modelo de segurança que discute as possíveis ameaças aos processos e aos canais de comunicação. Ele apresenta o conceito de canal seguro, que é protegido dessas ameaças.

2.1 Introdução

Os sistemas destinados a uso em ambientes do mundo real devem ser projetados para funcionar corretamente na maior variedade possível de circunstâncias e perante muitas dificuldades e ameaças possíveis (para alguns exemplos, veja o quadro abaixo). A discussão e os exemplos do Capítulo 1 sugerem que sistemas distribuídos de diferentes tipos compartilham importantes propriedades subjacentes e dão origem a problemas de projeto comuns. Neste capítulo, apresentamos as propriedades e problemas comuns de projeto de sistemas distribuídos, na forma de modelos descritivos. Cada modelo é destinado a fornecer uma descrição abstrata e simplificada, mas consistente, de um aspecto relevante do projeto de um sistema distribuído.

Um modelo de arquitetura define a forma pela qual os componentes dos sistemas interagem e a maneira pela qual eles são mapeados em uma rede de computadores subjacente. Na Seção 2.2, descreveremos a estrutura em camadas de software de um sistema distribuído e os principais modelos de arquitetura que determinam as localizações e as interações dos componentes. Discutiremos as variantes do modelo cliente-servidor, incluindo aquelas relacionadas ao uso de código móvel. Consideraremos as características de um sistema distribuído no qual os dispositivos móveis podem ser adicionados ou removidos convenientemente. Finalmente, veremos os requisitos gerais de projeto para sistemas distribuídos.

Na Seção 2.3, apresentaremos três modelos fundamentais que ajudam a revelar aspectos importantes para os projetistas de sistemas distribuídos. O objetivo deles é especificar os principais problemas, dificuldades e ameaças que devem ser considerados no desenvolvimento de sistemas distribuídos para que esses executem suas tarefas corretamente, de forma confiável e segura. Os modelos fundamentais fornecem visões abstratas das características dos sistemas distribuídos que afetam sua dependabilidade* – correção, confiabilidade e segurança.

Dificuldades e ameaças para os sistemas distribuídos ◊ Aqui estão alguns dos problemas que os projetistas de sistemas distribuídos enfrentam:

Modos de uso que variam muito: as partes componentes dos sistemas estão sujeitas a amplas variações na carga de trabalho – por exemplo, algumas páginas web são acessadas vários milhões de vezes por dia. Alguns componentes de um sistema podem estar desconectados ou mal conectados em parte do tempo – por exemplo, quando computadores móveis são incluídos em um sistema. Alguns aplicativos têm requisitos especiais de necessitar grande largura de banda de comunicação e baixa latência – por exemplo, aplicativos multimídia.

Ampla variedade de ambientes de sistema: um sistema distribuído deve acomodar hardware, sistemas operacionais e redes heterogêneas. As redes podem diferir amplamente no desempenho – as redes sem fio operam a uma taxa de transmissão inferior a das redes locais cabeadas. Os sistemas computacionais podem apresentar ordens de grandeza totalmente diferentes – variando desde dezenas até milhões de computadores – devendo ser igualmente suportados.

Problemas internos: relógios não sincronizados, atualizações conflitantes de dados, diferentes modos de falhas de hardware e de software envolvendo os componentes individuais de um sistema.

Ameaças externas: ataques à integridade e ao sigilo dos dados, negação de serviço, etc.

* N. de R.T.: *Dependabilidade* é um neologismo empregado nas áreas de tolerância a falhas, sistemas de tempo real e sistemas distribuídos como tradução para o termo inglês *dependability* e significa a garantia (confiança) que um sistema oferece no provimento de um serviço ou tarefa.

2.2 Modelos de arquitetura de sistemas distribuídos

A arquitetura de um sistema é sua estrutura em termos de componentes especificados separadamente. O objetivo global é garantir que a estrutura atenda as demandas atuais e, provavelmente, as futuras impostas sobre ela. As maiores preocupações são tornar o sistema confiável, gerenciável, adaptável e rentável. O projeto arquitetônico de um prédio tem aspectos similares – ele determina não apenas sua aparência, mas também sua estrutura geral e seu estilo arquitetônico (gótico, neoclássico, moderno) fornecendo um padrão de referência consistente para seu projeto.

Nesta seção, vamos descrever os principais modelos de arquitetura empregados nos sistemas distribuídos – os estilos arquitetônicos desses sistemas. Construímos nossos modelos de arquitetura em torno dos conceitos de processo e objeto apresentados no Capítulo 1. O modelo de arquitetura de um sistema distribuído primeiramente simplifica e abstrai as funções dos componentes individuais de um sistema distribuído e, em seguida, considera:

- o posicionamento dos componentes em uma rede de computadores – buscando definir padrões para a distribuição de dados e da carga de trabalho;
- os inter-relacionamentos entre os componentes – isto é, seus papéis funcionais e os padrões de comunicação entre eles.

Uma simplificação inicial é obtida por meio da classificação dos processos como *processos servidores*, *processos clientes* e *processos peer-to-peer* – sendo estes últimos os processos que colaboram e se comunicam de maneira simétrica para realizar uma tarefa. Essa classificação de processos identifica as responsabilidades de cada um e, assim, nos ajuda a avaliar suas cargas de trabalho e a determinar o impacto das falhas em cada um deles. Os resultados dessa análise podem então ser usados para especificar o posicionamento dos processos de uma maneira que atenda os objetivos de desempenho e confiabilidade do sistema resultante.

Alguns sistemas mais dinâmicos podem ser construídos como variações do modelo cliente-servidor:

- A possibilidade de mover código de um processo para outro permite que um processo delegue tarefas; por exemplo, os clientes podem fazer o *download* de código a partir de servidores e executá-lo localmente. Essa possibilidade reduz os atrasos no acesso a ele e minimiza o tráfego de comunicação.
- Alguns sistemas distribuídos são projetados de forma que computadores e outros dispositivos móveis sejam adicionados ou removidos de forma transparente, permitindo que esses descubram os serviços disponíveis, ou que ofereçam seus serviços para aos demais.

Existem vários padrões amplamente usados para a divisão de tarefas em um sistema distribuído que têm um impacto importante sobre o desempenho e a eficiência do sistema resultante. O posicionamento dos processos que constituem um sistema distribuído em uma rede de computadores também é influenciado por problemas de desempenho, confiabilidade, segurança e custo. Os modelos de arquitetura aqui descritos oferecem uma visão simplificada dos padrões mais importantes.

2.2.1 Camadas de software

Originalmente, o termo *arquitetura de software* se referia à estruturação do software em camadas ou módulos em um único computador e, mais recentemente, em termos de serviços oferecidos e solicitados entre processos localizados em um mesmo computador ou em computadores diferentes. Essa visão orientada para processo e serviço pode ser expressa em termos de *camadas de serviço*. Apresentamos essa visão na Figura 2.1 e a desenvolvemos, cada vez com mais detalhes, nos Capítulos 3 a 6. Um *servidor* é um processo que aceita pedidos de outros processos. Um serviço distribuído pode ser fornecido por um ou mais processos servidores, interagindo uns com os outros e com processos clientes para manter uma visão consistente em nível de sistema dos recursos do serviço. Por exemplo, o serviço de sincronização dos relógios de computadores é implementado na Internet com base no protocolo NTP (*Network Time Protocol*) através de processos servidores que executam em diversos

Figura 2.1 Camadas de software e hardware em serviços de sistemas distribuídos.

computadores em toda a Internet. Esses servidores fornecem a hora atual para qualquer cliente que a solicite e acerta sua hora atual como resultado de interações realizadas com os outros servidores.

A Figura 2.1 apresenta dois termos importantes, *plataforma* e *middleware*, os quais definimos como segue:

Plataforma ◊ As camadas de hardware e software de mais baixo nível são freqüentemente denominadas de *plataforma* para sistemas e aplicativos distribuídos. Essas camadas de mais baixo nível fornecem serviços para as camadas que estão acima delas de forma a levar a interface de programação do sistema a um nível que facilita a comunicação e a coordenação entre processos. Intel x86/Windows, Intel x86/Solaris, PowerPC/Mac OS X, Intel x86/Linux são exemplos importantes de plataformas.

Middleware ◊ O *middleware* foi definido na Seção 1.4.1 como uma camada de software cujo objetivo é mascarar a heterogeneidade e fornecer um modelo de programação conveniente para os programadores de aplicativos. Um *middleware* é composto por um conjunto de processos ou objetos, em um grupo de computadores, que interagem entre si de forma a implementar comunicação e oferecer suporte para compartilhamento de recursos a aplicativos distribuídos. Ele se destina a fornecer blocos básicos de construção para a montagem de componentes de software que possam trabalhar juntos em um sistema distribuído. Em particular, ele simplifica as atividades de comunicação de programas aplicativos por meio do suporte de abstrações como a invocação a métodos remotos, a comunicação entre um grupo de processos, a notificação de eventos, o particionamento, posicionamento e recuperação de objetos de dados compartilhados entre computadores, a replicação de objetos de dados compartilhados e a transmissão de dados multimídia em tempo real. A comunicação em grupo será apresentada no Capítulo 4 e tratada com detalhes nos Capítulos 12 e 15. A notificação de eventos será descrita no Capítulo 5. O Capítulo 10 descreverá as estratégias para o compartilhamento de grandes conjuntos de objetos entre vários computadores. A replicação de dados será discutida no Capítulo 15 e os sistemas multimídia no Capítulo 17.

Os pacotes de chamada de procedimentos remotos (*Remote Procedure Call* – RPC), como o Sun RPC (Capítulo 5), e os sistemas de comunicação em grupo, como o Isis (Capítulo 15), são exemplos dos primeiros tipos de *middleware*. Atualmente há vários produtos e padrões de *middleware* orientados a objetos sendo amplamente empregados, eles incluem, entre outros:

- CORBA;
- RMI Java;
- serviços web (*web services*);
- DCOM (*Distributed Component Object Model*) da Microsoft;
- o RM-ODP (*Reference Model for Open Distributed Processing*) do ISO/ITU-T.

O RMI Java, os serviços web e CORBA serão descritos nos Capítulos 5, 19 e 20; detalhes sobre DCOM e RM-ODP podem ser encontrados em Redmond [1997] e Blair e Stefani [1997].

O *middleware* também pode fornecer serviços a serem usados por programas aplicativos. Essencialmente, eles são serviços de infra-estrutura fortemente vinculados ao modelo de programação distribuída fornecido pelo *middleware*. Por exemplo, o CORBA oferece uma variedade de serviços que fornecem aos aplicativos recursos que incluem, entre outros, atribuição de nomes (nomeação), segurança, transações, armazenamento persistente e notificação de eventos. Alguns dos serviços CORBA serão discutidos no Capítulo 20. Os serviços que estão na camada superior da Figura 2.1 são implementados com base nas primitivas de comunicação e dos próprios serviços oferecidos pelo *middleware* empregado.

Limitações do *middleware*: Muitos aplicativos distribuídos se baseiam completamente nos serviços oferecidos pelo *middleware* disponível para atender suas necessidades de comunicação e compartilhamento de dados. Por exemplo, um aplicativo que segue um modelo cliente-servidor, como um banco de dados de nomes e endereços, pode ser construído com um *middleware* que forneça apenas invocação a métodos remotos.

Por meio do desenvolvimento de *middlewares*, muito se tem conseguido na simplificação da programação de sistemas distribuídos, mas alguns aspectos de dependabilidade dos sistemas distribuídos exigem suporte em nível de aplicativo.

Considere a transferência de grandes mensagens de correio eletrônico, do computador do remetente ao do destinatário. À primeira vista, essa é uma simples aplicação do protocolo de transmissão de dados TCP (discutido no Capítulo 3). Mas considere o problema de um usuário que tenta transferir um arquivo muito grande por meio de uma rede potencialmente não confiável. O protocolo TCP fornece uma determinada capacidade de detecção e correção de erros, mas não se recupera de problemas mais sérios na rede. O serviço de transferência de correio eletrônico acrescenta outro nível de tolerância a falhas mantendo um registro do andamento da transferência e retomando a transmissão em uma nova conexão TCP, caso a original se desfaça.

Um artigo clássico de Saltzer, Reed e Clarke [Saltzer *et al.* 1984] menciona um ponto valioso a respeito do projeto de sistemas distribuídos, o que eles chamaram de princípio fim-a-fim. Citando sua colocação:

> Algumas funções relacionadas à comunicação podem ser completa e corretamente implementadas apenas com o conhecimento e a ajuda do aplicativo que está nos pontos extremos de um sistema de comunicação. Portanto, fornecer essa função como um recurso próprio de um sistema de comunicação nem sempre é sensato. (Uma versão mais simples de uma função fornecida pelo sistema de comunicação às vezes pode ser útil para a melhoria do desempenho).

Pode-se notar que esse princípio vai contra a visão de que todas as atividades de comunicação podem ser abstraídas da programação de aplicativos pela introdução de camadas de *middleware* apropriadas.

O ponto principal desse princípio é que o comportamento correto em programas distribuídos depende de verificações, mecanismos de correção de erro e medidas de segurança em muitos níveis, alguns dos quais exigindo acesso a dados dentro do espaço de endereçamento do aplicativo. Qualquer tentativa de realizar verificações dentro do próprio sistema de comunicação garantirá apenas parte da correção exigida. Portanto, o mesmo trabalho provavelmente será feito em programas aplicativos, desperdiçando esforço de programação e, o mais importante, acrescentando complexidade desnecessária e executando operações redundantes.

Não há espaço aqui para detalhar melhor os argumentos que embasam o princípio fim-a-fim; o artigo citado é fortemente recomendado para leitura – ele está repleto de exemplos esclarecedores. Um dos autores originais mostrou, recentemente, que os benefícios substanciais trazido pelo uso do princípio fim-a-fim no projeto da Internet são colocados em risco pelas atuais mudanças na especialização dos serviços de rede para atender os requisitos dos aplicativos atuais [www.reed.com].

2.2.2 Arquiteturas de sistema

A divisão de responsabilidades entre os componentes de um sistema (aplicativos, servidores e outros processos) e a atribuição destes nos diversos computadores de uma rede talvez seja o aspecto mais evidente do projeto de sistemas distribuídos. Isso tem implicações importantes no desempenho, na

confiabilidade e na segurança do sistema resultante. Nesta seção, destacaremos os principais modelos de arquitetura que servem como base para se realizar essa distribuição de responsabilidades e a atribuição de componentes a computadores.

Em um sistema distribuído, os processos possuem responsabilidades bem definidas e interagem para realizar uma atividade útil. Nesta seção, nossa atenção se voltará a atribuição dos processos a computadores como, por exemplo, se vê na Figura 2.2, que mostra a disposição de processos (elipses) nos computadores (caixas cinza). Usamos os termos invocação e resultado para rotular as mensagens – elas poderiam igualmente ser rotuladas como pedido e resposta

Os dois tipos principais de modelos de arquitetura estão ilustrados nas Figuras 2.2 e 2.3, e estão descritos a seguir.

Cliente-servidor ◊ Essa é a arquitetura mais citada quando os sistemas distribuídos são discutidos. Historicamente ela é a mais importante e continua sendo amplamente empregada. A Figura 2.2 ilustra a estrutura simples na qual os processos clientes interagem com processos servidores, localizados em distintos computadores hospedeiros, para acessar os recursos compartilhados que estes gerenciam.

Os servidores podem, por sua vez, ser clientes de outros servidores, conforme a figura indica. Por exemplo, um servidor web é freqüentemente um cliente de um servidor de arquivos local que gerencia os arquivos nos quais as páginas web estão armazenadas. Os servidores web, e a maioria dos outros serviços Internet, são clientes do serviço DNS, que mapeia nomes de domínio Internet a endereços de rede (IP). Outro exemplo relacionado à web diz respeito aos *mecanismos de busca*, os quais permitem aos usuários pesquisar resumos de informações disponíveis em páginas web, em sites de toda a Internet. Esses resumos são feitos por programas chamados *web crawlers**, que são executados em segundo plano (*background*) em um site de mecanismo de busca, usando pedidos HTTP para acessar servidores web em toda a Internet. Assim, um mecanismo de busca é tanto um servidor como um cliente: ele responde às consultas de clientes navegadores e executa *web crawlers* que atuam como clientes de outros servidores web. Nesse exemplo, as tarefas do servidor (responder às consultas dos usuários) e as tarefas do *web crawler* (fazer pedidos para outros servidores web) são totalmente independentes; há pouca necessidade de sincronizá-las e elas podem ser executadas concomitantemente. Na verdade, um mecanismo de busca típico, normalmente, é feito por muitas *threads* concorrentes, algumas servindo seus clientes e outras executando *web crawlers*. No Exercício 2.4, o leitor é convidado a refletir sobre o problema de sincronização que surge para um mecanismo de busca concorrente do tipo aqui esboçado.

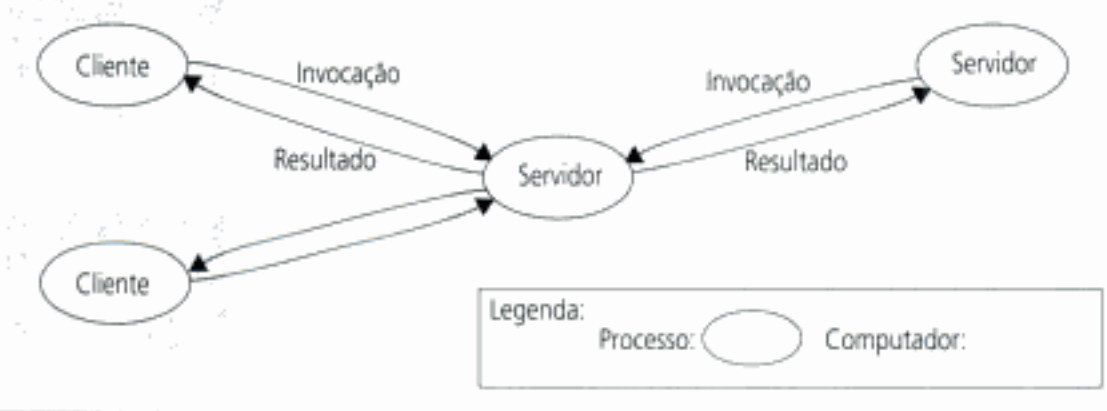


Figura 2.2 Os clientes realizam pedidos a servidores.

* N. de R.T.: Também denominados de *spiders* (aranhas) em analogia ao fato de que passeiam sobre a *web* (teia); entretanto, é bastante comum o uso do termo *web crawler* e, por isso, preferimos não traduzi-lo.

Peer-to-peer* ◊ Nessa arquitetura, todos os processos envolvidos em uma tarefa ou atividade desempenham funções semelhantes, interagindo cooperativamente como *pares* (*peers*), sem distinção entre processos clientes e servidores, nem entre os computadores em que são executados. Embora o modelo cliente-servidor ofereça uma estratégia direta e relativamente simples para o compartilhamento de dados e outros recursos, ele não é flexível em termos de escalabilidade. A centralização do fornecimento e gerenciamento de serviços, acarretada pela colocação de um serviço em um único computador, não favorece um aumento de escala, além daquela limitada pela capacidade do computador que contém o serviço e da largura de banda de suas conexões de rede.

Na próxima seção, descreveremos diversas variações da arquitetura cliente-servidor que evoluíram como uma resposta a esse problema, mas nenhuma delas trata do problema fundamental – a necessidade de distribuir recursos compartilhados de uma forma mais ampla para dividir as cargas de computação e de comunicação entre um número muito grande de computadores e de conexões de rede.

A capacidade do hardware e a funcionalidade do sistema operacional dos computadores do tipo *desktops* atuais ultrapassam aquelas dos servidores antigos e ainda, a maioria desses computadores, está equipada com conexões de rede de banda larga e sempre ativas. O objetivo da arquitetura *peer-to-peer* é explorar os recursos (tanto dados como de hardware) de um grande número de computadores para o cumprimento de uma dada tarefa ou atividade. Tem-se construído, com sucesso, aplicativos e sistemas *peer-to-peer* que permitem a dezenas, ou mesmo, centenas de milhares de computadores, fornecerem acessos a dados e a outros recursos que eles armazenam e gerenciam coletivamente. O exemplo mais antigo, e conhecido, desse tipo de arquitetura é o aplicativo Napster, empregado para o compartilhamento de arquivos de música digital. Embora tenha se tornado famoso por outro motivos que não a sua arquitetura, sua demonstração de exeqüibilidade resultou no desenvolvimento desse modelo de arquitetura em muitas direções importantes.

A Figura 2.3 ilustra o formato de um aplicativo *peer-to-peer*. Os aplicativos são compostos de grandes números de processos (*peers*) executados em distintos computadores e o padrão de comunicação entre eles depende inteiramente do que o aplicativo faz. Nessa figura, um grande número de objetos de dados são compartilhados e um computador individual contém apenas uma pequena parte de um banco de dados desse aplicativo, o que faz com que as cargas de armazenamento, processamento e comunicação para acessar diferentes objetos sejam distribuídas por muitos computadores e conexões de rede. Cada objeto pode ser replicado em vários computadores para distribuir melhor a carga e para proporcionar resiliência, no caso de desconexão de computadores individuais (isso é inevitável em redes grandes e heterogêneas para os quais os sistemas *peer-to-peer* se destinam). A necessidade de colocar objetos individuais, de recuperá-los e manter réplicas entre muitos computadores torna essa arquitetura substancialmente mais complexa do que a arquitetura cliente-servidor.

O desenvolvimento de aplicativos *peer-to-peer* e de *middlewares* para suportá-los será descrito com profundidade no Capítulo 10.

2.2.3 Variações

Diversas variações dos modelos anteriores podem ser extraídas a partir da consideração dos seguintes fatores:

- o uso de vários servidores e de caches para aumentar o desempenho e a resiliência;
- o uso de código móvel e agentes móveis;
- a necessidade dos usuários possuírem computadores de baixo custo, com recursos de hardware limitados e simples de gerenciar;
- o requisito de adicionar e remover dispositivos móveis de maneira conveniente.

Serviços fornecidos por vários servidores ◊ Os serviços podem ser implementados como vários processos servidores em diferentes computadores hospedeiros, interagindo conforme for necessário

* N. de R.T.: Por questões de clareza, manteremos o termo técnico *peer-to-peer* em inglês para denotar a arquitetura onde os processos (*peers*) não possuem hierarquia entre si.

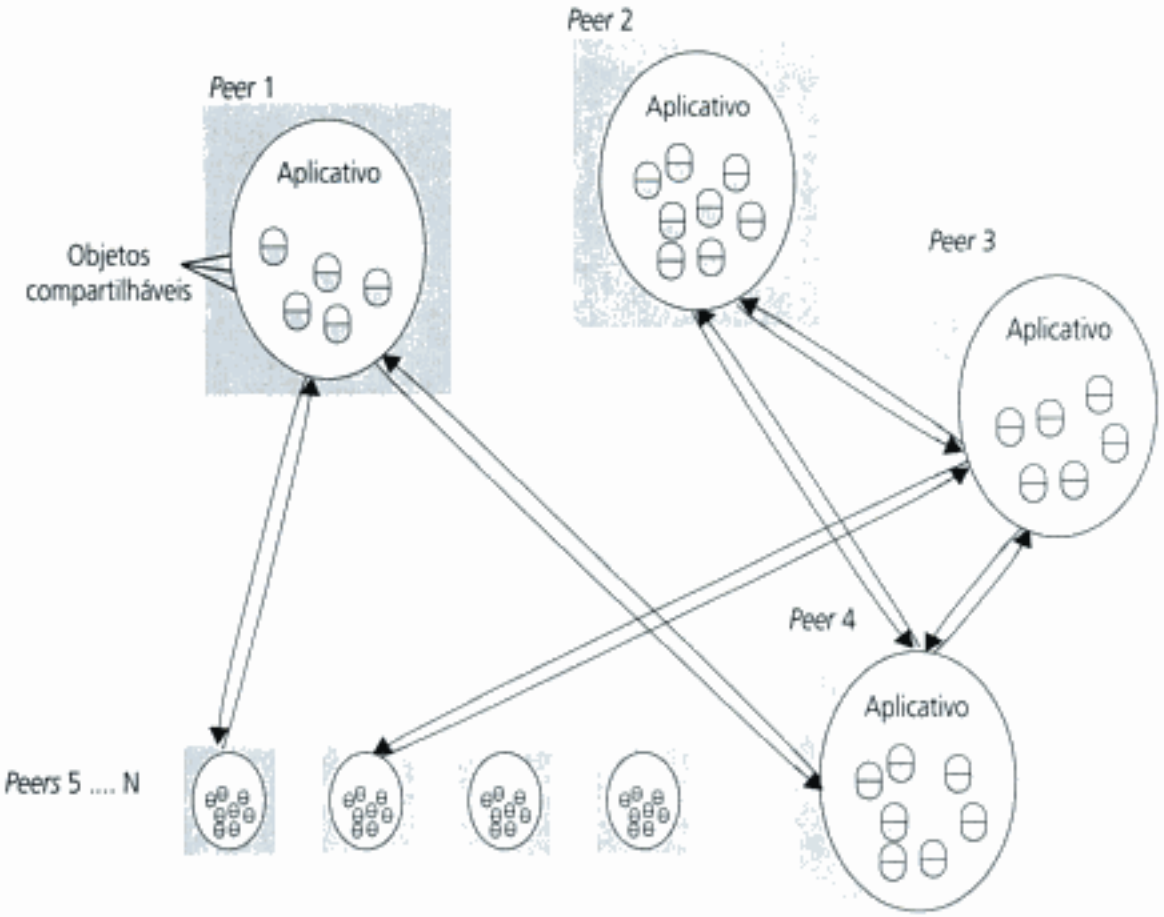


Figura 2.3 Um aplicativo distribuído baseado em arquitetura *peer-to-peer*.

para fornecer um serviço para processos clientes (Figura 2.4). Os servidores podem particionar o conjunto de objetos nos quais o serviço é baseado e distribuí-los entre eles mesmos ou podem, ainda, manter cópias duplicadas deles em vários outros hospedeiros. Essas duas opções são ilustradas pelos exemplos a seguir.

A web oferece um exemplo comum de particionamento de dados no qual cada servidor web gerencia seu próprio conjunto de recursos. Um usuário pode usar um navegador para acessar um recurso em qualquer um desses servidores.

Um exemplo de serviço baseado em dados replicados é o NIS (*Network Information Service*) da Sun, usado por computadores em uma rede local quando os usuários se conectam. Cada servidor NIS tem sua própria cópia (réplica) de um arquivo de senhas que contém uma lista de nomes de *login* dos usuários e suas respectivas senhas criptografadas. O Capítulo 15 discutirá as técnicas de replicação em detalhes.

Um tipo de arquitetura onde ocorre uma interação maior entre vários servidores, e por isso, denominada de arquitetura fortemente acoplada, é o baseado em *cluster**. Essa arquitetura é comumente empregada por serviços web que necessitam oferecer um grande grau de escalabilidade, como os mecanismos de busca e as lojas on-line. Um cluster é construído a partir de várias, às vezes centenas,

* N. de R.T.: É comum encontrarmos os termos agregado, ou agrupamento, como tradução da palavra *cluster*. Na realidade existem dois tipos de *clusters*. Os denominados de fortemente acoplados são compostos por vários processadores e atuam como multiprocessadores. Normalmente são empregados para atingir alta disponibilidade e balanceamento de carga. Os fracamente acoplados são formados por um conjunto de computadores interligados em rede e são comumente utilizados para processamento paralelo e de alto desempenho.

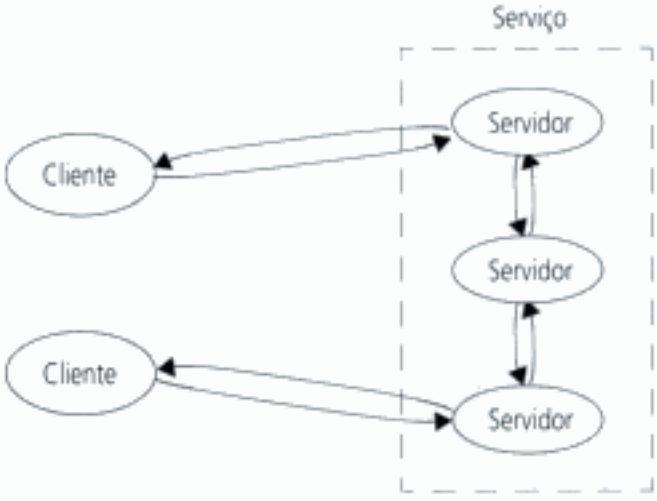


Figura 2.4 Um serviço fornecido por vários servidores.

de unidades de processamento (veja a Seção 6.4.2). Nessa arquitetura, a execução de um serviço pode ser particionada ou duplicada entre as unidades de processamento.

Servidores *proxies* e caches ◊ Uma *cache* consiste em realizar um armazenamento de objetos de dados recentemente usados em um local mais próximo que a origem real dos objetos em si. Quando um novo objeto é recebido em um computador, ele é adicionado a cache, substituindo, se houver necessidade, alguns objetos já existentes. Quando um processo cliente requisita um objeto, o serviço de cache primeiro verifica se possui armazenado uma cópia atualizada desse objeto, caso esteja disponível, o mesmo é entregue ao processo cliente. Se o objeto não estiver armazenado, ou se a cópia não está atualizada, o mesmo é acessado diretamente em sua origem. As caches podem ser mantidas nos próprios clientes, ou localizadas em um servidor *proxy* que possa ser compartilhado por eles.

Na prática, o emprego de caches é bastante comum. Por exemplo, os navegadores web mantêm no sistema de arquivos local uma cache das páginas recentemente visitadas e, antes de exibi-las, com o auxílio de uma requisição HTTP especial, verifica nos servidores originais se as páginas armazenadas na cache estão atualizadas. Um servidor *proxy* web (Figura 2.5) fornece uma cache compartilhada de recursos web para máquinas clientes de um ou vários sites. O objetivo dos servidores *proxies* é aumentar a disponibilidade e o desempenho do serviço, reduzindo a carga sobre a rede remota e sobre os servidores web. Os servidores *proxies* podem assumir outras funções; como por exemplo, serem usados para acessar servidores web através de um *firewall*.

Código móvel ◊ O Capítulo 1 apresentou o conceito de código móvel. Os *applets* representam um exemplo bem conhecido e bastante utilizado de código móvel – o usuário, executando um navegador, seleciona um *link* que aponta para um *applet*, cujo código é armazenado em um servidor web; o código é carregado no navegador e, como se vê na Figura 2.6, posteriormente executado. Uma vantagem

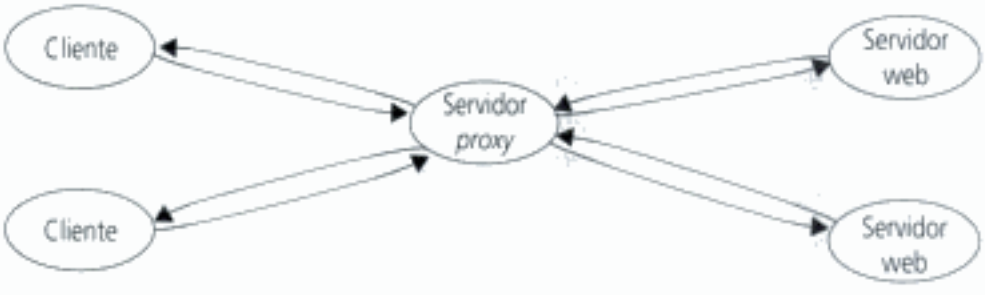


Figura 2.5 Servidor *proxy* web.

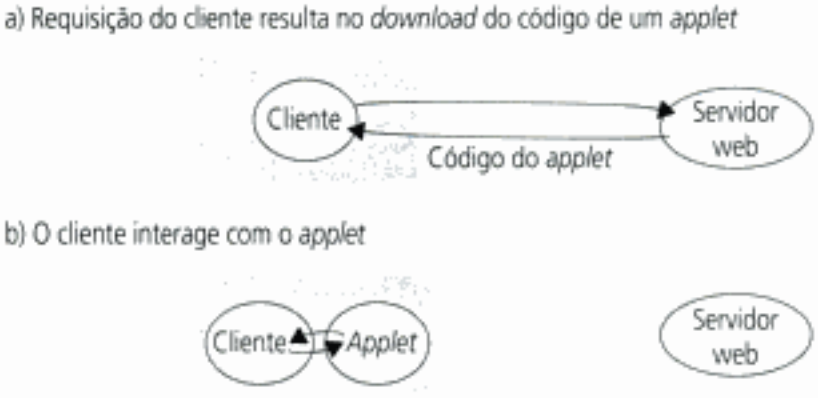


Figura 2.6 *Applets* web.

de executar um código localmente é que ele pode dar uma boa resposta interativa, pois não sofre os atrasos, nem a variação da largura de banda associada à comunicação na rede.

Acessar serviços significa executar código que pode ativar suas operações. Alguns serviços são tão padronizados que podemos acessá-los com um aplicativo já existente e bem conhecido – a web é o exemplo mais comum disso, mas mesmo nela, alguns sites usam funcionalidades não disponíveis em navegadores padrão e exigem o *download* de código adicional. Esse código adicional pode, por exemplo, se comunicar com um servidor. Considere uma aplicação que exige que os usuários precisem estar atualizados com relação às alterações que ocorrerem em uma fonte de informações em um servidor. Isso não pode ser obtido pelas interações normais com o servidor web, pois essas são sempre iniciadas pelo cliente. A solução é usar software adicional que opere de uma maneira freqüentemente referida como modelo *push* – no qual o servidor inicia as interações, em vez do cliente.

Por exemplo, um corretor de bolsa de valores poderia fornecer um serviço personalizado para notificar os usuários sobre alterações nos preços das ações. Para usar esse serviço, cada indivíduo teria que fazer o *download* de um *applet* especial que recebesse atualizações do servidor do corretor, as exibisse para o usuário e, talvez, executasse automaticamente operações de compra e venda, disparadas por condições preestabelecidas e armazenadas por uma pessoa em seu computador.

O Capítulo 1 mencionou que o uso de código móvel é uma ameaça potencial à segurança do computador destino. Como uma das formas empregadas para reduzir esse risco, os navegadores dão aos *applets* um acesso limitado a seus recursos locais usando um esquema discutido na Seção 7.1.1.

Agentes móveis ◊ Um agente móvel é um programa em execução (inclui código e dados) que passa de um computador para outro em um ambiente de rede, realizando uma tarefa em nome de alguém, como uma coleta de informações, e finalmente retornando com os resultados obtidos a esse alguém. Um agente móvel pode efetuar várias requisições aos recursos locais de cada site que visita como, por exemplo, acessar entradas de banco de dados. Se compararmos essa arquitetura com um cliente estático que solicita, via requisições remotas, acesso a alguns recursos, possivelmente transferindo grandes volumes de dados, há uma redução no custo e no tempo da comunicação graças à substituição das requisições remotas por requisições locais.

Os agentes móveis podem ser usados para instalar e manter software em computadores dentro de uma empresa, ou para comparar os preços de produtos de diversos fornecedores, visitando o site de cada fornecedor e executando uma série de operações de consulta. Um exemplo já antigo, de uma idéia semelhante, é o chamado programa *worm*, desenvolvido no Xerox PARC [Shoch e Hupp 1982], o qual foi projetado para fazer uso de computadores ociosos para efetuar cálculos intensivos.

Os agentes móveis (assim como o código móvel) são uma ameaça em potencial à segurança para os recursos existentes nos computadores que visitam. O ambiente que recebe um agente móvel deve decidir, com base na identidade do usuário, em nome de quem o agente está atuando, qual dos recursos locais ele pode usar. A identidade deve ser incluída de maneira segura com o código e com

os dados do agente móvel. Além disso, os agentes móveis, em si, podem ser vulneráveis – eles podem não conseguir completar sua tarefa, caso seja recusado o acesso às informações de que precisam. Para contornar esse problema, as tarefas executadas pelos agentes móveis podem ser feitas usando outras técnicas. Por exemplo, os *web crawlers* que precisam acessar recursos em servidores web em toda a Internet funcionam com muito sucesso, fazendo requisições remotas de processos servidores. Por esses motivos, a aplicabilidade dos agentes móveis é limitada.

Computadores de rede ◊ Na arquitetura ilustrada na Figura 1.2, um usuário pode executar uma série de aplicativos localmente em computador do tipo *desktop*. Os sistemas operacionais e softwares aplicativos desse tipo de computador exigem que grande parte do código e dos dados ativos estejam armazenados em um disco local. Entretanto, o gerenciamento de arquivos de aplicativos, e a própria manutenção de uma base de software local, exige considerável esforço técnico, de uma natureza que a maioria dos usuários não está qualificada a fornecer.

O uso de computador em rede surge como uma resposta a esse problema permitindo que tanto o sistema operacional como todo o software aplicativo necessário para o usuário, seja carregado a partir de um servidor de arquivos remoto. Os aplicativos são executados de forma local no computador do usuário, mas os arquivos são gerenciados pelo servidor de arquivos remoto. Aplicativos de rede, como um navegador web, também podem ser executados dessa forma. Como todos os dados e código de aplicativo são armazenados em um servidor de arquivos, os usuários podem migrar de um computador de rede para outro sem afetar o seu trabalho. A capacidade do processador e da memória de um computador em rede pode ser restrita de forma a reduzir seu custo.

Caso o computador em rede possua um disco rígido, ele poderá conter um mínimo necessário de software. O restante do disco poderá ser usado como *armazenamento cache*, contendo cópias de software e de arquivos de dados recentemente recuperados dos servidores. A manutenção da validade das informações da cache não exige nenhum esforço por parte dos usuários: os objetos ali armazenados podem ser automaticamente invalidados quando uma nova versão de um arquivo for gravada em um determinado servidor.

Clientes "leves" ◊ O termo *cliente "leve"* se refere a uma camada de software, em um computador local, que oferece ao usuário uma interface baseada em janelas para que este possa executar programas aplicativos em um computador remoto (Figura 2.7). Essa arquitetura tem os mesmos baixos custos de gerenciamento e de hardware que o esquema de computador em rede, porém, em vez de fazer o *download* do código de aplicativos no computador do usuário, ela os executa em um *servidor de computação* – um computador com capacidade suficiente para executar um grande número de aplicativos simultaneamente. Normalmente, o servidor de computação é um computador com vários processadores ou um *cluster* (veja o Capítulo 6) executando uma versão para multiprocessadores de um sistema operacional, como o UNIX ou o Windows.

O principal inconveniente da arquitetura cliente leve está nas atividades gráficas altamente interativas, como CAD e processamento de imagens, onde os atrasos sentidos pelos usuários aumentam em função da necessidade de transferir imagens entre o cliente leve e o processo aplicativo, o que provoca latências na rede e no sistema operacional.

Implementações de cliente leve: Os sistemas de cliente leve são conceitualmente simples e sua implementação é direta em alguns ambientes. Por exemplo, a maioria das variantes do UNIX inclui o sistema de janelas X-11, discutido no quadro a seguir.

Figura 2.7 Clientes leves e servidores de computação.

O sistema de janelas X-11 ◊ O sistema de janelas X-11 [Scheifler e Gettys 1986] é um processo que gerencia a tela e dispositivos de entrada interativos (teclado, mouse) do computador em que é executado. O X-11 fornece uma ampla biblioteca de funções (o *protocolo X-11*) para exibição e modificação de objetos gráficos em janelas, assim como a criação e manipulação das próprias janelas.

O sistema X-11 é referenciado como um processo *servidor de janelas*. Os programas aplicativos com os quais o usuário está interagindo no momento são os clientes de um servidor X-11. Os programas clientes se comunicam com o servidor através de requisições ao protocolo X-11; o que inclui as operações para desenhar texto e objetos gráficos em janelas. Os clientes podem ou não executar no mesmo computador que o servidor, pois as funções disponibilizadas pelo servidor são sempre acionadas com um mecanismo RPC (*Remote Procedure Call*). Portanto, o servidor de janelas X-11 tem as principais propriedades de um cliente leve. O X-11 inverte a terminologia cliente-servidor. O nome servidor X-11 é derivado dos serviços de exibição gráfica que ele fornece para os programas aplicativos, enquanto cliente leve faz referência ao software que usa programas aplicativos executando em um servidor de computação.

Existem vários pontos na seqüência de tratamento de informações necessários para suportar adequadamente uma interface gráfica com o usuário, nos quais o limite entre cliente e servidor pode ser definido. O X-11 se limita ao suporte a primitivas gráficas para o desenho de linhas e de formas básicas e do tratamento de janelas. O produto *WinFrame* da Citrix [www.citrix.com] é uma implementação comercial, bastante utilizada, do conceito de cliente leve que opera de maneira semelhante. Esse produto fornece um processo cliente leve que pode ser executado em uma grande variedade de plataformas e oferece um ambiente de trabalho que permite o acesso interativo a aplicativos executados em máquinas windows. Outras implementações incluem o sistema VNC (*Virtual Network Computer*), desenvolvido no AT&T Laboratories, em Cambridge, Inglaterra [Richardson *et al.* 1998]. O sistema VNC suporta operações gráficas em nível de pixels de tela e salva o contexto do ambiente gráfico dos usuários mesmo no caso de troca de computadores por parte deles.

Dispositivos móveis e interoperabilidade espontânea ◊ Discutimos a mobilidade do software na forma de agentes móveis, os quais migram entre computadores físicos. Em contraste, os dispositivos móveis são componentes de computação, em *hardware*, que se movem entre locais físicos e, portanto, em redes, transportando consigo componentes de software. Como mencionado no Capítulo 1, há no mundo cada vez mais dispositivos móveis, incluindo, entre outros, laptops, equipamentos de mão, como os PDAs (assistentes digitais pessoais), telefones móveis, câmeras digitais e computadores acoplados ao corpo, como os relógios de pulso inteligentes. Cada vez mais, muitos desses equipamentos possuem a capacidade de se comunicar através do uso de redes sem fio. O raio de ação das redes sem fio varia desde um nível nacional, ou mesmo maior, como as redes de telecomunicações GSM e 3G, à algumas centenas de metros, como redes WiFi (IEEE 802.11) ou ainda a cerca de uma dezena de metros, como o Bluetooth.

A mobilidade de dispositivo tem muitas implicações, dentre elas, várias para os sistemas baseados na arquitetura cliente-servidor. Tanto clientes como servidores podem existir em dispositivos móveis – sendo os clientes móveis o caso mais comum. Considere um ônibus que realiza um passeio pelo mundo, explorando conectividade WiFi, nos locais que ela estiver disponível, ou a infra-estrutura de rede de telecomunicações em outros lugares. Os passageiros e suas famílias querem atualizar e ler um site que contém fotos e informações sobre o passeio. A maneira mais óbvia de conseguir isso seria que todos acessassem um servidor fixo, com os passageiros do ônibus fazendo *upload* e lendo dados a partir de seus clientes móveis. Mas uma alternativa possível seria executar o servidor web no ônibus, para que o próprio servidor fosse móvel. O servidor não apenas poderia fornecer fotos e informações para os usuários, mas também poderia fornecer um mapa mostrando a posição correta do ônibus, usando um receptor GPS (*Global Positioning System*) [Kindberg *et al.* 2002a].

A transparência de mobilidade (Seção 1.4.7) é um problema freqüente para dispositivos móveis, assim como para agentes móveis. Em particular, os clientes de um servidor móvel, como o caso do

exemplo do ônibus, não precisam saber de sua movimentação entre as diferentes redes (mesmo que as famílias possam estar interessadas na localização geográfica do ônibus, conforme mostrado no mapa atualizado pelo GPS). O IP móvel, descrito no Capítulo 3, oferece uma solução por meio da qual o servidor do ônibus mantém o seu endereço Internet (IP), mesmo que se mova entre várias redes.

A conectividade variável é outra questão importante, a qual afeta tanto a capacidade do ônibus de atuar como servidor, como a capacidade de seus passageiros em acessar serviços externos. O ônibus pode perder conexão com a rede sem fio de forma intermitente, por breves períodos de tempo, enquanto passa por obstáculos que bloqueiam o sinal de transmissão, como túneis, ou por períodos maiores, em regiões onde a conectividade a rede sem fio cessa completamente. Além disso, o servidor do ônibus está sujeito a uma largura de banda bastante variável, ao se mover entre conexões da rede WiFi e da rede de telecomunicações. O Capítulo 14 discutirá o tópico do tratamento de desconexões. O Capítulo 16 discutirá a adaptação à largura de banda variável.

Finalmente, a mobilidade leva à *interoperabilidade espontânea*, uma variação no modelo cliente-servidor na qual associações entre dispositivos são rotineiramente criadas e destruídas. Na Seção 1.2.3, descrevemos um usuário visitando uma organização anfitriã e usando seus dispositivos móveis em conjunto com os dispositivos do anfitrião, como as impressoras. Analogamente, poderiam ser oferecidos aos passageiros do ônibus, serviços integrados com os ambientes físicos pelos quais o ônibus passa, como informações sobre as atrações turísticas locais.O principal desafio que se aplica a tais situações é tornar a interoperabilidade rápida e conveniente (isto é, espontânea), mesmo que o usuário esteja em um ambiente que nunca tenha visitado antes. Isso significa ativar o dispositivo do visitante para se comunicar na rede do anfitrião e associá-lo convenientemente aos serviços locais – um processo chamado de *descoberta de serviço*. Examinaremos a interoperabilidade espontânea e outros aspectos da mobilidade com mais profundidade, no Capítulo 16. O Capítulo 16 também abordará o tema de serviços com reconhecimento de contexto, relacionado com a área de computação ubíqua e redes de sensores, onde estados do sistema dependem do mundo físico que ele se encontra, como por exemplo, o mapa que mostra a posição atual do ônibus.

2.2.4 Interfaces e objetos

O conjunto de funções disponíveis para invocação em um processo (seja ele um processo servidor ou um *peer*) é especificado por uma ou mais *definições de interface*. As definições de interface são descritas no Capítulo 5, mas o conceito já é conhecido de quem é familiarizado com linguagens como Modula, C++ ou Java. Na forma mais pura da arquitetura cliente-servidor, cada processo servidor é visto como uma única entidade, com uma interface fixa que define as funções que podem ser invocadas por um cliente.

Nas linguagens orientadas a objetos, como C++ e Java, com suporte adicional apropriado, os processos distribuídos podem ser construídos de maneira orientada a objeto. Muitos objetos podem ser encapsulados em processos servidores e referências a eles são passadas aos outros processos para que seus métodos sejam acessados por invocação remota. Essa é a estratégia adotada pelo CORBA e pelo Java através de seus mecanismos de invocação a método remoto (RMI). Nesse modelo, o número e os tipos de objetos contidos em cada processo podem variar conforme as exigências das atividades do sistema e, em algumas situações, mesmo a localização dos objetos também pode mudar.

Seja adotando uma arquitetura cliente-servidor estática, ou o modelo orientado a objetos mais dinâmico delineado no parágrafo anterior, a distribuição de responsabilidades entre processos e entre computadores continua sendo um aspecto importante de projeto. No modelo de arquitetura tradicional, as responsabilidades são alocadas estaticamente (um servidor de arquivos, por exemplo, é responsável apenas por arquivos e não por páginas web, por seus servidores *proxies* ou outros tipos de objeto). Mas no modelo orientado a objetos, novos serviços e, em alguns casos, novos tipos de objeto, podem ser instanciados e disponibilizados imediatamente para invocação.

2.2.5 Requisitos de projeto para arquiteturas distribuídas

Os fatores que motivam a distribuição de objetos e processos em um sistema distribuído são numerosos e sua importância é relevante. O compartilhamento foi obtido pela primeira vez nos sistemas operacionais multiprogramados dos anos 60 com o uso de arquivos compartilhados. As vantagens do

compartilhamento foram rapidamente reconhecidas e exploradas em sistemas operacionais multiusuário, como o UNIX, e em sistemas de banco de dados multiusuário, como o Oracle, para permitir que os processos repartissem recursos de sistema e dispositivos (capacidade de armazenamento de arquivo, impressoras, fluxos de áudio e vídeo) e compartilhassem objetos de aplicativo.

A queda no custo dos microprocessadores, aliado ao aumento do poder de processamento deles, eliminou a necessidade de compartilhar o uso de processadores centrais. Entretanto, nos anos 70 e 80, a disponibilidade de redes de computador de desempenho médio e a contínua necessidade de dividir acesso a recursos de hardware de custo relativamente caros, como impressoras e discos de grande capacidade, levaram ao desenvolvimento dos sistemas distribuídos. Atualmente, o compartilhamento de recursos é corriqueiro. Mas o compartilhamento eficiente de dados em larga escala continua sendo um importante desafio – com a alteração dos dados (e a maioria dos dados muda com o tempo), a possibilidade de atualizações concorrentes e conflitantes aumenta. O controle de atualizações concorrentes em dados compartilhados é o assunto dos Capítulos 13 e 14.

Problemas de desempenho ◊ Os desafios advindos da distribuição de recursos vai muito além da necessidade do gerenciamento de atualizações concorrentes. Os principais problemas de desempenho devido à limitação finita da capacidade de recursos de processamento e de comunicação são apresentados a seguir:

Reatividade*: os usuários de aplicativos interativos exigem uma resposta rápida e consistente; mas os programas clientes freqüentemente acessam recursos compartilhados, o que introduz alguns atrasos de processamento. Quando um serviço remoto está envolvido, a velocidade com que a resposta é gerada é determinada não apenas pela carga e pelo desempenho do servidor e da rede, mas também pelos atrasos em todos os componentes de software envolvidos – os serviços de comunicação e *middleware* dos sistemas operacionais cliente e servidor (suporte para invocação remota, por exemplo), assim como o código do processo que implementa o serviço.

Além disso, a transferência de dados entre processos e o chaveamento de contexto associado são relativamente lentos, mesmo quando os processos residem no mesmo computador. Para obter bons tempos de resposta interativa, os sistemas devem possuir poucas camadas de software e transferir um volume pequeno de dados entre o cliente e o servidor.

Esses problemas são ilustrados no desempenho de clientes web (navegadores), onde a resposta mais rápida é obtida no acesso às páginas e imagens armazenadas na cache local. As páginas de texto remotas também são acessadas de forma razoavelmente rápida, mas imagens gráficas acarretam atrasos maiores, isso ocorre devido ao volume de dados envolvido em um ou outro caso.

Taxa de rendimento (*throughtput*): uma medida tradicional da atuação de sistemas de computador é a sua taxa de rendimento – a velocidade com que o trabalho computacional é feito. Estamos interessados na capacidade de um sistema distribuído realizar trabalho para todos os seus usuários. Isso é afetado pelas velocidades de processamento nos clientes e nos servidores e pelas taxas de transferência de dados. Os dados localizados em um servidor remoto devem ser transferidos do processo servidor para o processo cliente, passando por diversas camadas de software nos dois computadores. A taxa de rendimento das camadas de software intervenientes é importante, assim como o da rede.

Balanceamento de carga computacional: um dos objetivos dos sistemas distribuídos é permitir que aplicativos e processos de serviço prossigam concomitantemente, sem disputar os mesmos recursos, e explorem os recursos computacionais disponíveis (processador, memória e recursos de rede). Por exemplo, a capacidade de um computador cliente executar *applets* remove carga de processamento em servidor web, permitindo que ele ofereça um serviço melhor. Um exemplo mais significativo é o uso de vários computadores para realizar um único serviço. Isso é necessário, por exemplo, em alguns servidores web bastante carregados, como os de mecanismos de busca e de grandes sites de comércio eletrônico. Uma forma de se obter o balanceamento de carga é explorar a capacidade do serviço de pesquisa de nomes de domínio DNS de retornar um de vários endereços de computadores para um único nome de domínio (veja a Seção 9.2.3).

* N. de R.T.: Corresponde ao termo *responsiveness*, que significa *ability to respond*, ou seja, capacidade de resposta/reação.

Em alguns casos, o balanceamento da carga pode envolver a migração de uma tarefa parcialmente concluída de um local a outro, em função da mudança de carga de processamento. Isso exige um sistema capaz de suportar essa mudança de processos em execução entre computadores.

Qualidade do serviço ◊ Uma vez que os usuários recebam a funcionalidade que exigem de um serviço, como o serviço de arquivo em um sistema distribuído, podemos questionar a qualidade do serviço fornecido. As principais propriedades não-funcionais dos sistemas que afetam a qualidade do serviço experimentada pelos clientes e usuários são: *confiabilidade*, *segurança* e *desempenho*. A *adaptabilidade* para atender mudanças de configurações de sistema e a disponibilidade de recursos tem sido reconhecida como um aspecto adicional importante da qualidade do serviço. Em [Birman 1996], são discutidos a importância desses aspectos e algumas perspectivas interessantes de seu impacto sobre a qualidade de serviços no projeto de sistemas distribuídos.

Os problemas da confiabilidade e da segurança são essenciais no projeto da maioria dos sistemas computacionais. Eles estão fortemente relacionados com dois de nossos modelos fundamentais: o modelo de falha e o modelo de segurança, apresentados nas Seções 2.3.2 e 2.3.3.

O aspecto de qualidade do serviço foi definido originalmente em termos da reatividade e da taxa de rendimento, mas tem sido redefinido em termos da capacidade de satisfazer garantias temporais, conforme discutido nos próximos parágrafos. Com qualquer das duas definições, o aspecto do desempenho de sistemas distribuídos está fortemente relacionado com o modelo de interação definido na Seção 2.3.1.

Alguns aplicativos manipulam *dados sensíveis ao tempo* – fluxos de dados que são obrigados a serem processados, ou transferidos de um processo a outro, a uma velocidade fixa. Por exemplo, um serviço de exibição de filmes poderia consistir em um programa cliente recuperando um filme a partir de um servidor de vídeo e o apresentando na tela do usuário. Para se obter um resultado satisfatório, os sucessivos quadros do vídeo precisam ser exibidos para o usuário dentro de alguns limites de tempo.

Na verdade, a abreviação QoS (do inglês, *Quality of Service* – qualidade de serviço) tem sido efetivamente utilizada para se referir à capacidade dos sistemas de atender a tais prazos. Sua realização depende da disponibilidade dos recursos de computação e rede necessários, nos momentos apropriados. Isso implica no requisito do sistema de fornecer recursos de computação e comunicação garantidos, suficientes para permitir que os aplicativos concluam cada tarefa a tempo (por exemplo, a tarefa de exibir um quadro de vídeo).

As redes empregadas atualmente para, por exemplo, navegar na web, podem apresentar características de desempenho muito boas, mas quando estão muito carregadas seu desempenho se deteriora significativamente – de modo algum se pode dizer que elas fornecem qualidade de serviço. A QoS também se aplica a sistemas operacionais. Cada recurso importante deve ser reservado pelos aplicativos que exigem QoS e deve existir uma gerência desses recursos que dê essas garantias. Os pedidos de reserva que não podem ser atendidos são rejeitados. Esses problemas serão melhor tratados no Capítulo 17.

Uso de cache e replicação ◊ Os problemas de desempenho comentados anteriormente freqüentemente parecem ser grandes obstáculos para a implantação bem-sucedida de sistemas distribuídos, mas eles são em parte superados pelo uso de replicação de dados e cache. A Seção 2.2.2 apresentou as caches e os servidores *proxies* web, sem discutir como as cópias dos recursos armazenadas na cache podem ser mantidas atualizadas quando o recurso é modificado em um servidor. Para isso, em função de requisitos do próprio serviço, diversos protocolos de consistência de cache são usados; por exemplo, o Capítulo 8 menciona os protocolos usados por dois projetos diferentes de servidores de arquivos. A seguir, o protocolo usado em caches web, que faz parte do protocolo HTTP, é brevemente descrito (ele será melhor detalhado na Seção 4.4).

Protocolo de cache web: tanto os navegadores web quanto os servidores *proxies* armazenam em cache as respostas aos pedidos para servidores web. Portanto, o pedido de um cliente web pode ser atendido por uma resposta armazenada na cache do próprio navegador web ou na de um servidor *proxy* que está entre ele e o servidor web. O protocolo de consistência da cache pode ser configurado para fornecer aos navegadores cópias novas (razoavelmente atualizadas) dos recursos mantidos pelo servidor web. Entretanto, para aumentar o desempenho, a disponibilidade e a capacidade de operação

independente de conexão ao servidor web, essa verificação de dados atualizados pode ser feita em segundo plano (*background*).

Um navegador, ou um servidor *proxy*, pode *validar* uma resposta armazenada em cache, verificando no servidor web de origem se essa informação ainda está atualizada. Se o teste falhar, o servidor web retornará uma resposta atualizada, que é armazenada na cache, no lugar da antiga. Navegadores e servidores *proxies* validam as respostas armazenadas em cache quando os clientes solicitam os recursos correspondentes. Porém eles não precisam realizar uma validação, caso a resposta colocada na cache seja suficientemente nova. Mesmo que um servidor web saiba quando um recurso é atualizado, ele não notifica os navegadores e servidores *proxies* que mantém caches, pois para fazer isso, o servidor web deveria manter um registro de todos os navegadores e servidores *proxies* interessados em cada um de seus recursos. Para permitir que os navegadores e servidores *proxies* determinem se suas respostas armazenadas são antigas, os servidores web atribuem tempos de expiração para seus recursos, estimados, por exemplo, a partir da última vez que foram atualizados. O tempo de expiração pode ser enganador, pois um recurso web pode ser atualizado a qualquer momento. Quando um servidor web responde a um pedido, o tempo de expiração do recurso e a hora atual presente no servidor são anexados na resposta.

Os navegadores e servidores *proxies* armazenam na cache o tempo de expiração e a hora do servidor junto com a resposta. Isso permite que esses, ao receberem futuros pedidos, calculem se uma resposta em cache é considerada ou não como ainda válida. Esse cálculo é feito comparando a idade de uma resposta em cache com o seu tempo de expiração. A *idade* de uma resposta é a soma do tempo em que ela está armazenada na cache com a hora recebida do servidor. Note que esse procedimento independe dos relógios internos do servidor web, do navegador, ou do servidor *proxy*, estarem sincronizados.

Dependabilidade ◊ A dependabilidade é um requisito na maioria dos domínios de aplicação. Ela é crucial, não apenas nas atividades de comando e controle, onde a vida pode estar em jogo, mas também em muitas aplicações comerciais, incluindo eletrônicas, que estão se desenvolvendo rapidamente na Internet, e onde a segurança financeira e dos participantes depende da confiança que se tem nos sistemas que operam. Na Seção 2.1, definimos a dependabilidade dos sistemas de computador como *correção*, *segurança* e *confiabilidade*. Aqui, discutiremos o impacto das necessidades de segurança e confiabilidade na arquitetura de um sistema distribuído, deixando a descrição das técnicas relevantes para sua obtenção para mais adiante no livro. O desenvolvimento de técnicas para verificar ou garantir a correção de programas distribuídos e concorrentes é assunto de muitas pesquisas recentes. Embora alguns resultados promissores já tenham sido obtidos, poucos deles (se houver) estão maduros o suficiente para implantação em aplicações práticas.

Tolerância a falhas: é desejável que os aplicativos continuem a funcionar corretamente na presença de falhas no hardware, no software e na comunicação. A confiabilidade é obtida por intermédio da redundância – o aprovisionamento de vários recursos para que o sistema e o software aplicativo possam se reconfigurar e continuar a executar suas tarefas na presença de falhas. A redundância é dispendiosa e há limites para o seu emprego; portanto, também há limites para o grau de tolerância a falhas que pode ser obtido.

Em relação a arquitetura de um sistema, a redundância implica no uso de vários computadores, nos quais cada componente do sistema possa ser executado, e vários caminhos de comunicação, por meio dos quais as mensagens possam ser transmitidas. Dados e processos podem então ser replicados, quando necessário, para proporcionar o nível exigido de tolerância a falhas. Uma forma comum de redundância é o aprovisionamento de várias réplicas de um item de dados em diferentes computadores – desde que um dos computadores ainda esteja funcionando, os dados podem ser acessados. Algumas aplicações críticas, como os sistemas de controle de tráfego aéreo, exibem uma garantia de tolerância a falhas muito alta, o que envolve um alto custo para manter atualizadas as múltiplas réplicas de dados. O Capítulo 15 discute melhor esse assunto.

Outras formas de redundância são usadas para tornar os protocolos de comunicação confiáveis. Por exemplo, as mensagens são retransmitidas até que uma mensagem de confirmação tenha sido recebida. A confiabilidade dos protocolos subjacentes ao RMI será abordada nos Capítulos 4 e 5.

Segurança: o impacto no projeto de um sistema distribuído do requisito de segurança está ligado à necessidade de colocar dados sigilosos e outros recursos apenas em computadores que possam ser efetivamente seguros contra ataques. Por exemplo, o banco de dados de um hospital contém registros com informações sigilosas de pacientes que só devem ser vistos por certos clínicos, enquanto outros registros podem estar mais disponíveis. Não seria apropriado construir um sistema que, quando acessado, carregasse o registro inteiro de um paciente no computador de um usuário, pois, normalmente, esse tipo de equipamento não constitui um ambiente seguro – os usuários podem executar programas para acessar ou atualizar qualquer parte dos dados armazenados em seus computadores pessoais. Um modelo de segurança que trata de requisitos de segurança mais amplos será apresentado na Seção 2.3.3 e o Capítulo 7 descreverá as principais técnicas disponíveis para se obtê-lo.

2.3 Modelos fundamentais

Todos os modelos de arquitetura de sistemas distribuídos vistos anteriormente, apesar de bastante diferentes, apresentam algumas propriedades fundamentais idênticas. Em particular, todos são compostos de processos que se comunicam por meio do envio de mensagens através de uma rede de computadores. Ainda, é desejável que todos possuam os mesmos requisitos de projeto discutidos na seção anterior, os quais se preocupam principalmente com as características de desempenho e confiabilidade dos processos e das redes de comunicação e com a segurança dos recursos presentes no sistema. Nesta seção, apresentaremos modelos baseados nessas propriedades fundamentais as quais nos permitem ser mais específicos a respeito de características e das falhas e riscos para a segurança que possam apresentar.

Genericamente, um modelo contém apenas os ingredientes essenciais que precisamos considerar para entender e raciocinar a respeito de certos aspectos do comportamento de um sistema. Um modelo de sistema precisa tratar das seguintes questões:

- Quais são as principais entidades presentes no sistema?
- Como elas interagem?
- Quais são as características que afetam seu comportamento individual e coletivo?

O objetivo de um modelo é:

- Tornar explícitas todas as suposições relevantes sobre os sistemas que estamos modelando.
- Fazer generalizações a respeito do que é possível ou impossível, dadas essas suposições. As generalizações podem assumir a forma de algoritmos de propósito geral ou de propriedades desejáveis a serem garantidas. Essas garantias dependem da análise lógica e, onde for apropriado, de prova matemática.

Há muito a lucrar com o fato de saber do que dependem e do que não dependem nossos projetos. Isso nos permite saber se um projeto funcionará se tentarmos implementá-lo em um sistema específico: só precisamos perguntar se nossas suposições são válidas para esse sistema. Além disso, tornando nossas suposições claras e explícitas, podemos provar matematicamente as propriedades do sistema. Essas propriedades valerão então para qualquer sistema que atenda nossas suposições. Finalmente, a partir do momento que abstraímos detalhes específicos, como, por exemplo, o hardware empregado, e nos concentramos apenas em entidades e características comportamentais essenciais do sistema, facilitamos a nossa compreensão do mesmo.

Os aspectos dos sistemas distribuídos que desejamos considerar em nossos modelos fundamentais se destinam a ajudar-nos a discutir e raciocinar sobre:

Interação: a computação é feita por processos; e esses interagem passando mensagens, resultando na comunicação (isto é, fluxo de informações) e na coordenação (sincronização e ordenação das atividades) entre eles. Na análise e no projeto de sistemas distribuídos, nos preocupamos especialmente com essas interações. O modelo de interação deve refletir o fato de que a comunicação ocorre com atrasos que, freqüentemente, têm duração considerável. A precisão com a qual

processos independentes podem ser coordenados é limitada pelos atrasos de comunicação e pela dificuldade de se manter a mesma noção de tempo entre todos os computadores de um sistema distribuído.

Falha: a operação correta de um sistema distribuído é ameaçada quando ocorre uma falha em qualquer um dos computadores em que ele é executado (incluindo falhas de software) ou na rede que os interliga. O modelo de falhas define e classifica as falhas. Isso fornece uma base para a análise de seus efeitos em potencial e para projetar sistemas capazes de tolerar certos tipos de falhas e continuar funcionando corretamente.

Segurança: a natureza modular dos sistemas distribuídos, aliado ao fato de ser desejável que sigam uma filosofia de sistemas abertos, os expõem a ataques de agentes externos e internos. O modelo de segurança define e classifica as formas que tais ataques podem assumir, dando uma base para a análise das possíveis ameaças a um sistema e assim guiar o desenvolvimento de sistemas capazes de resistir a eles.

Para facilitar a discussão e o raciocínio, os modelos apresentados neste capítulo são simplificados omitindo-se grande parte dos detalhes existentes em sistemas reais. O relacionamento desses modelos com sistemas reais e a solução de problemas nesse contexto nos ajudam a revelar os principais assuntos deste livro.

2.3.1 Modelo de interação

A discussão sobre arquiteturas de sistema da Seção 2.2 indica que os sistemas distribuídos são compostos por muitos processos, interagindo de maneiras complexas. Por exemplo:

- Vários processos servidores podem cooperar entre si para fornecer um serviço; os exemplos mencionados anteriormente foram o *Domain Name Service*, que divide e replica seus dados em diferentes servidores na Internet, e o *Network Information Service* da Sun, que mantém cópias replicadas de arquivos de senha em vários servidores de uma rede local.
- Um conjunto de processos *peer-to-peer* pode cooperar entre si para atingir um objetivo comum: por exemplo, um sistema de teleconferência que distribui fluxos de dados de áudio de maneira similar, mas com restrições rigorosas de tempo real.

A maioria dos programadores é familiarizada com o conceito de *algoritmo* – uma seqüência de passos a serem executados para realizar um cálculo desejado. Os programas simples são controlados por algoritmos em que os passos são rigorosamente seqüenciais. O comportamento do programa e o estado das variáveis do programa são determinados por eles. Tal programa é executado por um único processo. Já os sistemas distribuídos são compostos de vários processos, como aqueles delineados anteriormente, o que os torna mais complexos. Seu comportamento e estado podem ser descritos por um *algoritmo distribuído* – uma definição dos passos a serem dados por cada um dos processos que compõem o sistema, *incluindo a transmissão de mensagens entre eles*. As mensagens são enviadas para transferir informações entre processos e para coordenar suas atividades.

Em geral, não é possível prever a velocidade com que cada processo é executado e a sincronização da troca das mensagens entre eles. Também é difícil descrever todos os estados de um algoritmo distribuído, pois é necessário considerar falhas que podem ocorrer em um ou mais dos processos envolvidos ou na própria troca de mensagens.

Em um sistema distribuído, as atividades são realizadas por processos que interagem entre si, porém cada processo tem seu próprio estado que consiste no conjunto de dados que ele pode acessar e atualizar, incluindo suas variáveis de programa. O estado pertencente a cada processo é privativo, isto é, ele não pode ser acessado, nem atualizado, por nenhum outro processo.

Nesta seção, discutiremos dois fatores que afetam significativamente a interação de processos em um sistema distribuído:

- o desempenho da comunicação, que é, freqüentemente, um fator limitante;
- a impossibilidade de manter uma noção global de tempo única.

Desempenho da comunicação ◊ Os canais de comunicação são modelados de diversas maneiras nos sistemas distribuídos; como por exemplo, por uma implementação de fluxos ou pela simples troca de mensagens em uma rede de computadores. A comunicação em uma rede de computadores tem as seguintes características de desempenho relacionadas à latência, largura de banda e *jitter*[*]:

- A *latência* é o atraso decorrido entre o início da transmissão de uma mensagem em um processo remetente e o início da recepção pelo processo destinatário. A latência inclui:
  - O tempo que o primeiro bit de um conjunto de bits transmitido em uma rede leva para chegar ao seu destino. Por exemplo, a latência da transmissão de uma mensagem por meio de um enlace de satélite é o tempo necessário para que um sinal de rádio vá até o satélite e retorne a terra para seu destinatário.
  - O atraso no acesso à rede, que aumenta significativamente quando a rede está muito carregada. Por exemplo, para transmissão Ethernet, a estação remetente espera que a rede esteja livre de tráfego para poder enviar sua mensagem.
  - O tempo de processamento gasto pelos serviços de comunicação do sistema operacional nos processos de envio e recepção, que varia de acordo com a carga momentânea dos computadores.
- A *largura de banda* de uma rede de computadores é o volume total de informações que pode ser transmitido em determinado momento. Quando um grande número de comunicações usa a mesma rede, elas compartilham a largura de banda disponível.
- *Jitter* é a variação no tempo exigida para distribuir uma série de mensagens. O *jitter* é crucial para dados multimídia. Por exemplo, se amostras consecutivas de dados de áudio são reproduzidas com diferentes intervalos de tempo, o som resultante será bastante distorcido.

Relógios de computador e eventos de temporização ◊ Cada computador possui seu próprio relógio interno, o qual pode ser usado pelos processos locais para obter o valor atual da hora. Portanto, dois processos sendo executados em diferentes computadores podem associar indicações de tempo aos seus eventos. Entretanto, mesmo que dois processos leiam seus relógios locais ao mesmo tempo, esses podem fornecer valores diferentes. Isso porque os relógios de computador se desviam de uma base de tempo e, mais importante, suas taxas de desvio diferem entre si. O termo *taxa de desvio do relógio* (*drift*) se refere à quantidade relativa pela qual um relógio de computador difere de um relógio de referência perfeito. Mesmo que os relógios de todos os computadores de um sistema distribuído fossem inicialmente ajustados com o mesmo horário, com o passar do tempo eles variariam entre si significativamente a menos que fossem reajustados.

Existem várias estratégias para corrigir os tempos nos relógios de computador. Por exemplo, os computadores podem usar receptores de rádio para obter leituras de tempo GPS (*Global Positioning System*) que oferece uma precisão de cerca de 1 microssegundo. Entretanto, os receptores GPS não funcionam dentro de prédios, nem o seu custo é justificado para cada computador. Em vez disso, um computador que tenha uma fonte de tempo precisa, como o GPS, pode enviar mensagens de sincronização para os outros computadores da rede. É claro que o ajuste resultante entre os tempos nos relógios locais é afetado pelos atrasos variáveis das mensagens. Para ver uma discussão mais detalhada sobre o desvio e a sincronização de relógio, consulte o Capítulo 12.

Duas variantes do modelo de interação ◊ Em um sistema distribuído é muito difícil estabelecer limites para o tempo que leva a execução dos processos, para a troca de mensagens ou para o desvio do relógio. Dois pontos de vistas opostos fornecem modelos simples: o primeiro é fortemente baseado na idéia de tempo, o segundo não.

Sistemas distribuídos síncronos: Hadzilacos e Toueg [1994] definem um sistema distribuído síncrono como aquele no qual são definidos os seguintes pontos:

- o tempo para executar cada etapa de um processo tem limites inferior e superior conhecidos;

[*] N. de R.T.: *Jitter* é a variação estatística do retardo (atraso) na entrega de dados em uma rede, a qual produz uma recepção não regular dos pacotes. Por não haver uma tradução consagrada para esse termo, preferimos mantê-lo em inglês.

- cada mensagem transmitida em um canal é recebida dentro de um tempo limitado, conhecido;
- cada processo tem um relógio local cuja taxa de desvio do tempo real tem um limite conhecido.

Dessa forma, é possível estimar prováveis limites superior e inferior para o tempo de execução de um processo, para o atraso das mensagens e para as taxas de desvio do relógio em um sistema distribuído. Mas é difícil chegar a valores realistas e dar garantias dos valores escolhidos. A menos que os valores dos limites possam ser garantidos, qualquer projeto baseado nos valores escolhidos não será confiável. Entretanto, modelar um algoritmo como um sistema síncrono pode ser útil para dar uma idéia sobre como ele se comportará em um sistema distribuído real. Em um sistema síncrono, é possível usar tempos limites para, por exemplo, detectar a falha de um processo, como mostrado na seção sobre o modelo de falha.

Os sistemas distribuídos síncronos podem ser construídos, desde que se garanta que os processos sejam executados de forma a respeitar as restrições temporais impostas. Para isso é preciso alocar os recursos necessários como tempo de processamento, capacidade de rede e limitar o desvio do relógio.

Sistemas distribuídos assíncronos: muitos sistemas distribuídos, como por exemplo, a Internet, são bastante úteis sem apresentarem características síncronas, portanto, precisamos de um modelo alternativo. Um sistema distribuído assíncrono é aquele em que não existem considerações sobre:

- as velocidades de execução de processos – por exemplo, uma etapa do processo pode levar apenas um picossegundo e outra, um século; tudo que pode ser dito é que cada etapa pode demorar um tempo arbitrariamente longo;
- os atrasos na transmissão das mensagens – por exemplo, uma mensagem do processo A para o processo B pode ser enviada em um tempo insignificante e outra pode demorar vários anos. Em outras palavras, uma mensagem pode ser recebida após um tempo arbitrariamente longo;
- as taxas de desvio do relógio – novamente, a taxa de desvio de um relógio é arbitrária.

O modelo assíncrono não faz nenhuma consideração sobre os intervalos de tempo envolvidos em qualquer tipo de execução. A Internet é perfeitamente representada por esse modelo, pois não há nenhum limite intrínseco sobre a carga no servidor ou na rede e, conseqüentemente, sobre quanto tempo demora, por exemplo, para transferir um arquivo usando ftp. Às vezes, uma mensagem de e-mail pode demorar vários dias para chegar. O quadro a seguir ilustra a dificuldade de se chegar a um acordo em um sistema distribuído assíncrono.

Porém, mesmo desconsiderando as restrições de tempo, as vezes, é necessário realizar algum tipo de tratamento para o problema de demoras e atrasos de execução. Por exemplo, embora a web nem sempre possa fornecer uma resposta específica dentro de um limite de tempo razoável, os navegadores são projetados de forma a permitir que os usuários façam outras coisas enquanto esperam. Qualquer solução válida para um sistema distribuído assíncrono também é válida para um sistema síncrono.

Muito freqüentemente, os sistemas distribuídos reais são assíncronos devido à necessidade dos processos de compartilhar tempo de processamento, canais de comunicação e acesso à rede. Por exemplo, se vários processos, de características desconhecidas, compartilharem um processador, então o desempenho resultante de qualquer um deles não poderá ser garantido. Contudo, existem problemas que não podem ser resolvidos para um sistema assíncrono, mas que podem ser tratados quando alguns aspectos de tempo são usados. Um desses problemas é a necessidade de fazer com que cada elemento de um fluxo de dados multimídia seja emitido dentro de um prazo final. Para problemas como esses, é exigido um modelo síncrono. O quadro a seguir ilustra a impossibilidade de sincronizar relógios em um sistema assíncrono.

Ordenação de eventos ◊ Em muitos casos, estamos interessados em saber se um evento (envio ou recepção de uma mensagem) ocorreu em um processo antes, depois ou simultaneamente com outro

Consenso em Pepperland* ◊ Duas divisões do exército de Pepperland, Apple e Orange, estão acampadas no topo de duas colinas próximas. Mais adiante, no vale, estão os invasores Blue Meanies. As divisões de Pepperland estão seguras, desde que permaneçam em seus acampamentos, e elas podem, para se comunicar, enviar mensageiros com toda segurança pelo vale. As divisões de Pepperland precisam concordar sobre qual delas liderará o ataque contra os Blue Meanies e sobre quando o ataque ocorrerá. Mesmo em uma Pepperland assíncrona, é possível concordar sobre quem liderará o ataque. Por exemplo, cada divisão envia o número de seus membros restantes e aquela que tiver mais liderará (se houver empate, a divisão Apple terá prioridade sobre a divisão Orange). Mas quando elas devem atacar? Infelizmente, na Pepperland assíncrona, os mensageiros têm velocidade muito variável. Se, digamos, a divisão Apple enviar um mensageiro com a mensagem "Atacar!", a divisão Orange poderá não recebê-la dentro de, digamos, três horas; ou então, poderá ter recebido em cinco minutos. O problema de coordenação de ataque ainda existe se considerarmos uma Pepperland síncrona, porém as divisões conhecerão algumas restrições úteis: toda mensagem leva pelo menos *min* minutos e no máximo *max* minutos para chegar. Se a divisão que liderará o ataque enviar a mensagem "Atacar!", ele esperará por *min* minutos e depois atacará. A outra divisão, após receber a mensagem, esperará por 1 minuto e depois atacará. É garantido que seu ataque ocorrerá após o da divisão líder, mas não mais do que $(max - min + 1)$ minutos depois dela.

evento em outro processo. Mesmo na ausência da noção de relógio, a execução de um sistema pode ser descrita em termos da ocorrência de eventos e de sua ordem.

Por exemplo, considere o seguinte conjunto de trocas de mensagens, entre um grupo de usuários de e-mail, X, Y, Z e A, em uma lista de distribuição:

1. o usuário X envia uma mensagem com o assunto *Reunião*;
2. os usuários Y e Z respondem, enviando uma mensagem com o assunto *Re: Reunião*.

Seguindo uma linha de tempo, a mensagem de X foi enviada primeiro, Y a lê e responde; Z lê a mensagem de X e a resposta de Y, e envia outra resposta fazendo referência às mensagens de X e de Y. Mas, devido aos diferentes atrasos envolvidos na distribuição das mensagens, elas podem ser entregues como ilustrado na Figura 2.8, e alguns usuários poderão ver essas duas mensagens na ordem errada; por exemplo, o usuário A poderia ver:

Caixa de entrada:

Item	*De*	*Assunto*
23	Z	Re: Reunião
24	X	Reunião
25	Y	Re: Reunião

Se os relógios nos computadores de X, de Y e de Z pudessem ser sincronizados, então cada mensagem, ao ser enviada, poderia transportar o tempo do relógio de seu computador local. Por exemplo, as mensagens m_1, m_2 e m_3 transportariam os tempos t_1, t_2 e t_3, onde $t_1 < t_2 < t_3$. As mensagens recebidas

* N. de R.T.: O consenso entre partes é um problema clássico em sistemas distribuídos. Os nomes foram mantidos em inglês para honrar sua origem. Pepperland é uma cidade imaginária do desenho animado intitulado *Yellow Submarine*, protagonizado em clima psicodélico pelos Beatles. Em Pepperland, seus pacíficos habitantes se divertem escutando a música da banda Sgt. Pepers Lonely Hearts Club. Entretanto, lá também habitam os Blue Meanies que encolhem ao escutar o som da música e assim atacam Pepperland acabando com a música e transformando todos em estátuas de pedra. É quando Lord Mayor consegue escapar e buscar ajuda, embarcando no submarino amarelo. Os Beatles entram em ação para enfrentar os Blue Meanies.

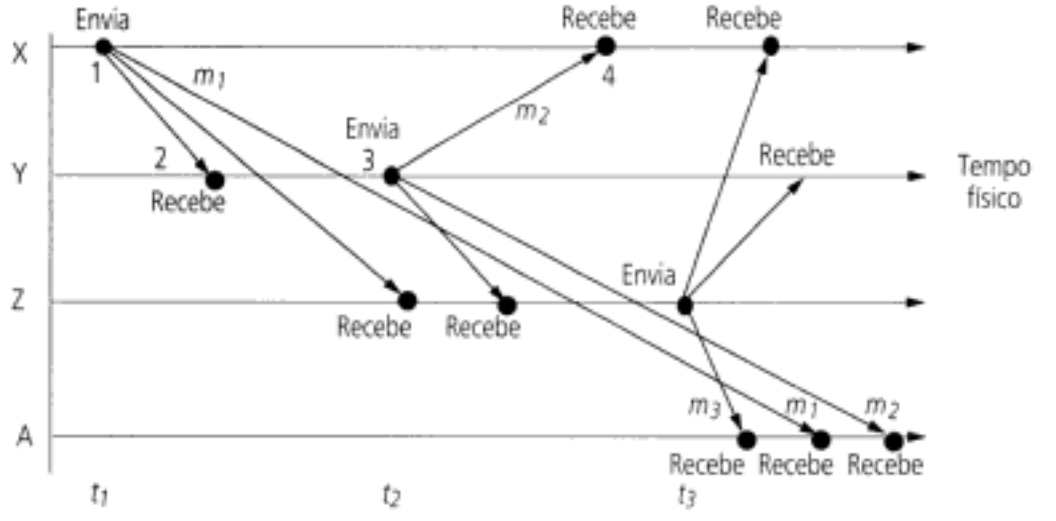


Figura 2.8 Ordenação de eventos no tempo físico.

seriam exibidas para os usuários de acordo com sua ordem temporal de emissão. Se os relógios estiverem aproximadamente sincronizados, então essas indicações de tempo freqüentemente estarão na ordem correta.

Como em um sistema distribuído os relógios não podem ser perfeitamente sincronizados, Lamport [1978] propôs um modelo de *relógio lógico* que pode ser usado para proporcionar uma ordenação de eventos ocorridos em processos executados em diferentes computadores. O relógio lógico permite deduzir a ordem que as mensagens devem ser apresentadas, sem apelar para os relógios físicos de cada máquina. O modelo de relógio lógico será apresentado com detalhes no Capítulo 11, mas comentamos, aqui, como alguns aspectos da ordenação lógica podem ser aplicados ao nosso problema de ordenação de e-mail.

Logicamente, sabemos que uma mensagem é recebida após ser enviada; portanto, podemos expressar a ordenação lógica de pares de eventos mostrada na Figura 2.8, por exemplo, considerando apenas os eventos relativos a X e Y:

X envia m_1 antes que Y receba m_1; Y envia m_2 antes que X receba m_2.

Também sabemos que as respostas são enviadas após o recebimento das mensagens; portanto, temos a seguinte ordenação lógica para Y:

Y recebe m_1 antes de enviar m_2.

O relógio lógico leva essa idéia mais adiante, atribuindo a cada evento um número correspondente à sua ordem lógica, de modo que os eventos posteriores tenham números mais altos do que os anteriores. Por exemplo, a Figura 2.8 mostra os números 1 a 4 para os eventos de X e Y.

2.3.2 Modelo de falhas

Em um sistema distribuído, tanto os processos como os canais de comunicação podem falhar – isto é, eles podem divergir do que é considerado um comportamento correto ou desejável. O modelo de falhas define como uma falha pode se manifestar em um sistema de forma a proporcionar um entendimento dos seus efeitos e conseqüências. Hadzilacos e Toueg [1994] fornecem uma taxonomia que distingue entre as falhas de processos e de canais de comunicação. Isso é apresentado sob os títulos falhas por omissão, falhas arbitrárias e falhas de sincronização.

O modelo de falhas será usado por todo o livro. Por exemplo:

- No Capítulo 4, apresentaremos as interfaces Java para comunicações baseadas em datagrama e por fluxo (*stream*), que proporcionam diferentes graus de confiabilidade.

- O Capítulo 4 apresentará o protocolo requisição-resposta (*request-reply*), que suporta RMI. Suas características de falhas dependem tanto dos processos como dos canais de comunicação. O protocolo requisição-resposta pode ser construído sobre datagramas ou *streams*. A decisão é feita considerando aspectos de simplicidade de implementação, desempenho e confiabilidade.
- O Capítulo 14 apresentará o protocolo de efetivação (*commit*) de duas fases para transações. Ele é projetado de forma a ser concluído na presença de falhas bem definidas de processos e canais de comunicação.

Falhas por omissão ◊ As falhas classificadas como *falhas por omissão* se referem aos casos em que um processo ou canal de comunicação deixa de executar as ações que deveria.

Falhas por omissão de processo: a principal falha por omissão de um processo é quando ele entra em colapso, parando e não executando outro passo de seu programa. Popularmente, isso é conhecido como "dar pau" ou "pendurar". O projeto de serviços que podem sobreviver na presença de falhas pode ser simplificado, caso se possa supor que os serviços dos quais dependem colapsam de modo limpo, isto é, ou os processos funcionam corretamente ou param. Outros processos podem detectar essa falha pelo fato do processo deixar repetidamente de responder às mensagens de invocação. Entretanto, esse método de detecção de falhas é baseado no uso de *timeouts* – ou seja, considera a existência de um tempo limite para que uma determinada ação ocorra. Em um sistema assíncrono, a ocorrência de um *timeout* indica apenas que um processo não está respondendo – porém ele pode ter entrado em colapso, estar lento, ou ainda, as mensagens podem não ter chegado.

O colapso de um processo é chamado de *parada por falha* se outros processos puderem detectar, com certeza, a ocorrência dessa situação. Em um sistema síncrono, uma parada por falha ocorre

Detecção de falha ◊ No caso das divisões de Pepperland acampadas no topo das colinas (veja na página 58), suponha que os Blue Meanies tenham afinal força suficiente para atacar e vencer uma das divisões, enquanto estiverem acampadas – ou seja, que uma das duas divisões possa falhar. Suponha também que, enquanto não são derrotadas, as divisões regularmente enviam mensageiros para relatar seus *status*. Em um sistema assíncrono, nenhuma das duas divisões pode distinguir se a outra foi derrotada ou se o tempo para que os mensageiros cruzem o vale entre elas é simplesmente muito longo. Em uma Pepperland síncrona, uma divisão pode saber com certeza se a outra foi derrotada, pela ausência de um mensageiro regular. Entretanto, a outra divisão pode ter sido derrotada imediatamente após ter enviado o último mensageiro.

Impossibilidade de chegar a um acordo na presença de falhas ◊ Até agora foi suposto que os mensageiros de Pepperland sempre conseguem cruzar o vale; mas, agora, considere que os Blue Meanies podem capturar qualquer mensageiro e impedir que ele chegue a seu destino. (Devemos supor que é impossível para os Blue Meanies fazer lavagem cerebral nos mensageiros para transmitirem a mensagem errada – os Meanies desconhecem seus traiçoeiros precursores: os generais bizantinos*.) As divisões Apple e Orange podem enviar mensagens para que ambas decidam atacar os Meanies ou que decidam se render? Infelizmente, conforme provou o teórico de Pepperland, Ringo, o Grande, nessas circunstâncias, as divisões não podem garantir a decisão correta do que fazer. Para entender como isso acontece, suponha o contrário, que as divisões executem um protocolo Pepperland de consenso: cada divisão propõe "Atacar!" ou "Render!" e, através de mensagens, finaliza com as divisões concordando com uma ou outra ação. Agora, considere que o mensageiro que transporta a última mensagem foi capturado pelos Blue Meanies, mas que isso, de alguma forma, não afete a decisão final de atacar ou se render. Nesse momento, a penúltima mensagem se tornou a última. Se, sucessivamente, aplicarmos o argumento que o último mensageiro foi capturado chegaremos a situação em que nenhuma mensagem foi entregue. Isso mostra que não pode existir nenhum protocolo que garanta o acordo entre as divisões de Pepperland, caso os mensageiros possam ser capturados.

* N. de R.T.: Referência ao problema dos generais bizantinos, no qual as divisões devem chegar a um consenso sobre atacar ou recuar, porém, existem generais que são traidores.

quando *timeouts* são usados para determinar que certos processos deixaram de responder a mensagens sabidamente entregues. Por exemplo, se os processos p e q estiverem programados para q responder a uma mensagem de p e, se o processo p não receber nenhuma resposta do processo q dentro de um tempo máximo (*timeout*), medido no relógio local de p, então o processo p poderá concluir que o processo q falhou. O quadro anterior ilustra a dificuldade para se detectar falhas em um sistema assíncrono ou de se chegar a um acordo na presença de falhas.

Falhas por omissão na comunicação: considere as primitivas de comunicação *send* e *receive*. Um processo p realiza um *send* inserindo a mensagem m em seu buffer de envio. O canal de comunicação transporta m para o buffer de recepção q. O processo q realiza uma operação *receive* recuperando m de seu buffer de recepção (veja a Figura 2.9). Normalmente, os buffers de envio e de recepção são fornecidos pelo sistema operacional.

O canal de comunicação produz uma falha por omissão quando não concretiza a transferência de uma mensagem m do buffer de envio de p para o buffer de recepção de q. Isso é conhecido como "perda de mensagens" e geralmente é causado pela falta de espaço no buffer de recepção, ou pela mensagem ser descartada ao ser detectado que houve um erro durante sua transmissão (isso é feito através de soma de verificação sobre os dados que compõem a mensagem, como por exemplo, cálculo de CRC). Hadzilacos e Toueg [1994] se referem à perda de mensagens entre o processo remetente e o buffer de envio como *falhas por omissão de envio*; à perda de mensagens entre o buffer de recepção e o processo destino como *falhas por omissão de recepção*; e à perda de mensagens no meio de comunicação como *falhas por omissão de canal*. Na Figura 2.10, as falhas por omissão estão classificadas junto com as falhas arbitrárias.

As falhas podem ser classificadas de acordo com sua gravidade. Por enquanto, todas as falhas descritas até aqui são consideradas *benignas*. A maioria das falhas nos sistemas distribuídos é benigna, as quais incluem as falhas por omissão, as de sincronização e as de desempenho.

Falhas arbitrárias ◊ O termo falha *arbitrária*, ou bizantina, é usado para descrever a pior semântica de falha possível na qual qualquer tipo de erro pode ocorrer. Por exemplo, um processo pode atribuir valores incorretos a seus dados ou retornar um valor errado em resposta a uma invocação.

Uma falha arbitrária de um processo é aquela em que ele omite arbitrariamente passos desejados do processamento ou efetua processamento indesejado. Portanto, as falhas arbitrárias não podem ser detectadas verificando-se se o processo responde às invocações, pois ele poderia omitir arbitrariamente a resposta.

Os canais de comunicação podem sofrer falhas arbitrárias; por exemplo, o conteúdo da mensagem pode ser corrompido, mensagens inexistentes podem ser enviadas ou mensagens reais podem ser entregues mais de uma vez. As falhas arbitrárias dos canais de comunicação são raras, pois o software de comunicação é capaz de reconhecê-las e rejeitar as mensagens com problemas. Por exemplo, somas de verificação são usadas para detectar mensagens corrompidas e números de seqüência de mensagem podem ser usados para detectar mensagens inexistentes ou duplicadas.

Figura 2.9 Processos e canais de comunicação.

Classe da falha	*Afeta*	*Descrição*
Parada por falha	Processo	O processo pára e permanece parado. Outros processos podem detectar esse estado.
Colapso	Processo	O processo pára e permanece parado. Outros processos podem não detectar esse estado.
Omissão	Canal	Uma mensagem inserida em um buffer de envio nunca chega no buffer de recepção do destinatário.
Omissão de envio	Processo	Um processo conclui um envio, mas a mensagem não é colocada em seu buffer de envio.
Omissão de recepção	Processo	Uma mensagem é colocada no buffer de recepção de um processo, mas esse processo não a recebe efetivamente.
Arbitrária (bizantina)	Processo ou canal	O processo/canal exibe comportamento arbitrário: ele pode enviar/transmitir mensagens arbitrárias em qualquer momento, cometer omissões; um processo pode parar ou realizar uma ação incorreta.

Figura 2.10 Falhas por omissão e arbitrárias.

Falhas de temporização ◊ As falhas de temporização são aplicáveis aos sistemas distribuídos síncronos onde limites de tempo são estabelecidos para o tempo de execução do processo, para o tempo de entrega de mensagens e para a taxa de desvio do relógio. As falhas de temporização estão listadas na Figura 2.11. Qualquer uma dessas falhas pode resultar em indisponibilidade de respostas para os clientes dentro de um intervalo de tempo predeterminado.

Em um sistema distribuído assíncrono, um servidor sobrecarregado pode responder muito lentamente, mas não podemos dizer que ele apresenta uma falha de temporização, pois nenhuma garantia foi oferecida.

Os sistemas operacionais de tempo real são projetados visando o fornecimento de garantias de temporização, mas seu projeto é mais complexo e pode exigir hardware redundante. A maioria dos sistemas operacionais de propósito geral, como o UNIX, não precisa satisfazer restrições de tempo real.

A temporização é particularmente relevante para aplicações multimídia, com canais de áudio e vídeo. As informações de vídeo podem exigir a transferência de um volume de dados muito grande. Distribuir tais informações sem falhas de temporização pode impor exigências muito especiais sobre o sistema operacional e sobre o sistema de comunicação.

Mascaramento de falhas ◊ Cada componente em um sistema distribuído geralmente é construído a partir de um conjunto de outros componentes. É possível construir serviços confiáveis a partir de componentes que exibem falhas. Por exemplo, vários servidores que contêm réplicas dos dados podem continuar a fornecer um serviço quando um deles apresenta um defeito. O conhecimento das características da falha de um componente pode permitir que um novo serviço seja projetado de forma a mascarar a falha dos componentes dos quais ele depende. Um serviço *mascara* uma falha ocultando-a

Classe da falha	*Afeta*	*Descrição*
Relógio	Processo	O relógio local do processo ultrapassa os limites de sua taxa de desvio em relação ao tempo físico.
Desempenho	Processo	O processo ultrapassa os limites do intervalo de tempo entre duas etapas.
Desempenho	Canal	A transmissão de uma mensagem demora mais do que o limite definido.

Figura 2.11 Falhas de temporização.

completamente ou convertendo-a em um tipo de falha mais aceitável. Como um exemplo desta última opção, somas de verificação são usadas para mascarar mensagens corrompidas – convertendo uma falha arbitrária em falha por omissão. Nos Capítulos 3 e 4, veremos que as falhas por omissão podem ser ocultas usando-se um protocolo que retransmite as mensagens que não chegam ao seu destino. O Capítulo 15 apresentará o mascaramento feito por meio da replicação. Até o colapso de um processo pode ser mascarado – criando-se um novo processo e restaurando, a partir de informações previamente gravadas em disco, o estado da memória de seu predecessor.

Confiabilidade da comunicação de um para um ◊ Embora um canal de comunicação possa exibir as falhas por omissão descritas anteriormente, é possível usá-lo para construir um serviço de comunicação que mascare algumas dessas falhas.

O termo *comunicação confiável* é definido em termos de validade e integridade, como segue:

validade: qualquer mensagem do buffer de envio é entregue ao buffer de recepção de seu destino, independentemente do tempo necessário para tal;

integridade: a mensagem recebida é idêntica à enviada e nenhuma mensagem é entregue duas vezes.

As ameaças à integridade vêm de duas fontes independentes:

- Qualquer protocolo que retransmita mensagens, mas não rejeite uma mensagem que entregue duas vezes. Os protocolos podem incluir números de seqüência nas mensagens para detectar aquelas que são entregues duplicadas.
- Usuários mal-intencionados que podem injetar mensagens espúrias, reproduzir mensagens antigas ou falsificar mensagens. Medidas de segurança podem ser tomadas para manter a propriedade da integridade diante de tais ataques.

2.3.3 Modelo de segurança

Na Seção 2.2, identificamos o compartilhamento de recursos como um fator motivador para os sistemas distribuídos e descrevemos sua arquitetura de sistema em termos de processos encapsulando objetos e fornecendo acesso a eles por meio de interações com outros processos. Esse princípio de funcionamento fornece a base de nosso modelo de segurança:

> a segurança de um sistema distribuído pode ser obtida tornando seguros os processos e os canais usados por suas interações e protegendo contra acesso não autorizado os objetos que encapsulam.

A proteção é descrita em termos de objetos, embora os conceitos se apliquem igualmente bem a qualquer tipo de recursos.

Proteção de objetos ◊ A Figura 2.12 mostra um servidor que gerencia um conjunto de objetos para alguns usuários. Os usuários podem executar programas clientes que enviam invocações para o servidor a fim de realizar operações sobre os objetos. O servidor executa a operação especificada em cada invocação e envia o resultado para o cliente.

Figura 2.12 Objetos e principais.

Os objetos são usados de diversas formas, por diferentes usuários. Por exemplo, alguns objetos podem conter dados privativos de um usuário, como sua caixa de correio, e outros podem conter dados compartilhados, como suas páginas web. Para dar suporte a isso, *direitos de acesso* especificam quem pode executar determinadas operações sobre um objeto – por exemplo, quem pode ler ou gravar seu estado.

Dessa forma, os usuários devem ser incluídos em nosso modelo de segurança como os beneficiários dos direitos de acesso. Fazemos isso associando a cada invocação, e a cada resultado, o tipo de autorização de quem a executa. Tal autorização é chamada de *principal*. Um principal pode ser um usuário ou um processo. Em nossa ilustração, a invocação vem de um usuário e o resultado, de um servidor.

O servidor é responsável por verificar a identidade do principal que está por trás de cada invocação e conferir se ele tem direitos de acesso suficientes para efetuar a operação solicitada em determinado objeto, rejeitando as que não pode efetuar. O cliente pode verificar a identidade do principal que está por trás do servidor, para garantir que o resultado seja realmente enviado por esse servidor.

Tornando processos e suas interações seguros ◊ Os processos interagem enviando mensagens. As mensagens ficam expostas a ataques, porque o acesso à rede e ao serviço de comunicação é livre para permitir que quaisquer dois processos interajam. Servidores e processos *peer-to-peer* publicam suas interfaces, permitindo que invocações sejam enviadas a eles por qualquer outro processo.

Freqüentemente, os sistemas distribuídos são implantados e usados em tarefas que provavelmente estarão sujeitas a ataques externos realizados por usuários mal-intencionados. Isso é especialmente verdade para aplicativos que manipulam transações financeiras, informações confidenciais ou secretas, ou qualquer outro tipo de informação cujo segredo ou integridade seja crucial. A integridade é ameaçada por violações de segurança, assim como por falhas na comunicação. Portanto, sabemos que existem prováveis ameaças aos processos que compõem os aplicativos e as mensagens que trafegam entre eles. Mas, como podemos analisar essas ameaças para identificá-las e anulá-las? A discussão a seguir apresenta um modelo para a análise de ameaças à segurança.

O invasor ◊ Para modelar as ameaças à segurança, postulamos um invasor (às vezes também conhecido como atacante) capaz de enviar qualquer mensagem para qualquer processo e ler ou copiar qualquer mensagem entre dois processos, como se vê na Figura 2.13. Tais ataques podem ser realizados usando-se simplesmente um computador conectado a uma rede para executar um programa que lê as mensagens endereçadas para outros computadores da rede, ou por um programa que gere mensagens que façam falsos pedidos para serviços e dêem a entender que sejam provenientes de usuários autorizados. O ataque pode vir de um computador legitimamente conectado à rede, ou de um que esteja conectado de maneira não autorizada.

As ameaças de um atacante em potencial são discutidas sob os títulos *ameaças aos processos*, *ameaças aos canais de comunicação* e *negação de serviço*.

Ameaças aos processos: um processo projetado para tratar de pedidos pode receber uma mensagem de qualquer outro processo no sistema distribuído e não ser capaz de determinar com certeza a identidade do remetente. Os protocolos de comunicação, como o IP, incluem o endereço do computador

Figura 2.13 O invasor (atacante).

de origem em cada mensagem, mas não é difícil para um atacante gerar uma mensagem com um endereço de origem falsificado. Essa falta de reconhecimento confiável da origem de uma mensagem é, conforme explicado a seguir, uma ameaça ao correto funcionamento tanto de servidores como de clientes:

Servidores: como um servidor pode receber pedidos de muitos clientes diferentes, ele não pode necessariamente determinar a identidade do principal que está por trás de uma invocação em particular. Mesmo que um servidor exija a inclusão da identidade do principal em cada invocação, um atacante poderia gerá-la com uma identidade falsa. Sem o reconhecimento garantido da identidade do remetente, um servidor não pode saber se deve executar a operação ou rejeitá-la. Por exemplo, um servidor de correio eletrônico que recebe de um usuário uma solicitação de leitura de mensagens de uma caixa de correio eletrônico em particular, pode não saber se esse usuário está autorizado a fazer isso ou se é uma solicitação indevida.

Clientes: quando um cliente receber o resultado de uma invocação feita a um servidor, ele não pode necessariamente identificar se a origem da mensagem com o resultado é o servidor desejado ou um invasor, talvez fazendo *spoofing* desse servidor. O *spoofing* é, na prática, o roubo de identidade. Assim, um cliente poderia receber um resultado não relacionado à invocação original, como por exemplo, uma mensagem de correio eletrônico falsa (que não está na caixa de correio do usuário).

Ameaças aos canais de comunicação: um invasor pode copiar, alterar ou injetar mensagens quando elas trafegam pela rede e em seus sistemas intermediários (roteadores, por exemplo). Tais ataques representam uma ameaça à privacidade e à integridade das informações quando elas trafegam pela rede, e a própria integridade do sistema. Por exemplo, uma mensagem com resultado contendo um correio eletrônico de um usuário poderia ser revelada a outro, ou ser alterada para dizer algo totalmente diferente.

Outra forma de ataque é a tentativa de salvar cópias de mensagens e reproduzi-las posteriormente, tornando possível reutilizar a mesma mensagem repetidamente. Por exemplo, alguém poderia tirar proveito, reenviando uma mensagem de invocação, solicitando uma transferência de um valor em dinheiro de uma conta bancária para outra.

Todas essas ameaças podem ser anuladas com o uso de *canais de comunicação seguros*, que estão descritos a seguir e são baseados em criptografia e autenticação.

Anulando ameaças à segurança ◊ Apresentamos aqui as principais técnicas nas quais os sistemas seguros são baseados. O Capítulo 7 discutirá com alguns detalhes o projeto e a implementação de sistemas distribuídos seguros.

Criptografia e segredos compartilhados: suponha que dois processos (por exemplo, um cliente e um servidor) compartilhem um segredo; isto é, ambos conhecem o segredo, mas nenhum outro processo no sistema distribuído sabe dele. Então, se uma mensagem trocada por esses dois processos incluir informações que provem o conhecimento do segredo compartilhado por parte do remetente, o destinatário saberá com certeza que o remetente foi o outro processo do par. É claro que se deve tomar os cuidados necessários para garantir que o segredo compartilhado não seja revelado a um invasor.

Criptografia é a ciência de manter as mensagens seguras e *cifrar* é o processo de embaralhar uma mensagem de maneira a ocultar seu conteúdo. A criptografia moderna é baseada em algoritmos que utilizam chaves secretas – números grandes difíceis de adivinhar – para transformar os dados de uma maneira que só possam ser revertidos com o conhecimento da chave de *decifração* correspondente.

Autenticação: o uso de segredos compartilhados e da criptografia fornece a base para a *autenticação* de mensagens – provar as identidades de seus remetentes. A técnica de autenticação básica é incluir em uma mensagem uma parte cifrada que possua conteúdo suficiente para garantir sua autenticidade. A autenticação de um pedido de leitura de um trecho de um arquivo enviado a um servidor de arquivos poderia, por exemplo, incluir uma representação da identidade do principal que está fazendo a solicitação, a identificação do arquivo e a data e hora do pedido, tudo cifrado com uma chave secreta compartilhada entre o servidor de arquivos e o processo solicitante. O servidor decifraria o pedido e verificaria se os mesmos correspondem realmente ao pedido.

Canais seguros: criptografia e autenticação são usadas para construir canais seguros como uma camada de serviço a mais sobre os serviços de comunicação já existentes. Um canal seguro é um canal de comunicação conectando dois processos, cada um dos quais atuando em nome de um principal, como se vê na Figura 2.14. Um canal seguro tem as seguintes propriedades:

- Cada um dos processos conhece com certeza a identidade do principal em nome de quem o outro processo está executando. Portanto, se um cliente e um servidor se comunicam por meio de um canal seguro, o servidor conhece a identidade do principal que está por trás das invocações e pode verificar seus direitos de acesso, antes de executar uma operação. Isso permite que o servidor proteja corretamente seus objetos e que o cliente tenha certeza de que está recebendo resultados de um servidor *fidedigno*.
- Um canal seguro garante a privacidade e a integridade (proteção contra falsificação) dos dados transmitidos por ele.
- Cada mensagem inclui uma indicação de relógio lógico, ou físico, para impedir que as mensagens sejam reproduzidas ou reordenadas.

A construção de canais seguros será discutida em detalhes no Capítulo 7. Os canais seguros têm se tornado uma importante ferramenta prática para proteger o comércio eletrônico e para a proteção de comunicações em geral. As redes virtuais privativas (VPNs – *Virtual Private Networks*, discutidas no Capítulo 3) e o protocolo SSL (*Secure Sockets Layer*) (discutido no Capítulo 7) são exemplos.

Outras ameaças possíveis ◊ A Seção 1.4.3 apresentou, muito sucintamente, duas ameaças à segurança – ataques de negação de serviço e a utilização de código móvel. Reiteramos essas ameaças como possíveis oportunidades para o invasor romper as atividades dos processos:

Negação de serviço: esta é uma forma de ataque na qual o atacante interfere nas atividades dos usuários autorizados, fazendo inúmeras invocações sem sentido em serviços ou transmitindo mensagens incessantemente em uma rede para gerar uma sobrecarga dos recursos físicos (capacidade de processamento do servidor, largura de banda da rede, etc). Tais ataques normalmente são feitos com a intenção de retardar, ou impedir, as invocações válidas de outros usuários. Por exemplo, a operação de trancas eletrônicas de portas em um prédio poderia ser desativada por um ataque que saturasse o computador que controla as trancas com pedidos inválidos.

Código móvel: o código móvel levanta novos e interessantes problemas de segurança para qualquer processo que receba e execute código proveniente de outro lugar, como o anexo de correio eletrônico mencionado na Seção 1.4.3. Esse código pode desempenhar facilmente o papel de cavalo de Tróia, dando a entender que vai cumprir um propósito inocente, mas que na verdade inclui código que acessa ou modifica recursos legitimamente disponíveis para o usuário que o executa. Os métodos pelos quais tais ataques podem ser realizados são muitos e variados e, para evitá-los, o ambiente que recebe tais códigos deve ser construído com muito cuidado. Muitos desses problemas foram resolvidos com a utilização de Java e em outros sistemas de código móvel, mas a história recente desse assunto inclui algumas vulnerabilidades embaraçosas. Isso ilustra bem a necessidade de uma análise rigorosa no projeto de todos os sistemas seguros.

O uso dos modelos de segurança ◊ Pode-se pensar que a obtenção de segurança em sistemas distribuídos seria uma questão simples, envolvendo o controle do acesso a objetos de acordo com direitos

Figura 2.14 Canais seguros.

de acesso predefinidos e com o uso de canais seguros para comunicação. Infelizmente, geralmente esse não é o caso. O uso de técnicas de segurança como a criptografia e o controle de acesso acarreta custos de processamento e de gerenciamento substanciais. O modelo de segurança delineado anteriormente fornece a base para a análise e o projeto de sistemas seguros, onde esses custos são mantidos em um mínimo. Entretanto, as ameaças a um sistema distribuído surgem em muitos pontos e é necessária uma análise cuidadosa das ameaças que podem surgir de todas as fontes possíveis no ambiente de rede, no ambiente físico e no ambiente humano do sistema. Essa análise envolve a construção de um *modelo de ameaças*, listando todas as formas de ataque que o sistema está exposto e uma avaliação dos riscos e conseqüências de cada um. A eficácia e o custo das técnicas de segurança necessárias podem então ser ponderadas em relação às ameaças.

2.4 Resumo

A maioria dos sistemas distribuídos é organizada com base em um, entre vários, modelos de arquitetura de sistema. O modelo cliente-servidor predomina – a web e outros serviços de Internet, como ftp, *news* e correio eletrônico, assim como serviços web e o DNS, são baseados nesse modelo, sem mencionar outros serviços locais. Serviços como o DNS, que têm grande número de usuários e gerenciam muitas informações, são baseados em múltiplos servidores e utilizam o particionamento de dados e a replicação para melhorar a disponibilidade e a tolerância a falhas. O uso de cache por clientes e servidores *proxies* é amplamente empregado para melhorar o desempenho de um serviço.

No modelo *peer-to-peer*, para realizar uma tarefa comum, todos os processos desempenham funções semelhantes na exploração dos recursos disponibilizados por um grande número de computadores participantes.

A capacidade de mover código de um processo para outro tem resultado em algumas variantes do modelo cliente-servidor. O exemplo mais comum é o *applet*, cujo código é fornecido por um servidor web para ser executado por um cliente, permitindo a realização de funcionalidades não disponíveis no cliente e um melhor desempenho, pelo fato de ser executado próximo ao usuário eliminando o atraso na rede e a carga no servidor.

A existência de computadores portáteis, PDAs e outros equipamentos digitais e sua integração nos sistemas distribuídos, permite que os usuários acessem serviços locais e de Internet quando estão longe de seus computadores de trabalho. Uma característica dos dispositivos móveis em um sistema distribuído é que eles se conectam e desconectam de forma imprevisível. Isso leva à variante do modelo cliente-servidor conhecida como interoperabilidade espontânea, na qual as associações entre os dispositivos são freqüentemente criadas e destruídas.

Apresentamos os modelos de interação, falha e segurança. Eles identificam as características comuns dos componentes básicos a partir dos quais os sistemas distribuídos são construídos. O modelo de interação se preocupa com o desempenho dos processos dos canais de comunicação e com a ausência de um relógio global. Ele identifica um sistema síncrono como aquele em que podem ser impostos limites conhecidos para o tempo de execução de um processo, para o tempo de entrega de mensagens e para o desvio do relógio. Ele identifica um sistema assíncrono como aquele em que nenhum limite pode ser imposto para o tempo de execução de um processo, para o tempo de entrega de mensagens e para o desvio do relógio. O comportamento da Internet segue esse modelo.

O modelo de falha classifica as falhas de processos e dos canais de comunicação básicos em um sistema distribuído. O mascaramento é uma técnica por meio da qual um serviço mais confiável é construído a partir de outro menos confiável, escondendo algumas das falhas que ele exibe. Em particular, um serviço de comunicação confiável pode ser construído a partir de um canal de comunicação básico por meio do mascaramento de suas falhas. Por exemplo, suas falhas por omissão podem ser mascaradas pela retransmissão das mensagens perdidas. A integridade é uma propriedade da comunicação confiável – ela exige que uma mensagem recebida seja idêntica àquela que foi enviada e que nenhuma mensagem seja enviada duas vezes. A validade é outra propriedade – ela exige que toda mensagem colocada em um buffer de envio seja entregue no buffer de recepção de um destinatário.

O modelo de segurança identifica as possíveis ameaças aos processos e canais de comunicação em um sistema distribuído aberto. Algumas dessas ameaças se relacionam com a integridade: usuários

mal-intencionados podem falsificar mensagens ou reproduzi-las. Outras ameaçam sua privacidade. Outro problema de segurança é a autenticação do principal (usuário ou servidor) em nome de quem uma mensagem foi enviada. Os canais seguros usam técnicas de criptografia para garantir a integridade, a privacidade das mensagens e para autenticar mutuamente os pares de principais que estejam se comunicando.

Exercícios

2.1 Descreva e ilustre a arquitetura cliente-servidor de um ou mais aplicativos de Internet importantes (por exemplo, a web, correio eletrônico ou *news*). *página 43–44*

2.2 Para os aplicativos discutidos no Exercício 2.1 diga como os servidores cooperam no fornecimento de um serviço. *página 44–46*

2.3 Como os aplicativos discutidos no Exercício 2.1 envolvem o particionamento e/ou a replicação (ou armazenamento em cache) dos dados entre os servidores? *página 44–46*

2.4 Um mecanismo de busca é um servidor web que responde aos pedidos do cliente para pesquisar em seus índices armazenados e (concomitantemente) executa várias tarefas de *web crawling* para construir e atualizar esses índices. Quais são os requisitos de sincronização entre essas atividades concomitantes? *página 43–44*

2.5 Freqüentemente, os computadores usados nos sistemas *peer-to-peer* são computadores *desktop* nos escritórios ou nas casas dos usuários. Quais são as implicações disso na disponibilidade e segurança dos objetos de dados compartilhados que eles contêm e até que ponto qualquer vulnerabilidade pode ser superada por meio da replicação? *página 43, página 64*

2.6 Liste os tipos de recurso local que são vulneráveis a um ataque de um programa não confiável, cujo *download* é feito de um site remoto e que é executado em um computador local. *página 44–46*

2.7 Dê exemplos de aplicações onde o uso de código móvel seja vantajoso. *página 44–46*

2.8 Quais fatores afetam a reatividade de um aplicativo que acessa dados compartilhados gerenciados por um servidor? Descreva as soluções que estão disponíveis e discuta sua utilidade. *página 50–51*

2.9 Diferencie o uso de buffer e cache. *página 52–53*

2.10 Dê alguns exemplos de falhas no hardware e no software que possam/não possam ser toleradas pelo uso de redundância em um sistema distribuído. Até que ponto o uso de redundância nos casos apropriados torna um sistema tolerante à falhas? *página 53–54*

2.11 Considere um servidor simples que executa pedidos do cliente sem acessar outros servidores. Explique por que geralmente não é possível estabelecer um limite para o tempo gasto por tal servidor para responder ao pedido de um cliente. O que precisaria ser feito para tornar o servidor capaz de executar pedidos dentro de um tempo limitado? Essa é uma opção prática? *página 55*

2.12 Para cada um dos fatores que contribuem para o tempo gasto para transmitir uma mensagem entre dois processos por um canal de comunicação, cite quais medidas seriam necessárias para estabelecer um limite para sua contribuição no tempo total. Por que essas medidas não são tomadas nos sistemas distribuídos de propósito geral atuais? *página 55–56*

2.13 O serviço *Network Time Protocol* pode ser usado para sincronizar relógios de computador. Explique por que, mesmo com esse serviço, nenhum limite garantido é dado para a diferença entre dois relógios. *página 55–56*

2.14 Considere dois serviços de comunicação para uso em sistemas distribuídos assíncronos. No serviço A, as mensagens podem ser perdidas, duplicadas ou retardadas e somas de verificação se aplicam apenas aos cabeçalhos. No serviço B, as mensagens podem ser perdidas, retardadas ou entregues rápido demais para o destinatário manipulá-las, mas sempre chegam com o conteúdo correto.

Descreva as classes de falha exibidas para cada serviço. Classifique suas falhas de acordo com seu efeito sobre as propriedades de validade e integridade. O serviço B pode ser descrito como um serviço de comunicação confiável? *página 59–60, página 62–63*

2.15 Considere dois processos X e Y que utilizam o serviço comunicação B do Exercício 2.14 para se comunicar entre si. Suponha que X seja um cliente e que Y seja um servidor e que uma invocação

consiste em uma mensagem de pedido de X para Y, seguida de Y executando o pedido, seguida de uma mensagem de resposta de Y para X. Descreva as classes de falha que podem ser exibidas por uma invocação. *página 59–60*

2.16 Suponha que uma leitura de disco possa às vezes ler valores diferentes dos gravados. Cite os tipos de falha exibidos por uma leitura de disco. Sugira como essa falha pode ser mascarada para produzir uma forma de falha benigna diferente. Agora, sugira como se faz para mascarar a falha benigna. *página 62–63*

2.17 Defina a propriedade de integridade da comunicação confiável e liste todas as possíveis ameaças à integridade de usuários e dos componentes do sistema. Quais medidas podem ser tomadas para garantir a propriedade de integridade diante de cada uma dessas fontes de ameaças. *páginas 62–63, 72–73*

2.18 Descreva as possíveis ocorrências de cada um dos principais tipos de ameaça à segurança (ameaças aos processos, ameaças aos canais de comunicação, negação de serviço) que poderiam ocorrer na Internet. *página 63–64*

3 Redes de Computadores e Interligação em Rede

Os sistemas distribuídos utilizam redes locais, redes de longa distância e redes interligadas para comunicação. O desempenho, a confiabilidade, a escalabilidade, a mobilidade e a qualidade das características do serviço das redes subjacentes afetam o comportamento dos sistemas distribuídos e, assim, têm impacto sobre seu projeto. Mudanças nos requisitos do usuário têm resultado no surgimento de redes sem fio e de redes de alto desempenho com garantia da qualidade do serviço.

Os princípios nos quais as redes de computadores são baseados incluem organização de protocolos em camadas, comutação de pacotes, roteamento e fluxo de dados (*streaming*). As técnicas de interligação em rede possibilitam a integração de redes heterogêneas. A Internet é o maior exemplo; seus protocolos são utilizados quase que universalmente em sistemas distribuídos. Os esquemas de endereçamento e de roteamento usados na Internet têm suportado o impacto de seu enorme crescimento. Atualmente, eles estão sendo revisados para acomodar o crescimento futuro e atender aos novos requisitos de aplicativos quanto a mobilidade, segurança e qualidade do serviço.

O projeto de tecnologias de rede é ilustrado em quatro estudos de caso: Ethernet, IEEE 802.11 (WiFi) e Bluetooth, estas duas últimas, tecnologias de redes sem fio (*wireless*) e, finalmente, redes ATM (*Asynchronous Transfer Mode*).

3.1 Introdução

As redes de comunicação usadas em sistemas distribuídos são construídas a partir de uma variedade de *mídias de transmissão*, que incluem cabos de cobre, fibras ópticas e comunicação sem fio; *dispositivos de hardware*, que englobam roteadores, comutadores (*switches*), pontes, hubs, repetidores e interfaces de rede; e *componentes de software*, como pilhas de protocolo, rotinas de tratamento de comunicação e drivers de dispositivos. A funcionalidade resultante e o desempenho disponível para o sistema distribuído e para os programas aplicativos são afetados por tudo isso. Vamos nos referir ao conjunto de componentes de hardware e software que fornecem os recursos de comunicação para um sistema distribuído como *subsistema de comunicação*. Os computadores e outros dispositivos que utilizam a rede para propósitos de comunicação são referidos como *hosts*. O termo *nó* é usado para se referir a qualquer computador ou dispositivo de comunicação ligado à rede.

A Internet é um subsistema de comunicação único que fornece comunicação entre todos os *hosts* conectados a ela. Ela é construída a partir de muitas *sub-redes*. Uma sub-rede é uma unidade de roteamento (mecanismo de encaminhamento de dados de uma parte da Internet a outra); composta por um conjunto de nós de uma mesma rede física. A infra-estrutura da Internet inclui uma arquitetura e componentes de hardware e software que integram diversas sub-redes em um único serviço de comunicação de dados.

O projeto de um subsistema de comunicação é fortemente influenciado pelas características dos sistemas operacionais empregados pelos computadores que compõem o sistema distribuído, assim como pelas redes que os interligam. Neste capítulo, consideraremos o impacto das tecnologias de rede sobre o subsistema de comunicação; os problemas relacionados aos sistemas operacionais serão discutidos no Capítulo 6.

Este capítulo se destina a fornecer um panorama introdutório das redes de computadores, com referência aos requisitos de comunicação dos sistemas distribuídos. Os leitores que não estiverem familiarizados com redes de computadores devem considerá-lo como uma base para o restante do livro, enquanto aqueles que já estão acostumados, verão que este capítulo oferece um resumo ampliado dos aspectos das redes de computadores particularmente relevantes para os sistemas distribuídos.

As redes de computadores foram concebidas logo após a invenção dos computadores. A base teórica da comutação de pacotes foi apresentada em um artigo de Leonard Kleinrock [1961]. Em 1962, J.C.R. Licklider e W. Clark, que participaram do desenvolvimento do primeiro sistema de tempo compartilhado, no MIT, no início dos anos 60, publicaram um artigo discutindo o potencial da computação interativa e das redes de longa distância que previu a Internet, em vários aspectos [DEC 1990]. Em 1964, Paul Baran produziu o esboço de um projeto prático para redes de longa distância confiáveis e eficientes [Baran 1964]. Mais material e *links* sobre a história das redes de computadores e da Internet podem ser encontrados nas seguintes fontes: [www.isoc.org, Comer 2000b, Kurose e Ross 2000].

No restante desta seção, discutiremos os requisitos de comunicação dos sistemas distribuídos. Forneceremos um panorama dos tipos de rede na Seção 3.2 e uma introdução aos princípios de interligação em rede, na Seção 3.3. A Seção 3.4 trata especificamente da Internet. O capítulo termina com estudos de caso sobre as tecnologias de interligação em rede Ethernet, IEEE 802.11 (WiFi), Bluetooth e ATM, na Seção 3.5.

3.1.1 Problemas de interligação em rede para sistemas distribuídos

As primeiras redes de computadores foram projetadas para atender alguns poucos requisitos de aplicações em rede relativamente simples, como a transferência de arquivos, *login* remoto, correio eletrônico e *newsgroups*. O desenvolvimento subseqüente de sistemas distribuídos, com suporte para programas aplicativos distribuídos, acesso a arquivos compartilhados e outros recursos, estabeleceu um padrão de desempenho mais alto para atender as necessidades das aplicações interativas.

Mais recentemente, acompanhando o crescimento, a comercialização e o emprego de novos usos da Internet, surgiram requisitos de confiabilidade, escalabilidade, mobilidade, segurança e qualidade de serviço mais rigorosos. Nesta seção, definimos e descrevemos a natureza de cada um desses requisitos.

Desempenho ◊ Os parâmetros de desempenho de rede que têm principal interesse para nossos propósitos são aqueles que afetam a velocidade com que as mensagens podem ser transferidas entre dois computadores interligados, são eles: latência e taxa de transferência de dados ponto a ponto.

Latência é o tempo decorrido após uma operação de envio ser executada e antes que os dados comecem a chegar em seu destino. Ela pode ser medida como o tempo necessário para transferir uma mensagem vazia. Aqui estamos considerando apenas a latência da rede, que forma uma parte da latência de processo a processo, definida na Seção 2.3.1.

A *taxa de transferência de dados* é a velocidade com que os dados podem ser transferidos entre dois computadores em uma rede, uma vez que a transmissão tenha começado. É normalmente medida em bits por segundo (bps).

Como resultado dessas definições, o tempo exigido para que uma rede transfira uma mensagem contendo uma certa *largura* de bits entre dois computadores é:

Tempo de transmissão da mensagem = latência + largura/taxa de transferência de dados

A equação acima é válida para mensagens cuja largura não ultrapasse o máximo determinado pela tecnologia de rede empregada para seu envio. Mensagens maiores precisam ser segmentadas (ou fragmentadas) e o tempo de transmissão é a soma dos tempos dos segmentos (fragmentos).

A taxa de transferência de uma rede é determinada principalmente por suas características físicas, enquanto a latência é determinada principalmente pelas sobrecargas do software, pelos atrasos de roteamento e por um fator estatístico dependente da carga da rede, proveniente do conflito de acesso aos canais de transmissão. Em sistemas distribuídos, é comum a transferência de muitas mensagens de pequeno tamanho entre processos; portanto, na determinação do desempenho, a latência freqüentemente assume uma importância igual ou maior do que a taxa de transferência.

A *largura de banda* de uma rede é uma medida de seu rendimento, isto é, o volume total de tráfego que pode ser transferido na rede em um determinado período de tempo. Em muitas tecnologias de rede local, como a Ethernet, a capacidade de transmissão máxima da rede é disponível para cada transmissão e a largura de banda do sistema é igual à taxa de transferência de dados. Mas, na maioria das redes de longa distância, as mensagens podem ser transferidas simultaneamente em diferentes canais e a largura de banda total do sistema não apresenta relacionamento direto com a taxa de transferência. O desempenho das redes se deteriora em condições de sobrecarga – quando existem muitas mensagens ao mesmo tempo na rede. O efeito preciso da sobrecarga sobre a latência, sobre a taxa de transferência de dados e sobre a largura de banda depende muito da tecnologia da rede.

Agora, considere o desempenho de uma comunicação cliente-servidor. O tempo necessário à transmissão de uma mensagem de pedido (curta) e para o recebimento de sua resposta, também curta, em uma rede local pouco carregada (incluindo as cargas de processamento) é de cerca de meio milissegundo. Isso deve ser comparado com o tempo inferior a um microssegundo exigido para uma aplicação invocar uma operação sobre um objeto armazenado na memória local. Assim, a despeito dos avanços no desempenho das redes, o tempo exigido para acessar recursos compartilhados em uma rede local permanece em cerca de mil vezes maior do que o tempo exigido para acessar recursos armazenados em memória local. Porém, a latência e a largura de banda da rede freqüentemente superam o desempenho do disco rígido; o acesso por rede a um servidor web, ou a um servidor de arquivos local que mantêm em cache uma grande quantidade de arquivos freqüentemente utilizados, pode equivaler, ou superar, o acesso aos arquivos armazenados em um disco rígido local.

Na Internet, as latências entre a ida e a volta de mensagens estão na faixa dos 50 a 750 ms, com uma média em torno de 200 ms; portanto, os pedidos transmitidos pela Internet são aproximadamente 100 vezes mais lentos do que nas redes locais mais rápidas. A maior parte dessa diferença de tempo é conseqüência dos atrasos em roteadores e na disputa por circuitos de comunicação da rede.

A Seção 6.5.1 discutirá e comparará o desempenho de operações locais e remotas com mais detalhes.

Escalabilidade ◊ As redes de computadores são uma parte indispensável da infra-estrutura das sociedades modernas. Na Figura 1.4, mostramos o crescimento do número de computadores conectados na Internet em um período de 25 anos. O potencial futuro tamanho da Internet é proporcional à população do planeta. É realístico esperar que ela incluirá vários bilhões de nós e centenas de milhões de computadores.

Esses números indicam as mudanças futuras no tamanho e na carga que a Internet deverá manipular. As tecnologias de rede em que ela é baseada não foram projetadas para suportar a atual escala da Internet, mas têm funcionado notavelmente bem. Para tratar a próxima fase de crescimento da Internet, algumas mudanças substanciais nos mecanismos de endereçamento e roteamento estão em andamento; elas serão descritas na Seção 3.4. Para aplicações cliente-servidor simples, como a web, podemos esperar que o tráfego futuro cresça pelo menos na proporção do número de usuários ativos. A capacidade da infra-estrutura da Internet suportar esse crescimento dependerá de fatores econômicos para seu uso, em especial das cobranças impostas aos usuários e dos padrões de demanda de comunicação que realmente ocorrerem – por exemplo, seu grau de localidade.

Confiabilidade ◊ Nossa discussão sobre os modelos de falhas, na Seção 2.3.2, descreveu o impacto dos erros de comunicação. Muitos aplicativos são capazes de se recuperar de falhas de comunicação e, assim, não exigem a garantia da comunicação isenta de erros. O "argumento do princípio fim-a-fim" (Seção 2.2.1) corrobora a visão de que o subsistema de comunicação não precisa oferecer comunicação totalmente isenta de erros; freqüentemente, a detecção de erros de comunicação e sua correção é melhor realizada por software aplicativo. A confiabilidade da maior parte da mídia de transmissão física é muito alta. Quando ocorrem erros, normalmente devido a falhas no software presente no remetente ou no destinatário (por exemplo, o computador destino deixa de aceitar um pacote) ou devido a estouros de buffer, em vez de erros na rede.

Segurança ◊ O Capítulo 7 estabelecerá os requisitos e técnicas para se obter segurança em sistemas distribuídos. O primeiro nível de defesa adotado pela maioria das organizações é proteger suas redes e os computadores ligados a elas com um *firewall*. Um *firewall* cria uma barreira de proteção entre a intranet da organização e o restante da Internet. O propósito do *firewall* é proteger os recursos presentes nos computadores de uma organização contra o acesso de usuários ou processos externos e, ao mesmo tempo, controlar o uso dos recursos externos por usuários da própria organização.

Um *firewall* funciona em um *gateway* – um computador posicionado no ponto de entrada da rede intranet de uma organização. O *firewall* recebe e filtra todas as mensagens que entram e saem de uma organização. Ela é configurada, de acordo com a política de segurança da organização, para permitir a passagem de certas mensagens recebidas e enviadas e para rejeitar as demais. Voltaremos a esse assunto na Seção 3.4.8.

Para permitir que os aplicativos distribuídos se movam para além das restrições impostas pelos *firewalls*, há necessidade de produzir um ambiente de rede seguro, no qual uma ampla variedade de aplicativos distribuídos possa ser implantada, com autenticação fim-a-fim, privacidade e segurança. Essa forma mais refinada e flexível de segurança pode ser obtida com o uso de técnicas de criptografia. Normalmente, ela é aplicada em um nível acima do subsistema de comunicação e, assim, não será tratada aqui, mas sim no Capítulo 7. As exceções incluem a necessidade de proteger componentes de rede, como os roteadores, contra interferência não autorizada em sua operação e a necessidade de se ter enlaces seguros para dispositivos móveis e nós externos, para permitir sua participação em uma intranet segura – o conceito de *rede privada virtual* (VPN – *Virtual Private Network*), discutido na Seção 3.4.8.

Mobilidade ◊ Os dispositivos móveis, como computadores laptop, PDAs e telefones móveis com acesso à Internet, mudam freqüentemente de lugar e são reconectados em diferentes pontos da rede, ou mesmo usados enquanto estão em movimento. As redes sem fio (*wireless*) fornecem conectividade para tais dispositivos, mas os esquemas de endereçamento e de roteamento da Internet foram desenvolvidos antes do surgimento desses dispositivos móveis e não são adaptados às suas características de conexão intermitente em diversas sub-redes diferentes. Os mecanismos da Internet foram adaptados e ampliados para suportar mobilidade, mas o crescimento esperado para o uso de dispositivos móveis exigirá ainda mais desenvolvimentos.

Qualidade do serviço ◊ No Capítulo 2, definimos qualidade do serviço como a capacidade de atender prazos finais ao transmitir e processar fluxos de dados multimídia em tempo real. Isso impõe as redes de computadores novos requisitos importantes. Os aplicativos que transmitem dados multimídia exigem largura de banda garantida e latências limitadas para os canais de comunicação que utilizam. Alguns aplicativos variam sua demanda dinamicamente e especificam uma qualidade de serviço mí-

nima aceitável e um ótimo desejado. O aprovisionamento de tais garantias e de sua manutenção será o assunto do Capítulo 17.

Difusão seletiva* (*multicasting*) ◊ A maior parte da comunicação em sistemas distribuídos se dá entre pares de processos, mas freqüentemente também há necessidade de uma comunicação de um para muitos. Embora isso possa ser simulado por múltiplos envios para vários destinos, isto é mais dispendioso do que o necessário e pode não exibir as características de tolerância a falhas exigida pelos aplicativos. Por esses motivos, muitas tecnologias de rede suportam a transmissão de uma mensagem para vários destinatários.

3.2 Tipos de redes

Apresentamos aqui os principais tipos de redes usados pelos sistemas distribuídos: *redes pessoais*, *redes locais*, *redes de longa distância*, *redes metropolitanas* e suas variantes sem fio. As *redes interligadas*, como a Internet, são construídas a partir de redes de todos os tipos. A Figura 3.1 mostra as características de desempenho dos vários tipos de rede discutidos a seguir.

Alguns dos nomes usados para definir os tipos de redes são confusos, pois parecem se referir ao alcance físico (local, longa distância), porém eles também identificam tecnologias de transmissão físicas e protocolos de baixo nível, os quais diferem entre redes locais e longa distância. Entretanto, algumas tecnologias de rede, como o ATM (*Asynchronous Transfer Mode*), são convenientes tanto para aplicativos locais como remotos e algumas redes sem fio também suportam transmissão local e metropolitana.

Denominamos de redes interligadas as redes compostas por várias redes integradas de forma a fornecer um único meio de comunicação de dados. A Internet é o protótipo das redes interligadas; ela é composta de milhões de redes locais, metropolitanas e longa distância. Vamos descrever sua implementação com alguns detalhes, na Seção 3.4.

Redes pessoais (PAN – *Personal Area Networks*) ◊ As PANs representam uma subcategoria das redes locais onde vários dispositivos eletrônicos-digitais transportados pelo usuário são conectados por uma rede de baixo custo e baixa energia. As PANs cabeadas não têm muito interesse, pois poucos usuários desejam ser incomodados com a presença ou necessidade de fios, mas as redes pessoais sem

	Exemplo	*Alcance*	*Largura de banda (Mbps)*	*Latência (ms)*
Redes cabeadas:				
LAN	Ethernet	1–2 kms	10–1000	1–10
WAN	Roteamento IP	mundial	0,010–600	100–500
MAN	ATM	2–50 kms	1–150	10
Interligação em rede	Internet	mundial	0,5–600	100–500
Redes sem fio:				
WPAN	Bluetooth (IEEE 802.15.1)	10–30m	0,5–2	5–20
WLAN	WiFi (IEEE 802.11)	0,15–1,5 km	2–54	5–20
WMAN	WiMAX (IEEE 802.16)	5–50 km	1,5–20	5–20
WWAN	Redes telefônicas GSM, 3G	mundial	0,010–2	100–500

Figura 3.1 Desempenho de rede.

* N. de R.T.: Embora difusão seletiva seja empregada como tradução para *multicast*, por questões de clareza, optamos por manter o termo em inglês.

fio (WPANs – *Wireless Personal Area Networks*) têm cada vez mais importância, devido ao número de dispositivos pessoais, como telefones móveis, PDAs, câmeras digitais, equipamentos sonoros, etc, disponíveis. Descreveremos a WPAN Bluetooth na Seção 3.5.3.

Redes locais (LANs – *Local Area Networks*) ◊ As LANs transportam mensagens em velocidades relativamente altas entre computadores conectados em um único meio de comunicação, como um fio de par trançado, um cabo coaxial ou fibra óptica. Um *segmento* é uma seção de cabo que atende um departamento, ou um piso de um prédio, e que pode ter muitos computadores ligados. Nenhum roteamento de mensagens é necessário dentro de um segmento, pois o meio permite uma comunicação direta entre todos os computadores conectados a ele. A largura de banda total é compartilhada entre os computadores de um segmento. Redes locais maiores, como aquelas que atendem a um campus ou a um prédio de escritórios, são compostas de muitos segmentos interconectados por hubs ou *switches* (veja a Seção 3.3.7). Nas redes locais, a largura de banda total é alta e a latência é baixa, exceto quando o tráfego de mensagens é muito alto.

Várias tecnologias para redes locais foram desenvolvidas nos anos 70 – *Ethernet*, *token rings* e *slotted rings*. Cada uma delas oferece uma solução eficiente e de alto desempenho, porém a Ethernet se tornou a tecnologia dominante para redes locais cabeadas. Ela foi originalmente introduzida no início dos anos 70, com uma largura de banda de 10 Mbps (milhões de bits por segundo) e, mais recentemente, ampliada para versões de 100 Mbps e 1000 Mbps (1 gigabit por segundo). Descreveremos os princípios de operação das redes Ethernet na Seção 3.5.1.

Existe uma base instalada muito grande de redes locais, as quais possuem um ou mais computadores pessoais ou estações de trabalho. Seu desempenho geralmente é adequado para a implementação de sistemas e aplicativos distribuídos. A tecnologia Ethernet não possui as garantias de latência e de largura de banda necessárias para muitos aplicativos multimídia. As redes ATM foram desenvolvidas para preencher essa lacuna, mas seu custo inibe a sua adoção em redes locais. Para superar esses inconvenientes, foram implantadas redes Ethernet comutadas, de altas velocidades, embora não tenham tanta eficiência como as ATM.

Redes de longa distância (WANs – *Wide Area Networks*) ◊ As WANs transportam mensagens em velocidades mais lentas, freqüentemente entre nós que estão em organizações diferentes e que podem estar separadas por grandes distâncias. As WANs podem estar localizadas em diferentes cidades, países ou continentes. O meio de transmissão empregado é formado por um conjunto de circuitos de comunicação interligando um grupo de equipamentos dedicados, chamados *roteadores*. Eles gerenciam a rede de comunicação e direcionam as mensagens, ou pacotes, para seus destinos. Na maioria das redes, as operações de roteamento introduzem um atraso em cada um dos pontos de uma rota; portanto, a latência total da transmissão de uma mensagem depende da rota que ela segue e das cargas de tráfego nos diversos segmentos de rede pelos quais ela passa. Nas redes atuais, essas latências podem ser de até 0,1 a 0,5 segundos. A velocidade de propagação dos sinais eletromagnéticos na maioria das mídias de comunicação é próxima à velocidade da luz e isso estabelece um limite inferior para a latência da transmissão para redes de longa distância. Por exemplo, o atraso da propagação de um sinal para ir da Europa para a Austrália por meio de um enlace terrestre é de aproximadamente 0,13 segundos, e os sinais via satélite geoestacionário entre quaisquer dois pontos na superfície da Terra estão sujeitos a um atraso de aproximadamente 0,20 segundos.

As larguras de banda disponíveis pela Internet também variam bastante. Entre dois pontos da Internet, existem enlaces que disponibilizam velocidades de até 600 Mbps, mas a maior parte das transferências de dados "sentem" uma velocidades de 1 a10 Mbps.

Redes metropolitanas (MANs – *Metropolitan Area Networks*) ◊ Este tipo de rede é baseada em uma infra-estrutura de cabeamento de fibra óptica e cabos de cobre de alta largura de banda, instalados em algumas cidades para transmissão de vídeo, voz e outros dados em distâncias de até 50 quilômetros. Têm sido usada uma variedade de tecnologias para implementar o roteamento de dados nas MANs, variando de Ethernet a ATM.

As conexões DSL (*Digital Subscriber Line* – linha de assinante digital) e de modem a cabo, agora disponíveis em muitos países, são um exemplo do uso de comutadores ATM (Seção 3.5.4). Os comutadores ATM localizam-se nas estações telefônicas e servem para redirecionar os dados digitais que trafegam nos pares de fios de cobre trançados, usados pelas próprias conexões telefônicas, para a casa ou escritório do assinante, em velocidades na faixa de 0,25–8,0 Mbps. O uso de pares de cobre

trançados para as conexões de assinantes DSL limitam a sua abrangência a cerca de 5,5 km a partir do comutador. As conexões de modem a cabo usam as redes de televisão a cabo para atingir velocidades de 1,5 Mbps, com uma abrangência maior do que a DSL.

Redes locais sem fio (WLANs – *Wireless Local Area Networks*) ◊ As WLANs são projetadas para substituir as LANs cabeadas. Seu objetivo é fornecer conectividade para dispositivos móveis, ou simplesmente eliminar a necessidade de uma infra-estrutura com fios e cabos para interconectar computadores dentro de casas e prédios de escritório entre si e a Internet. Seu uso é difundido em diversas variantes do padrão IEEE 802.11 (WiFi), oferecendo larguras de banda entre 10 e 100 Mbps com abrangência de até 1,5 quilômetros. A Seção 3.5.2 fornecerá mais informações sobre seu método de operação.

Redes metropolitanas sem fio (WMANs – *Wireless Metropolitan Area Network*) ◊ O padrão IEEE 802.16 WiMAX é destinado a essa classe de rede. Seu propósito é fornecer uma alternativa às conexões cabeadas para casas ou prédios de escritório e substituir as redes 802.11 WiFi em algumas aplicações.

Redes de longa distância sem fio (WWANs – *Wireless Wide Area Networks*) ◊ A maioria das redes de telefonia móvel é baseada em tecnologias de rede digital sem fio, como o padrão GSM (*Global System for Mobile*), usado na maior parte dos países do mundo. As redes de telefonia móvel são projetadas para operar em áreas amplas (normalmente, países ou continentes inteiros) por meio de conexões de rádio. O sinal de rádio é confinado em regiões denominadas de células, e o alcance da comunicação celular, ou seja, sua cobertura, é garantida pela interligação e superposição dessas células. Com a infra-estrutura da telefonia celular é possível oferecer conexão à Internet para dispositivos móveis. As redes celulares oferecem taxas de transmissão de dados relativamente baixas – 9,6 a 33 Kbps, mas em sua "terceira geração" (3G), oferece taxas de transmissão de dados na faixa de 128–384 Kbps para células de raio de poucos quilômetros e até 2Mbps para células menores. Os leitores que estiverem interessados em estudar mais a fundo (do que podemos aqui) as tecnologias de redes móveis e sem fio, podem consultar o excelente manual de Stojmenovic [2002].

Inter-redes ◊ Uma rede inter-rede, ou redes interligadas, é um subsistema de comunicação onde várias redes são unidas para fornecer recursos de comunicação de dados comuns, abstraindo as tecnologias e os protocolos das redes componentes individuais e os métodos usados para sua interconexão.

As redes interligadas são necessárias para o desenvolvimento de sistemas distribuídos abertos e extensíveis. A característica aberta dos sistemas distribuídos sugere que as redes neles usadas devem ser extensíveis para um número muito grande de computadores, enquanto as redes individuais têm espaços de endereço restritos e algumas possuem limitações no desempenho, sendo incompatíveis para uso em larga escala. Uma variedade de tecnologias de rede local e de longa distância podem ser integradas para fornecer a capacidade de interligação em rede necessária para um grupo de usuários. Assim, as redes interligadas trazem muitas das vantagens dos sistemas abertos para o aprovisionamento de comunicação em sistemas distribuídos.

As redes interligadas são construídas a partir de uma variedade de redes componentes. Elas são interconectadas por equipamentos dedicados de comutação, os *roteadores*, e por computadores de propósito geral, os *gateways*. O subsistema de comunicação resultante é dado por uma camada de software que suporta o endereçamento e a transmissão de dados para todos os computadores das redes interligadas. O resultado pode ser considerado como uma "rede virtual" construída pela sobreposição de uma camada de integração de redes através de um meio de comunicação composto por redes individuais, roteadores e *gateways* subjacentes. A Internet é o maior exemplo de interligação em rede e seus protocolos TCP/IP são um exemplo da camada de integração mencionada anteriormente.

Erros em comunicação em rede ◊ Um ponto adicional de comparação, não mencionado na Figura 3.1, é a freqüência e os tipos de falha que podem ser esperadas nos diferentes tipos de rede. A confiabilidade da mídia de transmissão de dados subjacente é muito alta em todos os tipos, exceto nas redes sem fio, onde pacotes são freqüentemente perdidos devido à interferência externa. Mas os pacotes podem ser perdidos em todos os tipos de rede, devido aos atrasos de processamento e ao estouro de buffer nos comutadores e nos nós de destino, e essa é a causa mais comum de perda de pacotes.

Os pacotes também podem ser entregues em uma ordem diferente daquela em que foram emitidos. Isso acontece apenas em redes onde os pacotes são direcionados individualmente – principal-

mente em redes de longa distância. Outro erro comum é a entrega de cópias duplicadas dos pacotes; normalmente, isso é uma conseqüência da suposição pelo remetente de que um pacote foi perdido. Nesse caso, o pacote é retransmitido e, então, o original e a cópia retransmitida aparecem no destino.

3.3 Conceitos básicos de redes

A base de todas as redes de computadores é a técnica de comutação de pacotes, desenvolvida pela primeira vez nos anos 60. Essa técnica permite que pacotes de dados endereçados a diferentes destinos compartilhem um único enlace de comunicação, ao contrário do que ocorre com a tecnologia de comutação de circuitos, que fundamenta a telefonia convencional. Os pacotes são enfileirados em um buffer e transmitidos quando o enlace está disponível. A comunicação é assíncrona – as mensagens chegam ao seu destino após um atraso que varia de acordo com o tempo que os pacotes levam para trafegar pela rede.

3.3.1 Transmissão de pacotes

Na maioria das aplicações de redes de computadores o requisito fundamental é a transmissão de unidades lógicas de informação ou *mensagens* – seqüências de dados de comprimento arbitrário. Mas, antes que uma mensagem seja transmitida, ela é subdividida em *pacotes*. A forma mais simples de pacote é uma seqüência de dados binários (uma seqüência de bits ou bytes) de comprimento limitado, junto com informações de endereçamento suficientes para identificar os computadores de origem e de destino. Pacotes de comprimento limitado são usados:

- para que cada computador da rede possa alocar armazenamento em buffer suficiente para armazenar o maior pacote possível a ser recebido;
- para eliminar os atrasos excessivos que ocorreriam na espera da liberação de canais de comunicação, caso as mensagens longas fossem transmitidas sem nenhum tipo de subdivisão.

3.3.2 Fluxo de dados

Já mencionamos, no Capítulo 2, que os aplicativos multimídia necessitam contar com a transmissão de fluxos de dados de áudio e de vídeo em velocidades garantidas e com latências limitadas. Tais fluxos, ou *streams*, diferem substancialmente do tipo de tráfego baseado em mensagens para o qual a transmissão de pacotes foi projetada. O fluxo de áudio e vídeo exige larguras de banda muito superiores do que a maioria das outras formas de comunicação em sistemas distribuídos.

A transmissão de um fluxo de vídeo para exibição em tempo real exige uma largura de banda de cerca de 1,5 Mbps, se os dados forem compactados, ou 120 Mbps, caso contrário. Além disso, o fluxo de dados é contínuo, em oposição ao tráfego intermitente gerado pelas interações cliente-servidor típicas. O *tempo de reprodução* de um elemento multimídia é o tempo no qual ele deve ser exibido (para um elemento de vídeo) ou convertido em som (para uma amostra de som). Por exemplo, em um fluxo de quadros de vídeo com uma velocidade de projeção de 24 quadros por segundo, o quadro N tem um tempo de reprodução de $N/24$ segundos, após o tempo de início do fluxo. Os elementos que chegarem ao seu destino depois de seu tempo de reprodução não são mais úteis e serão eliminados pelo processo receptor.

A distribuição adequada de tais fluxos de dados depende da disponibilidade de conexões com qualidade de serviço garantida – largura de banda, latência e confiabilidade. Isso pode ser obtido com o estabelecimento de um canal de comunicação de um fluxo multimídia da origem para o destino, com uma rota predefinida pela rede, com um conjunto de recursos reservados em cada nó pelo qual ele passará e, onde for apropriado, com o uso de buffer, para atenuar as irregularidades no fluxo de dados através do canal de comunicação. Os dados podem então passar pelo canal, do remetente para o destinatário, na velocidade exigida.

As redes ATM (Seção 3.5.4) são especificamente projetadas para fornecer alta largura de banda com baixa latência e para suportar qualidade de serviço por meio da reserva de recursos de rede. O

IPv6, o novo protocolo de rede da Internet, descrito em linhas gerais na Seção 3.4.4, inclui recursos que permitem que cada um dos pacotes IP* de um fluxo em tempo real sejam identificados e tratados separadamente dos demais.

Os subsistemas de comunicação que fornecem garantia de qualidade do serviço exigem meios para a alocação prévia de recursos de rede e o cumprimento dessas alocações. O protocolo RSVP (*Resource Reservation Protocol*) [Zhang *et al.* 1993] permite que os aplicativos negociem a alocação prévia de largura de banda para fluxos de dados em tempo real. O protocolo RTP (*Real Time Transport Protocol*) [Schulzrinne *et al.* 1996] é um protocolo de transferência de dados, em nível de aplicação, que inclui em cada pacote os detalhes do tempo de reprodução e outros requisitos de sincronismo. A disponibilidade de implementações eficazes desses protocolos na Internet dependerá de alterações substanciais nas camadas de transporte e de rede. O Capítulo 17 discutirá em detalhes as necessidades dos aplicativos multimídia distribuídos.

3.3.3 Esquemas de comutação

Uma rede consiste em um conjunto de nós conectados por circuitos. Para transmitir informações entre dois nós arbitrários, é exigido um sistema de comutação. Definimos aqui os quatro tipos de comutação usados na interligação de redes de computadores.

Difusão (*broadcast*) ◊ O *broadcast* é uma técnica de transmissão que não envolve nenhuma comutação. Tudo é transmitido para cada nó e fica por conta dos receptores recuperar as transmissões a eles endereçadas. Algumas tecnologias de rede local, incluindo a Ethernet, são baseadas em *broadcast*. A interligação em rede sem fio é necessariamente baseada em *broadcast*, mas na ausência de circuitos fixos, as difusões são organizadas de modo a atingir nós agrupados em células.

Comutação de circuitos ◊ Houve um tempo em que as redes telefônicas eram as únicas redes de telecomunicações. Sua operação era simples de entender: quando o chamador discava um número, o par de fios de seu telefone, que ia até a estação local, era conectado por um comutador automático dessa central ao par de fios conectados ao telefone da outra pessoa. Para uma chamada interurbana, o processo era semelhante, mas a conexão seria comutada até seu destino por meio de diversas estações intermediárias. Às vezes, esse sistema é referenciado como sistema telefônico antigo (POTS – *Plain Old Telephone System*). Trata-se de uma *rede de comutação de circuitos* típica.

Comutação de pacotes ◊ O advento dos computadores e da tecnologia digital trouxe muitas possibilidades para as telecomunicações como as capacidades básicas de processamento e armazenamento. Isso possibilitou a construção de redes de comunicação de uma maneira bastante diferente. Esse novo tipo de rede de comunicação é chamada de *rede de armazenamento e encaminhamento* (*store-and-forward*). Em vez de estabelecer e desfazer conexões para construir circuitos, uma rede de armazenamento e encaminhamento apenas encaminha pacotes de sua origem para seu destino. Existe um computador em cada nó de comutação (onde vários circuitos precisam ser interconectados). Cada pacote que chega em um nó é primeiramente armazenado em sua memória e, depois, processado por um programa que o transmite a um circuito de saída, que transferirá o pacote para outro nó mais próximo de seu destino final.

Não há nada de realmente novo nessa idéia: o sistema postal é uma rede de armazenamento e encaminhamento de cartas, com o processamento feito por seres humanos ou automaticamente nos centros de triagem. Mas em uma rede de computadores, os pacotes podem ser armazenados e processados rápido o suficiente para dar a ilusão de uma transmissão instantânea, mesmo que o pacote tenha de ser direcionado por meio de muitos nós.

Frame relay ◊ Na realidade, leva desde algumas dezenas de microssegundos a alguns milissegundos para, em uma rede de armazenamento e encaminhamento, comutar um pacote em cada nó. Esse

* N. de R.T.: Pacote é um termo genérico para referenciar uma seqüência de dados binários com tamanho limitado usado como unidade de transmissão. Entretanto, conceitualmente, cada camada de um protocolo de rede define sua própria unidade de transmissão, denominada de PDUs (*Protocol Data Unit*), as quais possuem nomes específicos no TCP/IP: datagrama para o IPv4, pacote para o IPv6, e datagrama e segmento para o UDP e o TCP, respectivamente.

atraso de comutação depende do tamanho do pacote, da velocidade do hardware e da quantidade do tráfego, mas seu limite inferior é determinado pela largura de banda da rede, pois o pacote inteiro deve ser recebido antes que possa ser encaminhado para outro nó. Grande parte da Internet é baseada na comutação de pacotes e, conforme já vimos, na Internet, mesmo os pacotes pequenos normalmente demoram até 200 milissegundos para chegar aos seus destinos. Atrasos dessa magnitude são longos demais para aplicações em tempo real, como telefonia e videoconferência, onde atrasos de menos de 50 milissegundos são necessários para manter uma conversação em alta qualidade.

O método de comutação *frame relay* traz algumas das vantagens da comutação de circuitos para as redes de comutação de pacotes. O problema de atraso é superado através do emprego de pequenos pacotes, denominados de quadros (*frames*), que são comutados dinamicamente. Os nós de comutação (que normalmente são processadores de propósito específico) redirecionam os quadros baseados apenas no exame de seus primeiros bits; sem armazená-los como um todo. As redes ATM, Seção 3.5.4, são o melhor exemplo do emprego dessa técnica. As redes ATM de alta velocidade podem transmitir pacotes por redes compostas de muitos nós em poucas dezenas de microssegundos.

3.3.4 Protocolos

O termo *protocolo* é usado para designar um conjunto bem conhecido de regras e formatos a serem usados na comunicação entre processos a fim de realizar uma determinada tarefa. A definição de protocolo tem duas partes importantes:

- uma especificação da seqüência de mensagens que devem ser trocadas;
- uma especificação do formato dos dados nas mensagens.

A existência de protocolos bem conhecidos permite que os vários componentes de software de um sistema distribuído sejam desenvolvidos e implementados de forma independente, usando diferentes linguagens de programação, em computadores que podem usar distintas representações internas para códigos e dados.

Um protocolo é implementado por meio de dois módulos de software localizados nos computadores origem e destino. Por exemplo, um *protocolo de transporte* transmite mensagens de qualquer comprimento entre um processo remetente e um processo destino. Um processo que queira transmitir uma mensagem para outro emite uma chamada para o módulo de protocolo de transporte, passando a ele uma mensagem em um formato especificado. O software de transporte se ocupa então com o envio da mensagem para o destino, subdividindo-a em pacotes de tamanho e formato especificados, que podem ser transmitidos para o destino por meio do *protocolo de rede* – um outro protocolo de nível mais inferior. No computador destino, o módulo de protocolo de transporte correspondente recebe o pacote por meio do módulo de protocolo em nível de rede e realiza transformações inversas para recompor a mensagem, antes de passá-la para um processo destino.

Camadas de protocolo ◊ O software de rede é organizado em uma hierarquia de camadas ou níveis. Cada camada apresenta uma interface bem definida para as camadas que estão acima dela e implementa novos serviços sobre as funcionalidades do sistema de comunicação subjacente. Uma camada é representada por um módulo de software em cada computador conectado à rede. A Figura 3.2 ilustra a estrutura e o fluxo de dados que ocorre quando uma mensagem é transmitida usando um protocolo em camadas. Logicamente, cada módulo de software de um nível de um computador se comunica diretamente com seu correspondente no outro computador, mas, na realidade, os dados não são transmitidos diretamente entre os módulos de protocolo de cada nível. Em vez disso, cada camada de software de rede se comunica com as camadas acima e abaixo dela por intermédio de chamadas de procedimentos locais.

No lado remetente, cada camada (com exceção a superior, ou camada de aplicação) aceita dados da camada que está acima dela em um formato especificado e aplica transformações para encapsular esses dados em um formato próprio, específico a si mesma, antes de repassá-los para a camada abaixo dela. A Figura 3.3 ilustra esse procedimento aplicado às quatro camadas superiores do conjunto de protocolos OSI. A figura mostra a inclusão de cabeçalhos que contêm informações de controle de cada uma das camadas. Conceitualmente, é possível que as camadas também insiram informações de controle na parte posterior de cada pacote, porém, por questões de clareza, isso foi omitido na Figura

Figura 3.2 Organização conceitual de protocolos em camadas.

3.3; ela também presume que a mensagem da camada de aplicação a ser transmitida é menor do que o tamanho de pacote máximo da rede subjacente. Se não fosse, então ela teria que ser segmentada e encapsulada em vários pacotes. No lado destino, os dados recebidos por uma camada sofrem as transformações inversas antes de serem repassados para a camada superior. O tipo de protocolo da camada superior é incluído no cabeçalho de cada camada para permitir que a pilha de protocolos no destino selecione os componentes de software corretos para desempacotar os pacotes.

Cada camada fornece serviços para a camada acima dela. A camada mais inferior é a *camada física* que oferece o serviço de transmissão física dos dados. Isso é implementado por um meio de comunicação (cabos de cobre ou fibra óptica, canais de comunicação via satélite ou transmissão de rádio) e por circuitos de sinalização analógicos que inserem sinais eletromagnéticos no meio de comunicação no nó de envio e os capta no nó de recepção. Nos nós de recepção, os dados são recebidos e repassados para as camadas superiores da hierarquia de protocolos. Cada camada realiza transformações nos dados até estarem em uma forma que possa ser entregue para o processo destinatário pretendido.

Conjuntos de protocolo ◊ Um conjunto completo de camadas de protocolo é referido como *conjunto de protocolos* ou *pilha de protocolos*, refletindo a estrutura em camadas. A Figura 3.4 mostra uma pilha de protocolos que obedece ao modelo de referência de sete camadas do padrão *OSI (Open System Interconnection)*, adotado pela ISO (*International Organization for Standardization*) [ISO 1992]. O modelo de referência OSI foi adotado para estimular o desenvolvimento de padrões de protocolo que atendessem os requisitos dos sistemas abertos.

O objetivo de cada camada no modelo de referência OSI está resumido na Figura 3.5. Conforme seu nome implica, ele é uma estrutura para a definição de protocolos e não uma definição de um conjunto de protocolos específico. Os conjuntos de protocolo que obedecem ao modelo OSI devem incluir pelo menos um protocolo específico em cada um dos sete níveis, ou camadas, definidos pelo modelo.

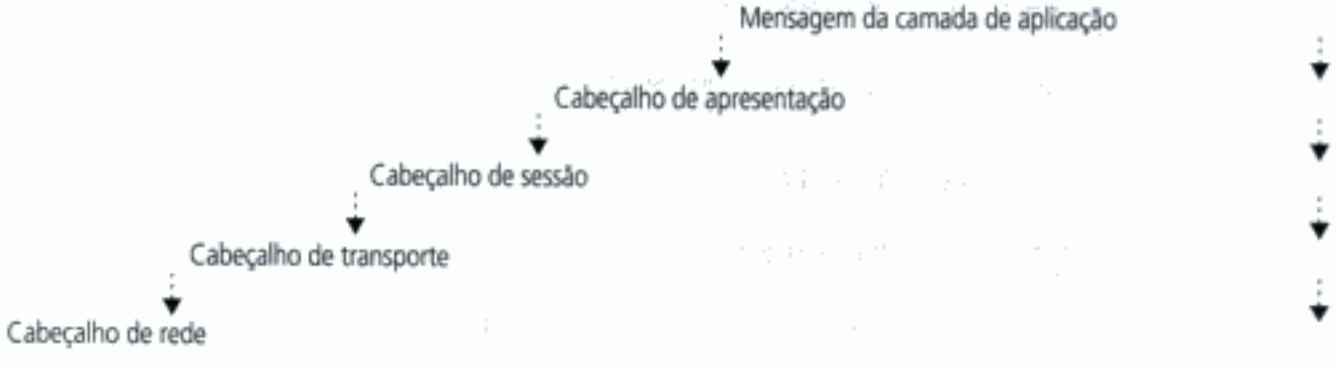


Figura 3.3 Encapsulamento em protocolos dispostos em camadas.

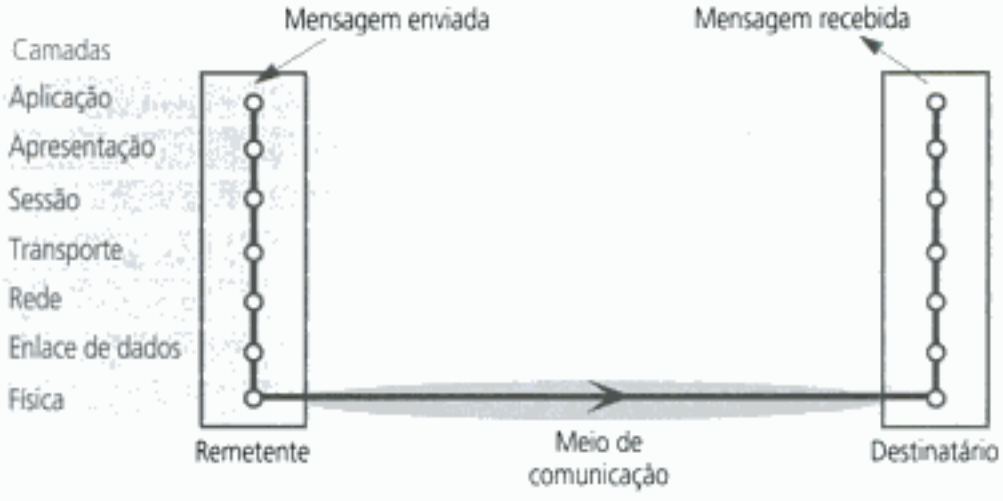


Figura 3.4 Camadas de protocolo no modelo de protocolo *Open System Interconnection (OSI)* da ISO.

A disposição de protocolos em camadas apresenta vantagens substanciais na simplificação e generalização das interfaces de software para acesso aos serviços de comunicação das redes, mas também acarreta custos de desempenho significativos. A transmissão de uma mensagem em nível de aplicação, por meio de uma pilha de protocolos com *N* camadas, envolve *N* transferências de controle

Camada	*Descrição*	*Exemplos*
Aplicativo	Protocolos projetados para atender os requisitos de comunicação de aplicativos específicos, freqüentemente definindo uma interface para um serviço.	HTTP, FTP, SMTP, CORBA IIOP
Apresentação	Os protocolos deste nível transmitem dados em uma representação de rede independente das usadas em cada computador, as quais podem diferir. A criptografia, se exigida, também é feita nesta camada.	Segurança TLS, CORBA Data Rep.
Sessão	Neste nível são realizadas a confiabilidade e a adaptação, como a detecção de falhas e a recuperação automática.	SIP
Transporte	Este é o nível mais baixo em que mensagens (em vez de pacotes) são manipuladas. As mensagens são endereçadas para portas de comunicação associadas aos processos. Os protocolos desta camada podem ser orientados com ou sem conexão.	TCP, UDP
Rede	Transfere pacotes de dados entre computadores em uma rede específica. Em uma rede de longa distância, ou em redes interligadas, isso envolve a geração de uma rota passando por roteadores. Em uma única rede local, nenhum roteamento é exigido.	IP, circuitos virtuais ATM
Enlace de dados	Responsável pela transmissão de pacotes entre nós que estão diretamente conectados por um enlace físico. Em uma rede de longa distância, a transmissão é feita entre pares de roteadores ou entre roteadores e *hosts*. Em uma rede local, ela é feita entre qualquer par de *hosts*.	MAC Ethernet, transferência de célula ATM, PPP
Física	São os circuitos e o hardware que materializam a rede. Essa camada transmite seqüências de dados binários através de sinalização analógica, usando modulação em amplitude ou em freqüência de sinais elétricos (em circuitos a cabo), sinais luminosos (em circuitos de fibra óptica) ou outros sinais eletromagnéticos (em circuitos de rádio e microondas).	Sinalização de banda-base Ethernet, ISDN

Figura 3.5 Exemplos de protocolos OSI.

entre as camadas de software, sendo uma delas a entrada do sistema operacional e, como parte dos mecanismos de encapsulamento, são necessários *N* cópias dos dados. Todas essas sobrecargas resultam em taxas de transferência de dados entre processos aplicativos muito mais lentas do que a largura de banda disponível da rede.

A Figura 3.5 inclui exemplos de protocolos usados na Internet, mas sua implementação não segue o modelo OSI em dois aspectos. Primeiro, as camadas de aplicação, apresentação e sessão não são claramente distinguidas na pilha de protocolos Internet. Em vez disso, as camadas de aplicação e apresentação são implementadas como uma única camada de *middleware* ou, separadamente, dentro de cada aplicativo. Assim, o CORBA implementa invocações entre objetos e representação de dados em uma biblioteca de *middleware* que é incluída em cada processo aplicativo (veja o Capítulo 20 para obter mais detalhes sobre o CORBA). De forma similar, os navegadores web e outros aplicativos que exigem canais seguros empregam a *Secure Sockets Layer* (Capítulo 7) como biblioteca de funções.

Segundo, a camada de sessão é integrada com a camada de transporte. Os conjuntos de protocolos de redes interligadas incluem uma camada de aplicação, uma camada de transporte e uma *camada inter-rede*. A camada inter-rede implementa um rede virtual, responsável por transmitir pacotes das redes interligadas para um computador de destino. Um *pacote inter-rede* é a unidade de dados transmitidos entre as redes interligadas.

Os protocolos de redes interligadas são sobrepostos em redes subjacentes, conforme ilustrado na Figura 3.6. A camada de enlace aceita pacotes inter-rede e os converte adequadamente em pacotes de transmissão de cada rede subjacente.

Montagem de pacotes ◊ A tarefa de dividir mensagens em pacotes antes da transmissão e sua remontagem no computador destino normalmente é executada na camada de transporte.

Os pacotes do protocolo da camada de rede consistem em um *cabeçalho* e em um *campo de dados*. Na maioria das tecnologias de rede, o campo de dados tem comprimento variável, com o comprimento máximo chamado de *unidade de transferência máxima* (MTU – *Maximum Transfer Unit*). Se o comprimento de uma mensagem ultrapassar o MTU da camada de rede subjacente, ela deve ser fragmentada em porções de tamanho apropriado, identificada com números de seqüência para ser remontada, e transmitida em vários pacotes. Por exemplo, o MTU da Ethernet é de 1500 bytes – não mais do que esse volume de dados pode ser transmitido em um único pacote Ethernet.

Embora o protocolo IP (*Internet Protocol*) seja um protocolo de camada de rede, seu MTU é extraordinariamente grande, 64 Kbytes, (na prática, são usados freqüentemente 8 Kbytes, pois alguns nós não conseguem manipular pacotes grandes assim). Qualquer que seja o valor do MTU adotado pelo IP, pacotes maiores do que o MTU Ethernet devem ser fragmentados para transmissão em redes Ethernet. Os pacotes IP são denominados de datagramas.

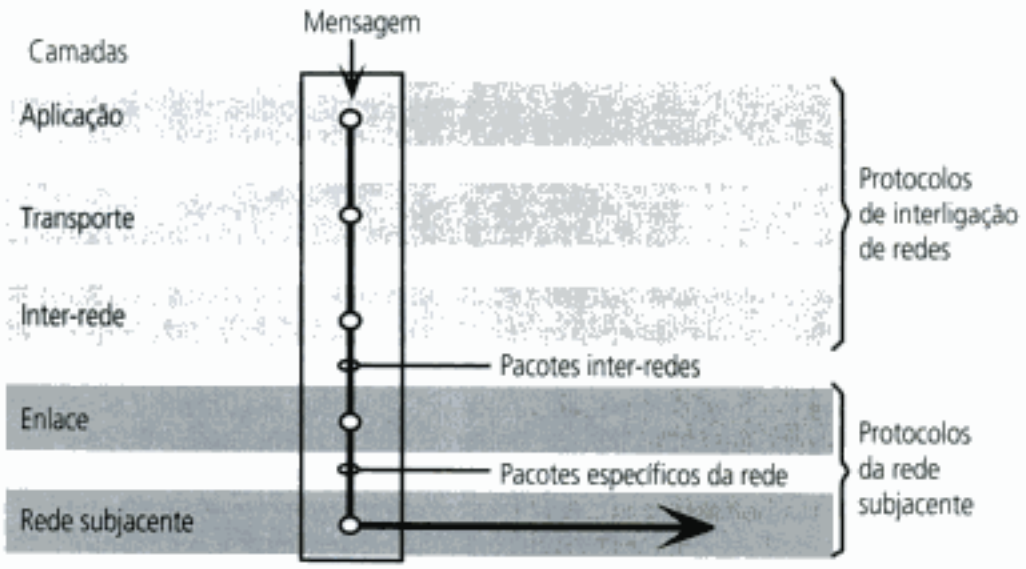


Figura 3.6 Camadas de interligação de redes.

Portas ◊ A tarefa da camada de transporte é fornecer um serviço de transporte de mensagens entre pares de *portas* independentemente do tipo de rede realmente empregado. As portas são pontos de destino definidos pelo software em um computador *host*. Elas são ligadas a processos, permitindo que a transmissão de dados seja endereçada para um processo específico em um nó de destino. Aqui, discutimos o endereçamento via portas de acordo com sua implementação na Internet e na maioria das redes. O Capítulo 4 descreverá sua programação.

Endereçamento ◊ A camada de transporte é responsável pela entrega de mensagens para seus destinos baseados em *endereços de transporte* que são compostos pelo *endereço de rede* de um computador *host* e um *número de porta*. Um endereço de rede é um identificador numérico que identifica exclusivamente um computador *host* e permite sua localização pelos nós responsáveis pelo roteamento. Na Internet, cada computador *host* recebe um número IP, o qual o identifica e a sub-rede à qual ele está conectado, permitindo que os dados sejam direcionados a ele a partir de qualquer outro nó, conforme descrito nas seções a seguir. Em uma rede local Ethernet não existe nós de roteamento; cada *host* é responsável por reconhecer e selecionar os pacotes endereçados a ele.

Os serviços Internet bem conhecidos, como HTTP ou FTP, receberam *números de porta de contato* que são registrados junto a uma autoridade central, a IANA (*Internet Assigned Numbers Authority*) [www.iana.org]. Para acessar um serviço em um determinado *host*, o pedido é enviado para a porta associada a esse serviço, nesse *host*. Alguns serviços, como o FTP (porta 21), em sua execução, alocam uma nova porta (com um número privado) e enviam o número dessa nova porta para o cliente. O cliente usa essa nova porta para efetuar o restante de uma transação ou de uma sessão. Outros serviços, como o HTTP (porta 80), fazem todas as suas transações diretamente pela porta de contato.

Os números de porta abaixo de 1023 são definidos como *portas bem conhecidas*, cujo uso é restrito a processos privilegiados na maioria dos sistemas operacionais. As portas entre 1024 e 49151 são *portas registradas*, para as quais a IANA mantém descrições de serviço, e as portas restantes até 65535 estão disponíveis para propósitos gerais. Na prática, todas as portas acima de 1023 podem ser usadas para propósitos gerais, mas os computadores que as utilizam para isso não podem acessar simultaneamente os serviços registrados correspondentes.

A alocação de um número de porta fixo não é apropriada para o desenvolvimento de sistemas distribuídos, os quais freqüentemente abrangem uma multiplicidade de servidores, incluindo aqueles alocados dinamicamente. As soluções para esse problema envolvem a alocação dinâmica de portas para serviços e o aprovisionamento de mecanismos de vinculação para permitir que os clientes localizem serviços e suas portas usando nomes simbólicos. Algumas dessas técnicas serão discutidas mais a fundo no Capítulo 5.

Entrega de pacotes ◊ Existem duas estratégias para a entrega de pacotes pela camada de rede:

Redes baseadas em datagramas: o termo datagrama se refere à semelhança desse modo de entrega de dados com a maneira pela qual cartas e telegramas são distribuídos. A característica essencial das redes de datagramas é que a distribuição de cada pacote é um procedimento independente; nenhuma configuração é exigida e uma vez que o pacote é entregue, a rede não mantém mais nenhuma informação sobre ele. Em uma rede de datagramas, uma seqüência de pacotes transmitida por um *host* para um único destino pode seguir rotas diferentes (se, por exemplo, a rede for capaz de se adaptar às falhas de manipulação ou de reduzir os efeitos de congestionamentos localizados) e quando isso ocorre, eles podem chegar fora da seqüência em que foram emitidos.

Todo datagrama contém o endereço de rede dos *hosts* de origem e destino; este último é um parâmetro essencial para o processo de roteamento, conforme descreveremos na próxima seção. As redes baseadas em datagramas é o conceito sobre o qual as redes de pacotes foram originalmente baseadas e que é encontrado na maioria das redes de computadores atuais. A camada de rede da Internet – IP –, a Ethernet e um grande número das tecnologias de rede local com e sem fio são baseadas em datagramas.

Redes baseadas em circuito virtual: alguns serviços da camada de rede implementam o envio de pacotes de maneira análoga a uma rede telefônica. Um circuito virtual deve ser configurado antes que os pacotes possam passar de um *host* de origem A para um *host* de destino B. O estabelecimento de um circuito virtual envolve a definição de uma rota entre a origem e o destino, possivel-

mente passando por vários nós intermediários. Em cada nó ao longo da rota, é criada uma entrada em uma tabela, indicando qual enlace deve ser usado para atingir a próxima etapa da rota.

Uma vez configurado o circuito virtual, ele pode ser usado para transmitir qualquer quantidade de pacotes. Cada pacote da camada de rede contém apenas o número de circuito virtual no lugar dos endereços de origem e de destino. Os endereços não são necessários, pois os pacotes são direcionados nos nós intermediários pela referência ao número do circuito virtual. Quando um pacote chega ao seu destino, a origem pode ser determinada a partir do número do circuito virtual.

A analogia com as redes telefônicas não deve ser tomada literalmente. No sistema telefônico antigo, uma chamada telefônica resultava no estabelecimento de um circuito físico entre os correspondentes, e os enlaces de voz a partir dos quais ele era construído ficavam reservados para seu uso exclusivo. Na distribuição de pacotes por circuito virtual, os circuitos são representados apenas por entradas de tabela nos nós de roteamento, e os enlaces em que os pacotes são redirecionados são ocupados apenas pelo tempo de transmissão do pacote; no restante do tempo, eles estão liberados. Portanto, um único enlace pode ser empregado em muitos circuitos virtuais separados. A tecnologia de rede de circuito virtual mais importante em uso atualmente é a ATM; já mencionamos (na Seção 3.3.3) que ela tira proveito de latências menores para a transmissão de pacotes individuais; isso é um resultado direto do uso de circuitos virtuais. Contudo, a necessidade de uma fase de configuração resulta em um pequeno atraso, antes que os pacotes possam ser enviados para um novo destino.

A distinção entre a distribuição de pacotes baseados em redes de datagramas e de circuito virtual na camada de rede não deve ser confundida com dois mecanismos de nome semelhante na camada de transporte – transmissão orientada a conexão e não orientada a conexão. Vamos descrevê-los na Seção 3.4.6, no contexto dos protocolos de transporte da Internet, UDP (não orientado a conexão) e TCP (orientado a conexão). Aqui, simplesmente mencionamos que cada um desses modos da camada de transporte pode ser implementado sobre qualquer tipo de rede.

3.3.5 Roteamento

O roteamento é uma função necessária sempre que se têm a interligação de redes permitindo a comunicação entre seus *hosts*. Note que em redes locais, como a Ethernet, o roteamento é desnecessário já que os *hosts* têm a capacidade de se comunicar diretamente. Em redes de grande dimensão é empregado o *roteamento adaptativo*: a melhor rota de comunicação entre dois pontos é periodicamente reavaliada, levando em conta o tráfego corrente na rede e as falhas, como conexões desfeitas ou nós de roteamento danificados.

A entrega de pacotes aos seus destinos, em uma inter-rede como a ilustrada na Figura 3.7, é de responsabilidade coletiva dos nós de roteamento localizados nos pontos de interconexão. A não ser que os *hosts* de origem e destino estejam na mesma rede local, o pacote precisa ser transmitido em uma série de etapas ou passos (*hops*), passando por vários nós de roteamento intermediários. A determinação das rotas para a transmissão de pacotes para seus destinos é de responsabilidade de um *algoritmo de roteamento* – implementado na camada de rede em cada nó.

Um algoritmo de roteamento tem duas partes:

1. Tomar decisões para determinar a rota que cada pacote deve seguir ao passar pela rede. Em camadas de rede com comutação de circuitos, como o X.25, e em redes *frame relay*, como o ATM, a rota é determinada quando o circuito virtual ou uma conexão é estabelecida. Nas redes baseadas em comutação de pacotes, como o IP, a rota é determinada individualmente para cada pacote e o algoritmo deve ser particularmente simples e eficiente para não degradar o desempenho da rede.
2. Atualizar dinamicamente o seu conhecimento da topologia da rede com base no monitoramento do tráfego e na detecção de alterações de configuração ou falhas. Essa atividade é menos crítica quanto ao tempo; podendo ser usadas técnicas mais lentas e que utilizam mais poder de computação.

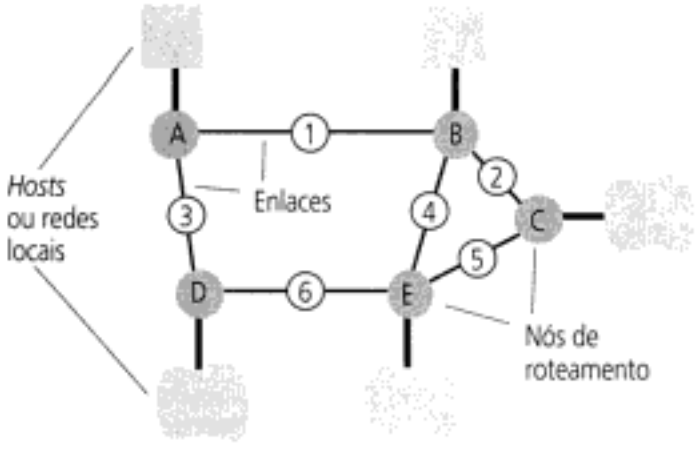


Figura 3.7 Roteamento em redes de longa distância.

Essas duas atividades são distribuídas em toda a rede. As decisões de roteamento são tomadas em cada nó de roteamento, usando informações mantidas localmente, para determinar o próximo *hop* que um pacote deve ser reencaminhado. As informações de roteamento mantidas localmente são atualizadas periodicamente por um algoritmo que distribui informações sobre os estados dos enlaces (suas cargas e status de falha).

Um algoritmo simples de roteamento ◊ O algoritmo que descrevemos aqui é o algoritmo de vetor de distância. Isso fornecerá a base para a discussão da Seção 3.4.3, sobre o algoritmo de estado de enlace, que é usado desde 1979 como o principal algoritmo de roteamento na Internet. O roteamento é um exemplo do problema de determinação de caminhos a serem percorridos em grafos. O algoritmo do menor caminho de Bellman, publicado bem antes que as redes de computadores fossem desenvolvidas [Bellman 1957], fornece a base do método do vetor de distância. O método de Bellman foi transformado por Ford e Fulkerson [1962] em um algoritmo distribuído, conveniente para implementação em redes de grande dimensão, e os protocolos baseados no trabalho deles são freqüentemente referidos como protocolos Bellman–Ford.

A Figura 3.8 mostra as tabelas de roteamento que seriam mantidas em cada um dos nós de roteamento da rede da Figura 3.7, pressupondo que não exista nenhum enlace ou nó defeituoso. Cada linha da tabela fornece as informações de roteamento dos pacotes endereçados a um certo destino. O campo *Saída* especifica o enlace a ser utilizado para os pacotes alcançarem um determinado destino. O campo *Custo* é uma métrica a ser usada no cálculo do vetor de distância, nesse caso, corresponde ao número de nós intermediários (*hops*) até um dado destino. Essa métrica é comum em redes que possuem enlaces de largura de banda semelhante. As informações de custo armazenadas nas tabelas de roteamento não são usadas durante o roteamento de pacotes executadas pela parte 1 do algoritmo de roteamento, mas são exigidas para as ações de construção e manutenção da tabela de roteamento, pela parte 2.

As tabelas de roteamento contêm uma única entrada para cada destino possível, mostrando o próximo passo (*hop*) que um pacote deve dar em direção ao seu destino. Quando um pacote chega a um nó de roteamento, o endereço de destino é extraído e pesquisado na tabela de roteamento local. A entrada na tabela correspondente ao endereço de destino identifica o enlace de saída que deve ser usado para encaminhar o pacote para diante, em direção ao seu destino. O enlace de saída corresponde a uma interface de rede do nó.

Por exemplo, quando um pacote endereçado para C é recebido pelo nó A, o roteamento feito por este nó examina a entrada corresponde a C em sua tabela de roteamento. Ela mostra que o pacote deve ser direcionado para fora de A pela interface de saída (enlace) rotulada como 1. O pacote chega ao nó B e o mesmo procedimento é realizado. A tabela de roteamento do nó B mostra que a rota na direção de C é por meio da interface de saída rotulada como 2. Finalmente, quando o pacote chega ao nó C, sua entrada da tabela de roteamento mostra "local", em vez de uma interface de saída. Isso indica que o pacote deve ser entregue para um *host* local a rede associada ao nó C.

	Nó A			*Nó B*			*Nó C*	
Para	*Saída*	*Custo*	*Para*	*Saída*	*Custo*	*Para*	*Saída*	*Custo*
A	local	0	A	1	1	A	2	2
B	1	1	B	local	0	B	2	1
C	1	2	C	2	1	C	local	0
D	3	1	D	1	2	D	5	2
E	1	2	E	4	1	E	5	1

	Nó D			*Nó E*	
Para	*Saída*	*Custo*	*Para*	*Saída*	*Custo*
A	3	1	A	4	2
B	3	2	B	4	1
C	6	2	C	5	1
D	local	0	D	6	1
E	6	1	E	local	0

Figura 3.8 Tabelas de roteamento para a rede da Figura 3.7.

Agora, vamos considerar como as tabelas de roteamento são construídas e mantidas quando ocorrem falhas na rede, ou seja, como é executada a parte 2 do algoritmo de roteamento descrito anteriormente? Como cada tabela de roteamento especifica apenas um único *hop* para cada rota, a construção ou atualização das informações de roteamento pode ocorrer de maneira distribuída. Um nó de roteamento troca informações sobre a rede com os nós de roteamento vizinhos enviando um resumo de sua tabela de roteamento, usando um *protocolo de informações de roteamento* (RIP – *Routing Information Protocol*). As ações do RIP feitas por um nó de roteamento são descritas, informalmente, a seguir:

1. *Periodicamente e quando a tabela de roteamento local muda*, um nó de roteamento envia sua tabela (em forma resumida) para todos seus vizinhos imediatos que estão acessíveis. Isto é, envia através de cada interface de saída (enlace) não defeituosa um pacote RIP contendo uma cópia de sua tabela de roteamento.
2. *Quando uma tabela é recebida de um vizinho imediato*, o nó verifica se a tabela recebida possui uma rota para um novo destino ou uma rota melhor (de custo mais baixo) para um destino já existente. Em caso afirmativo, a tabela de roteamento local é atualizada com essa nova rota. Se a tabela recebida em um enlace n possuir um custo diferente para as rotas que iniciam em n, o custo armazenado na tabela é substituído por esse novo custo. Esse procedimento é feito porque a tabela recebida foi originada por um nó de roteamento mais próximo de um destino em questão e, portanto, possui informações mais exatas sobre uma determinada rota. Nesse caso, diz-se que este nó de roteamento é *autoritativo* sobre a rota.

O comportamento básico do algoritmo de vetor de distância é descrito pelo programa em pseudocódigo mostrado na Figura 3.9, onde *Tr* é uma tabela recebida de um nó vizinho e *Tl* é a tabela local. Ford e Fulkerson [1962] mostraram que os passos descritos anteriormente são suficientes para garantir que as tabelas de roteamento convirjam para obter as melhores rotas para cada destino, quando houver uma alteração na rede. A freqüência t com que as tabelas de roteamento são propagadas, mesmo quando nenhuma alteração tiver ocorrido, é projetada de modo a garantir a estabilidade do algoritmo de roteamento; para, por exemplo, no caso de alguns pacotes RIP serem perdidos. O valor de t adotado pela Internet é de 30 segundos.

Para tratar das falhas, cada roteador monitora seus enlaces e atua como segue:

Envia: a cada *t* segundos ou quando *Tl* muda, envia *Tl* para todos enlaces não defeituosos.

Recebe: quando uma tabela de roteamento *Tr* é recebida no enlace *n*:

```
Para todas as linhas Rr em Tr {
    if (Rr.enlace ≠ n) {
        Rr.custo = Rr.custo + 1;
        Rr.enlace = n;
        if (Rr.destino não está em Tl) adiciona Rr em Tl;      // adiciona novo destino em Tl
        else para todas as linhas Rl em Tl {
            if (Rr.destino = Rl.destino e
            (Rr.custo < Rl.custo ou Rl.enlace = n)) Rl = Rr;
            // Rr.custo < Rl.custo : o nó remoto tem uma rota melhor
            // Rl.link = n : o nó remoto é autoritativo.
        }
    }
}
```

Figura 3.9 Pseudo-código do algoritmo de roteamento RIP.

Quando um enlace defeituoso n é detectado, configura o custo como ∞ para todas as entradas da tabela local que se referem a esse enlace defeituoso e executa a ação *Envia*.

Assim, a informação de que o enlace está danificado é representada por um valor infinito do custo para os destinos que são alcançáveis por ele. Quando essa informação for propagada para os nós vizinhos, ela será processada de acordo com a ação *Recebe* (note que ∞ + 1 = ∞) e, então, propagada novamente, até que seja atingido um nó que tenha uma rota que funcione para os destinos relevantes, caso houver uma. O nó que ainda tem a rota que funciona finalmente propagará sua tabela e a rota que funciona substituirá a defeituosa em todos os nós.

O algoritmo do vetor de distância pode ser melhorado de várias maneiras: os custos (também conhecidos como *métricas*) podem ser baseados nas larguras de banda dos enlaces; o algoritmo pode ser modificado de forma a aumentar sua convergência e evitar que alguns estados intermediários indesejados, como laços de encaminhamento, ocorram antes que a convergência seja obtida. Um protocolo de informações de roteamento com esses aprimoramentos foi o primeiro protocolo de roteamento empregado na Internet, agora conhecido como RIP-1 e descrito no RFC 1058 [Hedrick 1988]. Mas as soluções para os problemas causados pela convergência lenta não são totalmente eficazes e isso leva a um roteamento ineficiente e à perda de pacotes enquanto a rede está em estados intermediários.

Existe uma série de desenvolvimentos subseqüentes de algoritmos de roteamento, no sentido de aumentar o conhecimento da rede mantido em cada nó. A família mais importante dos algoritmos desse tipo são os algoritmos de estado de enlace. Eles são baseados na distribuição e na atualização de um banco de dados em cada nó, representando toda a rede ou uma parte substancial dela. Cada nó é então responsável por calcular as rotas ótimas para os destinos mostrados em seu banco de dados. Esse cálculo pode ser feito por uma variedade de algoritmos, alguns dos quais evitam conhecidos problemas no algoritmo de Bellman–Ford, como a convergência lenta e os estados intermediários indesejáveis. O projeto de algoritmos de roteamento é um assunto muito importante e nossa discussão, aqui, sobre eles, é necessariamente limitada. Voltaremos a esse ponto na Seção 3.4.3, com uma descrição do funcionamento do algoritmo RIP-1, um dos primeiros utilizados para roteamento de IP e ainda em uso em muitas partes da Internet. Para uma abordagem ampla sobre roteamento na Internet, consulte Huitema [2000] e, para mais material sobre algoritmos de roteamento em geral, consulte Tanenbaum [2003].

3.3.6 Controle de congestionamento

A capacidade de uma rede é limitada pelo desempenho de seus enlaces de comunicação e nós de comutação. Quando a carga em um enlace, ou em um nó específico, se aproximar de sua capacidade máxima, serão criadas filas de espera nos *hosts* que estão tentando enviar pacotes e nos nós interme-

diários que contêm pacotes cuja transmissão para diante está bloqueada por outro tráfego. Se a carga continuar no mesmo nível alto, as filas continuarão a crescer até atingirem o limite do espaço de buffer disponível.

Quando esse estado é atingido em um nó, o nó não tem outra opção a não ser descartar os novos pacotes recebidos. Conforme já mencionamos, a perda ocasional de pacotes na camada de rede é aceitável e pode ser corrigida por retransmissões iniciadas pelas camadas superiores. Mas se a taxa de perda e a retransmissão de pacotes atingirem um nível significativo, o efeito sobre o desempenho da rede poderá ser devastador. É fácil ver por que isso acontece: se pacotes são eliminados em nós intermediários, os recursos da rede que eles já consumiram são desperdiçados e as retransmissões resultantes exigirão uma quantidade de recursos semelhante para atingir o mesmo ponto na rede. Como regra geral, quando a carga em uma rede ultrapassa 80% de sua capacidade nominal, o desempenho de saída total tende a cair como resultado das perdas de pacote, a não ser que a utilização de enlaces muito sobrecarregados seja controlada.

Em vez de permitir que os pacotes trafeguem pela rede até atingirem nós congestionados, onde serão eliminados, seria melhor mantê-los em nós anteriores, até que o nível de congestionamento seja reduzido. Isso resultará em atrasos maiores para os pacotes, mas não degradará significativamente o desempenho total da rede. *Controle de congestionamento* é o nome dado às técnicas empregadas para se obter isso.

Em geral, o controle de congestionamento é feito informando-se os nós ao longo de uma rota de que ocorreu o congestionamento e que, portanto, sua taxa de transmissão de pacotes deve ser reduzida. Nos nós intermediários, isso implica em bufferizar os pacotes recebidos por um período maior. Para *hosts* que são origens de pacotes, o resultado pode ser o enfileiramento dos pacotes antes da transmissão ou o bloqueio do processo aplicativo que os está gerando, até que a rede possa manipulá-los.

Todas as camadas de rede baseadas em datagrama, incluindo IP e Ethernet, contam apenas com um controle de tráfego fim-a-fim, isto é, o nó emissor deve reduzir a taxa com que transmite pacotes com base somente na informação que recebe do destinatário. A informação de congestionamento é fornecida para o nó que está enviando os pacotes por meio da transmissão explícita de mensagens especiais (chamadas de *pacotes reguladores – choke packets*) solicitando uma redução na taxa de transmissão, ou pela implementação de um protocolo de controle de transmissão específico (do qual o TCP deriva seu nome – a Seção 3.4.6 explica o mecanismo usado no TCP) ou ainda, pela observação da ocorrência de pacotes eliminados (caso o protocolo seja um no qual cada pacote é confirmado).

Em algumas redes baseadas em circuito virtual, a informação de congestionamento pode ser recebida e tratada em cada nó. Embora o ATM utilize circuito virtual, ele conta com gerenciamento de qualidade do serviço (veja a Seção 3.5.4 e o Capítulo 17) para garantir que cada circuito possa transmitir o tráfego exigido.

3.3.7 Interligação de redes

Existem muitas tecnologias de rede com diferentes protocolos de camadas de rede, enlace e física. As redes locais são construídas a partir das tecnologias Ethernet e ATM, as redes de longa distância são construídas sobre redes telefônicas digitais e analógicas de vários tipos, enlaces de satélite e redes ATM. Os computadores individuais e as redes locais são ligadas à Internet ou às intranets por conexões de modems, sem fio e DSL.

Para construir uma rede integrada, ou seja, para interligar redes, devemos associar muitas sub-redes, cada uma das quais baseada em uma dessas tecnologias de rede. Para tornar isso possível, é necessário o seguinte:

1. Um esquema de endereçamento unificado de rede que permita os pacotes serem endereçados para qualquer *host* conectado a qualquer sub-rede.
2. Um protocolo definindo o formato dos pacotes de rede e fornecendo regras de acordo com as quais eles são manipulados.
3. Interconectar componentes que direcionam pacotes para seus destinos, em termos de endereços unificados de rede, transmitindo os pacotes usando sub-redes com uma variedade de tecnologias de rede.

Para a Internet, (1) é fornecido por endereços IP, (2) é o protocolo IP e (3) é realizado por componentes chamados *roteadores Internet*. O protocolo IP e o endereçamento de IP serão descritos com alguns detalhes na Seção 3.4. Aqui, vamos descrever as funções dos roteadores Internet e alguns outros componentes usados para integrar as redes.

A Figura 3.10 mostra uma pequena parte da intranet localizada no Queen Mary College da Universidade de Londres (QMUL), conforme se encontrava em meados de 2000. Muitos dos detalhes mostrados serão explicados em seções posteriores. Aqui, notamos que a parte mostrada na figura compreende várias sub-redes interconectadas por roteadores. Existem cinco sub-redes; três das quais compartilham a rede IP 138.37.95 (usando o esquema de roteamento interdomínios sem classe – *Classless InterDomain Routing* – descrito na Seção 3.4.3). Os números no diagrama são endereços IP; sua estrutura será explicada na Seção 3.4.1. Os roteadores no diagrama são membros de várias sub-redes e têm um endereço IP para cada sub-rede, mostrado nos enlaces de conexão.

Os roteadores (nomes de *host hammer* e *sickle*) são, na verdade, computadores de propósito geral que também cumprem outros objetivos. Um desses objetivos é servir como *firewalls*; a função de um *firewall* está intimamente ligada à função de roteamento, conforme vamos descrever a seguir. A sub-rede 138.37.95.232/29 não está conectada ao restante da rede em nível IP. Apenas o servidor de arquivos *custard* pode acessá-la para fornecer um serviço de impressão por intermédio de um processo que monitora e controla o uso das impressoras.

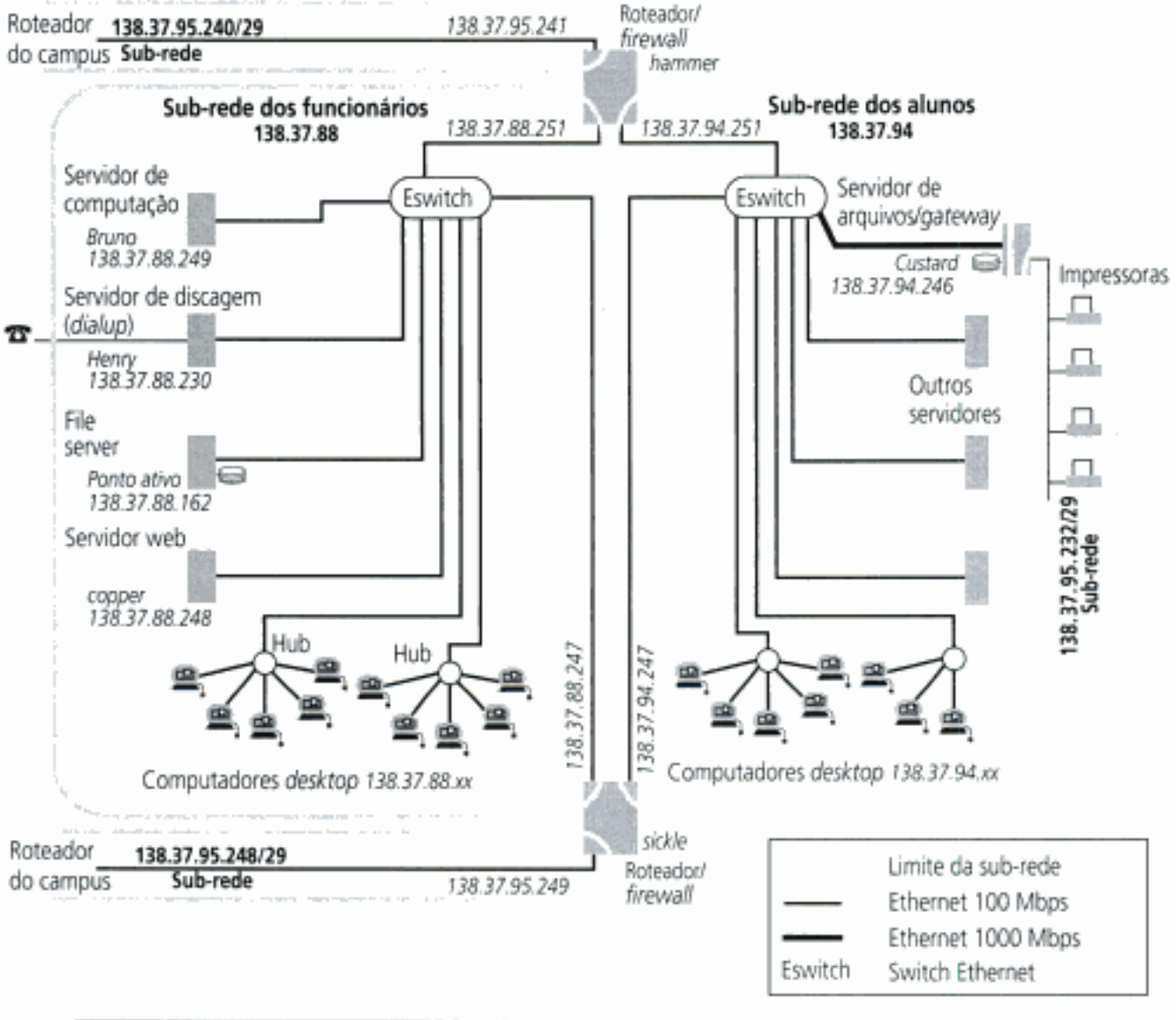


Figura 3.10 Visão simplificada da rede de ciência da computação do QMUL (em meados de 2000).

Todos os enlaces da Figura 3.10 são Ethernet. A largura de banda da maioria deles é de 100 Mbps, mas um é de 1000 Mbps, pois transporta um alto volume de tráfego entre um grande número de computadores usados pelos alunos e o servidor de arquivos *custard*, que contém todos os arquivos deles.

Existem dois *switches* e vários hubs Ethernet na parte da rede ilustrada. Os dois tipos de componentes são transparentes para os datagramas IP. Um hub Ethernet é apenas um meio de conectar vários segmentos de cabo Ethernet, todos os quais formando uma única rede física Ethernet. Todos os pacotes Ethernet enviados por um *host* são retransmitidos para todos os segmentos. Um *switch* Ethernet conecta várias redes Ethernet, mas direciona os pacotes apenas para a rede Ethernet na qual o *host* de destino está conectado.

Roteadores ◊ Mencionamos que o roteamento é exigido em todas as redes, exceto aquelas nas quais todos os *hosts* são conectados por um único meio de transmissão, como as Ethernet e as redes sem fio. A Figura 3.7 mostra uma rede com cinco roteadores conectados por seis enlaces. Na interligação de várias redes, os roteadores podem ser conectados diretamente entre si, como mostra a Figura 3.7, ou por meio de sub-redes, como mostrado para *custard* na Figura 3.10. Nos dois casos, os roteadores são responsáveis por encaminhar os pacotes de rede que chegam em um enlace qualquer para a saída apropriada. Conforme explicado anteriormente, eles mantêm tabelas de roteamento para esse propósito.

Pontes (*bridges*) ◊ As pontes ligam redes de diferentes tipos. Algumas pontes ligam várias redes e essas são denominadas de pontes/roteadores, pois também executam funções de roteamento. Por exemplo, a rede do campus no QMW inclui um *backbone* FDDI (*Fibre Distributed Data Interface*) (não mostrado na Figura 3.10), e esse *backbone* está ligado às sub-redes da figura por meio de pontes/roteadores.

Hubs ◊ Hubs são simplesmente uma maneira conveniente de conectar *hosts* e estender segmentos Ethernet e outras tecnologias de rede local baseadas em transmissão *broadcast*. Eles têm várias, comumente de 4 a 64, tomadas, ou portas, onde um computador *host* pode ser conectado. Eles também podem ser usados para superar as limitações de distância de um segmento e fornecer um modo de acrescentar mais *hosts* na rede.

Switches ◊ Os *switches* executam uma função semelhante aos roteadores, mas apenas para redes locais (normalmente, Ethernet). Isto é, eles interligam várias redes Ethernet separadas, direcionando os pacotes recebidos para a rede de saída apropriada. Eles executam sua tarefa no nível do protocolo de enlace Ethernet. Os *switches* quando iniciam sua operação não possuem nenhum conhecimento sobre as redes que interligam e começam a construir suas tabelas de roteamento observando o tráfego. Na falta da informação para onde direcionar um pacote, eles direcionam para todas as suas portas de saída. Isso é denominado de *broadcast*.

A vantagem dos *switches* em relação aos hubs é que eles retransmitem o tráfego recebido apenas para a rede de saída relevante, reduzindo o congestionamento nas outras redes em que estão conectados.

Tunelamento ◊ As pontes e os roteadores transmitem pacotes em uma variedade de redes subjacentes, adequando os pacotes ao tipo de tecnologia empregada por seus enlaces. No entanto, há uma situação em que, sem o uso de um protocolo de rede, a tecnologia de rede subjacente fica oculta das camadas que estão acima dele. Dois nós conectados em redes separadas, de um mesmo tipo, podem se comunicar por meio de uma rede de tipo diferente. Isso é possível através da construção de um "túnel" de protocolo. Um túnel de protocolo é uma camada de software que transmite pacotes em um ambiente de rede diferente daquele que nativamente eles existem.

A analogia a seguir explica o motivo da escolha da terminologia e fornece outra maneira de pensar sobre o uso de túneis. Um túnel através de uma montanha permite a existência de uma estrada para transportar carros onde, de outro modo, isso seria impossível. A estrada é contínua – o túnel é transparente para a aplicação (carros). A estrada é o mecanismo de transporte e o túnel permite que ela funcione em um ambiente estranho.

A Figura 3.11 ilustra o uso de túneis na migração da Internet para o protocolo IPv6. O protocolo IPv6 se destina a substituir a versão de IP atualmente em uso, o IPv4, e ambos são incompatíveis entre si. (Tanto o protocolo IPv4 como o IPv6 serão descritos na Seção 3.4.) Durante o período de transição

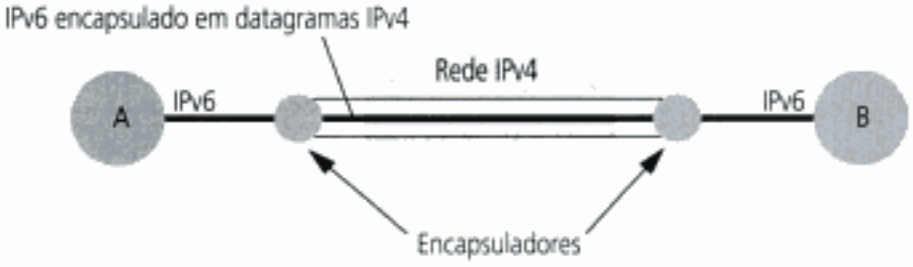


Figura 3.11 Utilização de túnel para migração para o IPv6.

para o IPv6, haverão "ilhas" de interligação em rede IPv6 no mar do IPv4. Em nossa ilustração, A e B são essas ilhas. Nos limites das ilhas, os pacotes IPv6 são encapsulados em IPv4 e, dessa maneira, transportados pelas redes IPv4 intervenientes.

Como outro exemplo, o MobileIP (descrito na Seção 3.4.5) transmite datagramas IP para *hosts* móveis em qualquer parte da Internet, construindo um túnel para eles a partir de sua base de origem. Os nós das redes intervenientes não precisam ser modificados para manipular o protocolo MobileIP. O protocolo *multicast* IP opera de maneira semelhante. Para os roteadores que oferecem suporte a *multicast,* o protocolo explora essa capacidade e, para os demais, envia os pacotes de *multicast* encapsulando-os sobre o IP padrão. O protocolo PPP para a transmissão de datagramas IP sobre enlaces seriais fornece mais um exemplo.

3.4 Protocolos Internet

Aqui, descreveremos os principais recursos da pilha de protocolos TCP/IP e discutiremos suas vantagens e limitações quando usados em sistemas distribuídos.

A Internet surgiu a partir de duas décadas de pesquisa e desenvolvimento sobre redes de longa distância nos EUA, começando no início dos anos 70 com a ARPANET – a primeira rede de computadores em larga escala desenvolvida [Leiner *et al.* 1997]. Uma parte importante dessa pesquisa foi o desenvolvimento da pilha de protocolos TCP/IP. TCP significa *Transmission Control Protocol* (protocolo de controle de transmissão) e IP é *Internet Protocol* (protocolo Internet). A ampla adoção do protocolo TCP/IP e dos protocolos de aplicação da Internet em redes de pesquisa em nível nacional e, mais recentemente, em redes comerciais de muitos países, permitiu a integração dessas em uma única rede que cresceu de forma extremamente rápida, até seu tamanho atual, com mais de 60 milhões de *hosts*. Atualmente, existem muitos serviços e protocolos em nível de aplicação baseados em TCP/IP, incluindo, entre outros, a web (HTTP), e-mail (SMTP, POP), *netnews* (NNTP), transferência de arquivos (FTP) e o Telnet (telnet). O TCP é um protocolo de transporte; ele pode ser usado para suportar aplicativos diretamente, ou protocolos adicionais podem ser dispostos em camadas sobre ele para fornecer recursos adicionais. Por exemplo, o protocolo HTTP é normalmente transportado diretamente sobre o TCP, mas quando é exigida segurança fim-a-fim, o protocolo TLS (*Transport Layer Security*), descrito na Seção 7.6.3, é disposto em uma camada sobre o TCP para produzir canais seguros para transmitir as mensagens HTTP.

Originalmente, os protocolos Internet foram desenvolvidos principalmente para suportar aplicativos remotos simples, como transferências de arquivos e correio eletrônico, envolvendo comunicação com latências relativamente altas entre computadores geograficamente dispersos. No entanto, eles se mostraram eficientes o bastante em redes locais e de longa distância para suportar os requisitos de muitos aplicativos distribuídos, e agora são quase universalmente usados nos sistemas distribuídos. A padronização resultante dos protocolos de comunicação tem trazido imensos benefícios.

A ilustração geral das camadas de protocolos da Figura 3.6 é transformada, no caso específico da Internet, para a da Figura 3.12. Existem dois protocolos de transporte – TCP (*Transmission Control Protocol*) e UDP (*User Datagram Protocol*). TCP é um protocolo confiável, orientado a conexão, e UDP é um protocolo baseado em datagrama que não garante uma transmissão confiável. O *Internet*

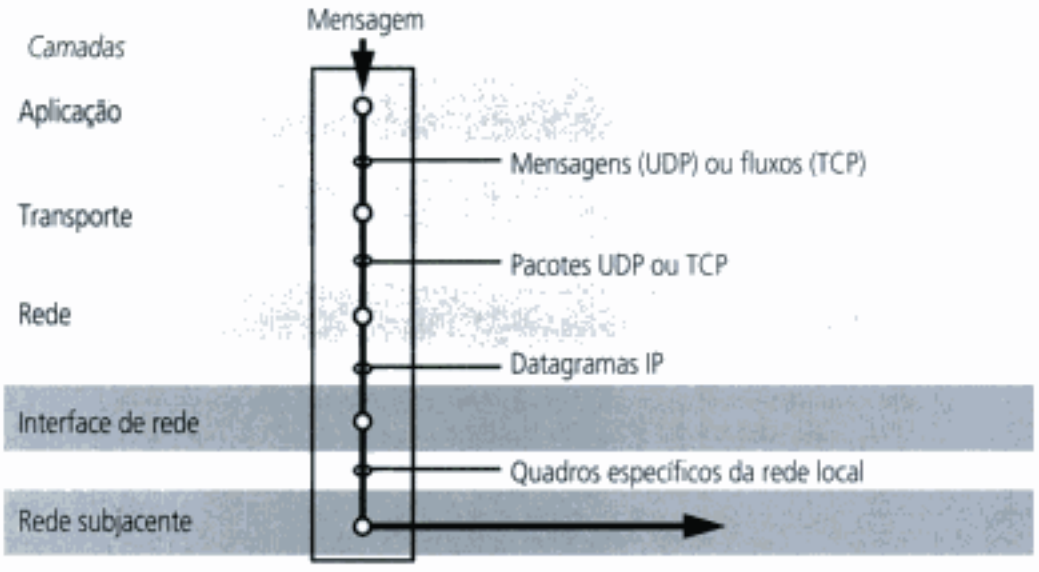


Figura 3.12 Camadas TCP/IP.

Protocol (IP) é o protocolo da "rede virtual" Internet – isto é, datagramas IP fornecem o mecanismo de transmissão básico da Internet e de outras redes TCP/IP. Colocamos a palavra "rede" entre aspas na frase anterior porque ela não é a única camada envolvida na implementação da comunicação na Internet. Isso porque os protocolos Internet normalmente são dispostos em camadas sobre outras tecnologias de rede locais, como a Ethernet, a qual permite aos computadores ligados em uma mesma rede local trocarem pacotes de dados (quadros Ethernet). A Figura 3.13 ilustra o encapsulamento de dados que ocorre na transmissão de uma mensagem via TCP sobre uma rede Ethernet. Os cabeçalhos de uma camada possuem rótulos que indicam o tipo de protocolo da camada que está acima. Isso é necessário para que a pilha de protocolos do lado destino possa desempacotar os pacotes corretamente. Na camada TCP, o número da porta destino tem um propósito semelhante: permitir que o módulo de software TCP no *host* de destino repasse a mensagem para um processo específico da camada de aplicação.

As especificações do protocolo TCP/IP [Postel 1981a; 1981b] não definem as camadas inferiores à camada do datagrama Internet. Para sua transmissão, os datagramas IP são apropriadamente formatados nos quadros de dados das tecnologias de redes locais ou enlaces subjacentes.

Por exemplo, a princípio, o protocolo IP era executado na ARPANET, que consistia em *hosts* e uma versão inicial de roteadores (chamados PSEs), conectados por enlaces de dados de longa distância. Atualmente, o IP é usado em praticamente todas as tecnologias de rede conhecidas, incluindo ATM, redes locais como Ethernet e redes *token ring*. O protocolo IP é implementado sobre linhas se-

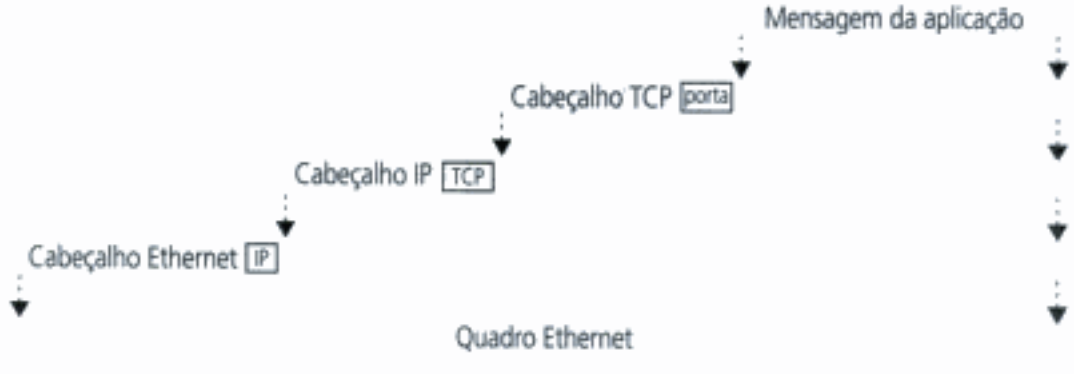


Figura 3.13 Encapsulamento que ocorre quando uma mensagem é transmitida via TCP sobre uma rede local Ethernet.

riais e circuitos telefônicos, por intermédio do protocolo PPP [Parker 1992], permitindo ser utilizado em comunicações via modem e outros canais seriais.

O sucesso do protocolo TCP/IP é devido à sua independência em relação à tecnologia de transmissão subjacente, o que permite a interligação de muitas redes e enlaces de dados heterogêneos. Os usuários e os programas aplicativos percebem uma única rede virtual suportando TCP e UDP, e os desenvolvedores de programas baseados em TCP e UDP vêem uma única rede IP virtual, ocultando a diversidade da mídia de transmissão subjacente. A Figura 3.14 ilustra essa visão.

Nas próximas duas seções, descreveremos o esquema de endereçamento e o protocolo IP. O *Domain Name System*, que converte em endereços IP os nomes de domínio como *www.amazon.com*, *hpl.hp.com*, *stanford.edu* e *qmw.ac.uk*, com os quais os usuários da Internet estão tão familiarizados será apresentado na Seção 3.4.7 e descrito de forma mais completa no Capítulo 9.

A versão predominante por toda a Internet é o IPv4 (desde janeiro de 1984) e essa é a versão que vamos descrever nas duas próximas seções. Porém, o rápido crescimento da Internet levou à publicação da especificação de uma nova versão, o IPv6, para superar as limitações de endereçamento do IPv4 e acrescentar recursos para suportar alguns novos requisitos. Descreveremos a versão IPv6 na Seção 3.4.4. Devido à enorme quantidade de software que será afetada, uma migração gradual para a versão IPv6 está sendo planejada para um período de dez anos ou mais.

Figura 3.14 Visão conceitual de um programador de uma rede TCP/IP.

3.4.1 Endereçamento IP

Talvez o aspecto mais desafiador do projeto dos protocolos Internet tenha sido a construção de esquemas para a atribuição de endereços a *hosts* e de roteamento dos datagramas IP para seus destinos. O esquema usado para atribuição de endereços para redes e para os *hosts* nelas conectados tinha que satisfazer os seguintes requisitos:

- Ser universal – qualquer *host* deveria conseguir enviar pacotes para qualquer outro *host* na Internet.
- Ser eficiente no uso do espaço de endereçamento – é impossível prever o tamanho final da Internet e o número de endereços de rede e *hosts* que provavelmente será exigido. O espaço de endereçamento deveria ser cuidadosamente particionado para garantir que os endereços não acabassem. Em 1978–82, quando as especificações dos protocolos TCP/IP estavam sendo desenvolvidas, e o aprovisionamento de 2^{32} de endereços, ou seja, aproximadamente 4 bilhões (praticamente igual à população mundial da época), era considerado adequado. Esse julgamento se mostrou imprevidente por dois motivos:
  - a taxa de crescimento da Internet superou em muito todas as previsões;
  - o espaço de endereços foi alocado e usado de forma muito menos eficiente do que o esperado.
- Servir para o desenvolvimento de um esquema de roteamento flexível e eficiente, mas sem que os endereços em si contivessem toda informação necessária para direcionar um datagrama até seu destino.

O esquema escolhido atribui um endereço IP para cada *host* na Internet – um identificador numérico de 32 bits, contendo um identificador de rede, que identifica exclusivamente uma sub-rede na

Figura 3.15 Estrutura de endereços Internet, mostrando os tamanhos dos campos em bits.

Internet, e um identificador de *host*, que identifica exclusivamente um *host* nessa rede. São esses os endereços colocados nos datagramas IP e usados para direcioná-los até seus destinos.

O projeto adotado para o espaço de endereçamento da Internet aparece na Figura 3.15. Existem quatro classes alocadas de endereços Internet – A, B, C e D. A classe D é reservada para comunicação *multicast* na Internet e, atualmente, é suportada apenas em alguns roteadores e será melhor discutida na Seção 4.5.1. A classe E contém um intervalo de endereços não alocados, os quais estão reservados para requisitos futuros.

Os endereços Internet são um número em 32 bits, que contêm um identificador de rede e um identificador de *host*, são normalmente escritos como uma seqüência de quatro números decimais separados por pontos. Cada número decimal representa um dos quatro bytes, ou *octetos*, do endereço IP. Os valores permitidos para cada classe de endereço de rede aparecem na Figura 3.16.

Três classes de endereço foram projetadas para satisfazer os requisitos de diferentes tipos de organização. Os endereços de classe A, com uma capacidade de 2^{24} *hosts* em cada sub-rede, são reservados para redes muito grandes, como a US NSFNet e outras redes de longa distância em nível nacional. Os endereços de classe B são alocados para organizações que possuem redes que provavel-

	Octeto 1	*Octeto 2*	*Octeto 3*		*Intervalo de endereçamento*
	ID de rede		*ID de host*		
Classe A:	1 a 127	0 a 255	0 a 255	0 a 255	1.0.0.0 a 127.255.255.255
	ID de rede		*ID de host*		
Classe B:	128 a 191	0 a 255	0 a 255	0 a 255	128.0.0.0 a 191.255.255.255
	ID de rede			*ID de host*	
Classe C:	192 a 223	0 a 255	0 a 255	1 a 254	192.0.0.0 a 223.255.255.255
	Endereço de multicast				
Classe D (*multicast*):	224 a 239	0 a 255	0 a 255	1 a 254	224.0.0.0 a 239.255.255.255
Classe E (reservada):	240 a 255	0 a 255	0 a 255	1 a 254	240.0.0.0 a 255.255.255.255

Figura 3.16 Representação decimal dos endereços Internet.

mente conterão mais de 255 computadores e os endereços da classe C são usados nos demais tipos de redes restantes.

Os endereços Internet com identificadores de *host* 0, ou com todos seus bits em 1 (binário), são usados para propósitos especiais. Os endereços com identificador de *host* igual a 0 são usados para se referenciar *esta rede*. Aqueles com o identificador de *host* com todos seus bits igual a 1 é usado para endereçar uma mensagem para todos os *hosts* conectados na rede especificada na parte do identificador de rede do endereço. Esse endereço é denominado de endereço de *broadcast*.

Para as organizações com redes conectadas a Internet, os identificadores de rede são alocados pela IANA (*Internet Assigned Numbers Authority*). Os identificadores de *host* para os computadores de cada rede são atribuídos pelo administrador da rede em questão.

Como os endereços IP incluem um identificador de rede, qualquer computador que esteja conectado em mais de uma rede deve ter endereços separados em cada uma delas, e quando um computador é movido para uma rede diferente, seu endereço IP deve ser alterado. Esses requisitos podem levar a sobrecargas administrativas substanciais; por exemplo, no caso dos computadores portáteis.

Na prática, o esquema de alocação de endereço IP não se mostrou muito eficiente. A principal dificuldade é que os administradores de rede das diversas organizações não podem prever facilmente o crescimento futuro de suas necessidades de endereços IP e tendem a superestimá-las, solicitando endereços de Classe B, quando estão em dúvida. Por volta de 1990, tornou-se evidente que, com base na taxa de alocação da época, os endereços IP provavelmente estariam esgotados em 1996. Três medidas foram tomadas. A primeira foi iniciar o desenvolvimento de um novo esquema de protocolo e endereçamento IP, cujo resultado foi a especificação do IPv6.

A segunda medida foi modificar radicalmente a maneira pela qual os endereços IP eram alocados. Foi introduzido um novo esquema de alocação de endereços e roteamento, projetado para fazer uso mais eficiente do espaço de endereços IP, chamado CIDR (*Classless Interdomain Routing* – roteamento entre domínios sem classes). Descreveremos o CIDR na Seção 3.4.3. A rede local ilustrada na Figura 3.10 inclui várias sub-redes de classe C, no intervalo 138.37.88–138.37.95, ligadas por roteadores. Os roteadores gerenciam a distribuição de datagramas IP para todas as sub-redes e também manipulam o tráfego delas para o resto do mundo. A figura também ilustra o uso de CIDR para subdividir um espaço de endereços de classe B para produzir várias sub-redes classe C.

A terceira medida foi permitir que computadores não acessassem a Internet indiretamente, por meio de dispositivos que implementam um esquema NAT (*Network Address Translation*). Descreveremos esse esquema na Seção 3.4.3.

3.4.2 O protocolo IP

O protocolo IP transmite datagramas de um *host* para outro, se necessário por meio de roteadores intermediários. O formato completo do pacote IP é bastante complexo, mas a Figura 3.17 mostra os principais componentes. Existem vários campos de cabeçalho, não mostrados no diagrama, que são usados pelos algoritmos de transmissão e roteamento.

O protocolo IP fornece um serviço de entrega que oferece uma semântica descrita como *não confiável* ou de *melhor esforço* (*best effort delivery*), pois não há garantia de entrega dos datagramas. Os datagramas podem ser perdidos, duplicados, retardados ou entregues fora de ordem, mas esses erros surgem apenas quando as redes subjacentes falham ou os buffers do destino estão cheios. O IP possui uma única soma de verificação que é feita sobre os bytes que formam seu cabeçalho, cujo cálculo não é dispendioso e garante que quaisquer modificações nos endereços IP fonte/destino, ou nos dados de controle, do datagrama sejam detectadas. Não existe nenhuma soma de verificação na área de dados,

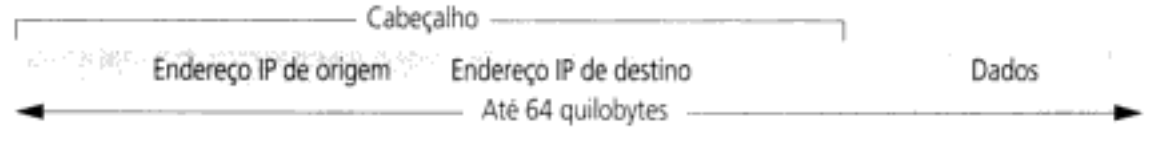


Figura 3.17 Leiaute de um datagrama IP.

o que evita sobrecargas ao passar por roteadores, deixando os protocolos de nível mais alto (TCP e UDP) fornecerem suas próprias somas de verificação – um exemplo prático do princípio fim-a-fim (Seção 2.2.1).

A camada IP coloca datagramas IP em um formato conveniente para transmissão na rede subjacente (a qual poderia, por exemplo, ser uma Ethernet). Quando um datagrama IP é maior do que o MTU da rede subjacente, ele é dividido na origem em datagramas menores e novamente montado em seu destino final. Esse procedimento de divisão se denomina segmentação ou fragmentação. Os datagramas resultantes podem ser subdivididos ainda mais, para estarem de acordo com as redes subjacentes encontradas durante a jornada da origem até o destino. (Cada fragmento tem um identificador e um número de seqüência para permitir a remontagem correta.)

A camada IP também deve inserir um endereço físico de rede do destino da mensagem na rede subjacente. Ela obtém isso através do módulo de resolução de endereços que será descrito a seguir.

Resolução de endereços ◊ O módulo de resolução de endereços é responsável por converter endereços Internet em endereços físicos de rede para uma rede subjacente específica. Por exemplo, se a rede subjacente é uma Ethernet, o módulo de resolução de endereços obtém o endereço Ethernet (48 bits) associado a um dado endereço IP (32 bits).

Essa transformação depende da tecnologia da rede:

- Quando os *hosts* são conectados diretamente, ponto a ponto; os datagramas IP podem ser direcionados para eles sem a transformação de endereço.
- Algumas redes locais permitem que endereços físicos de rede sejam atribuídos dinamicamente aos *hosts* e esses endereços podem ser convenientemente escolhidos para corresponder à parte do identificador de *host* do endereço de Internet – a resolução é simplesmente uma questão de extrair o identificador de *host* do endereço IP.
- Para redes Ethernet, e algumas outras redes locais, o endereço físico de rede de cada computador é incorporado no hardware de sua interface de rede e não tem nenhuma relação direta com seu endereço IP – a resolução depende do conhecimento da correspondência entre endereços IP e endereços físicos dos *hosts* na rede local e é feita usando um protocolo de resolução de endereços (*Address Resolution Protocol* – ARP).

Agora, vamos esboçar a implementação do ARP para redes Ethernet. Ele é baseado na troca de mensagens de requisição e resposta, mas explora o uso de cache para minimizar esse tráfego. Considere primeiramente o caso em que um computador *host* conectado a uma rede Ethernet utiliza IP para transmitir uma mensagem para outro computador na mesma rede Ethernet. O módulo de software IP no computador remetente deve transformar o endereço IP do destino em um endereço Ethernet, antes que o pacote possa ser enviado. Para fazer isso, ele ativa o módulo ARP no computador remetente.

O módulo ARP em cada *host* mantém uma cache de pares (*endereço IP, endereço Ethernet*) obtidos anteriormente. Se o endereço IP exigido está na cache, então a consulta é respondida imediatamente. Caso contrário, o módulo ARP transmite, em broadcast Ethernet, uma mensagem de requisição ARP na rede Ethernet local, contendo o endereço IP desejado. Cada um dos computadores da rede local Ethernet recebe o quadro de requisição ARP e verifica se o endereço IP nele contido corresponde ao seu próprio endereço IP. Se corresponder, uma mensagem de resposta ARP é enviada para a origem da requisição ARP, contendo o endereço Ethernet do remetente; caso contrário, a mensagem de requisição ARP é ignorada. O módulo ARP da origem adiciona o novo mapeamento *endereço IP* → *endereço Ethernet* em sua cache local de pares (*endereço IP, endereço Ethernet*), para que seja possível responder a pedidos semelhantes no futuro, sem propagar uma nova requisição ARP. Com o passar do tempo, a cache ARP de cada computador conterá o par (*endereço IP, endereço Ethernet*) de todos os computadores para os quais são enviados datagramas IP. Assim, o *broadcast* ARP só será necessário quando um computador tiver sido recentemente conectado na Ethernet local.

***Spoofing* de IP** ◊ Já vimos que os datagramas IP incluem um endereço de origem – o endereço IP do computador remetente, junto com um endereço de porta encapsulado no campo de dados (para mensagens UDP e TCP), os quais são usados pelos servidores para gerar um endereço de retorno. Infelizmente, não é possível garantir que o endereço de origem dado seja de fato o endereço do remeten-

te. Um remetente mal-intencionado pode substituir facilmente seu endereço por um diferente do seu próprio. Essa brecha tem sido a fonte de vários ataques conhecidos, incluindo os ataques de negação de serviço (*Denial of Service* – DoS), de fevereiro de 2000 [Farrow 2000], mencionado no Capítulo 1, Seção 1.4.3. O método usado foi efetuar muitos pedidos do serviço *ping* para um grande número de computadores em diversos sites (o *ping* é um serviço simples, projetado para verificar a disponibilidade de um *host*). Todos esses pedidos mal-intencionados de *ping* continham o endereço IP de um computador-alvo em seu campo de endereço do remetente. Portanto, as respostas do *ping* foram todas direcionadas para o alvo, cujos buffers de entrada ficaram sobrecarregados, impedindo a passagem de datagramas IP legítimos. Esse ataque será melhor discutido no Capítulo 7.

3.4.3 Roteamento IP

A camada IP direciona os datagramas de sua origem para seu destino. Cada roteador na Internet possui pelo menos um tipo de algoritmo de roteamento implementado.

Backbones ◊ A topologia da Internet é conceitualmente particionada em *sistemas autônomos* (SA), os quais são subdivididos em *áreas*. As intranets da maioria das grandes organizações, como universidades e empresas, são consideradas SAs, e normalmente incluem várias áreas. Na Figura 3.10, a intranet do campus é um SA e a parte mostrada é uma área. Todo SA tem uma área de *backbone*. O conjunto de roteadores que conectam áreas que não são *backbone* com o *backbone* e os enlaces que interconectam esses roteadores são chamados de *backbone* da rede. Os enlaces do *backbone* normalmente têm largura de banda alta e, por questão de confiabilidade, são replicados. Essa estrutura hierárquica é conceitual e explorada principalmente para o gerenciamento de recursos e manutenção dos componentes. Ela não afeta o roteamento de datagramas IP.

Protocolos de roteamento ◊ O RIP-1, o primeiro algoritmo de roteamento usado na Internet, é uma versão do algoritmo de vetor de distância descrito na Seção 3.3.5. O RIP-2 (descrito no RFC 1388 [Malkin 1993]) foi desenvolvido subseqüentemente a partir dele, para acomodar vários requisitos adicionais, incluindo roteamento CIDR, melhor suporte a roteamento *multicast* e a necessidade de autenticação de mensagens RIP para evitar ataques nos roteadores.

Como a escala da Internet se expandiu e a capacidade de processamento dos roteadores aumentou, houve uma mudança no sentido de adotar algoritmos que não sofram com os problemas de convergência lenta e da instabilidade potencial dos algoritmos de vetor de distância. A direção dessa mudança é a classe de algoritmos de estado de enlace, mencionados na Seção 3.3.5, e o algoritmo chamado *caminho mais curto primeiro* (OSPF – *Open Shortest Path First*). A letra O é originada do fato que esse protocolo segue uma filosofia de padrão aberto (*open*). O protocolo OSPF é baseado em um algoritmo de descoberta de caminho de Dijkstra [1959], e tem mostrado convergir mais rapidamente do que o algoritmo RIP.

Devemos notar que a adoção de novos algoritmos de roteamento em roteadores IP pode ocorrer paulatinamente. Uma mudança no algoritmo de roteamento resulta em uma nova versão do protocolo RIP, e o número de versão é identificado em cada mensagem RIP. O protocolo IP não muda quando uma nova versão do protocolo RIP é introduzida. Qualquer roteador IP encaminhará corretamente os datagramas IP recebidos por uma rota razoável, senão ótima, qualquer que seja a versão de RIP utilizada. Mas para os roteadores cooperarem na atualização de suas tabelas de roteamento, eles devem compartilhar um algoritmo semelhante. Para esse propósito são usadas as áreas definidas anteriormente. Dentro de cada área, é aplicado um único algoritmo de roteamento e os roteadores cooperam na manutenção de suas tabelas de roteamento. Os roteadores que suportam apenas RIP-1 ainda são comuns e coexistem com os roteadores que suportam RIP-2 e OSPF, usando recursos de compatibilidade com versões anteriores incorporados nos protocolos mais recentes.

Em 1993, observações empíricas [Floyd e Jacobson 1993] mostraram que a freqüência de 30 segundos com que os roteadores RIP trocam informações estava produzindo uma periodicidade no desempenho das redes IP. A latência média para transmissões de datagramas IP mostrava um pico em intervalos de 30 segundos. Isso foi devido ao comportamento dos roteadores que executavam o protocolo RIP – ao receber uma mensagem RIP, os roteadores retardavam a transmissão de todos os datagramas IP que mantinham, até que o processo de atualização da tabela de roteamento estivesse concluído para

todas as mensagens RIP recebidas até o momento. Isso tendia a fazer com que os roteadores executassem as ações do RIP lentamente. A correção recomendada foi fazer com que os roteadores adotassem um valor aleatório no intervalo de 15–45 segundos para o período de atualização do RIP.

Rotas padrão ◊ Até agora, nossa discussão sobre algoritmos de roteamento sugeriu que cada roteador mantém uma tabela de roteamento completa, mostrando a rota para cada destino (sub-rede ou *host* diretamente conectado) na Internet. Na escala atual da Internet, isso é claramente impraticável (o número de destinos provavelmente já ultrapassa 1 milhão e ainda está crescendo muito rapidamente).

Duas soluções possíveis para esse problema vêm à mente, e ambas foram adotadas em um esforço para atenuar os efeitos do crescimento da Internet. A primeira solução foi adotar alguma forma de agrupamento de endereços IP. Antes de 1993, a partir de um endereço IP, nada podia ser inferido sobre sua localização. Em 1993, como parte da mudança visando a simplificação e a economia na alocação de endereços IP, que será discutida mais adiante, quando estudarmos o CIDR, decidiu-se que para as futuras alocações seriam aplicadas as seguintes regras:

Os endereços de 194.0.0.0 a 195.255.255.255 ficariam na Europa
Os endereços de 198.0.0.0 a 199.255.255.255 ficariam na América do Norte
Os endereços de 200.0.0.0 a 201.255.255.255 ficariam na América Central e do Sul
Os endereços de 202.0.0.0 a 203.255.255.255 ficariam na Ásia e no Pacífico

Como essas regiões geográficas também correspondem a regiões topológicas bem definidas na Internet, e apenas alguns roteadores *gateway* fornecem acesso a cada região, isso permite uma simplificação substancial das tabelas de roteamento para esses intervalos de endereço. Por exemplo, um roteador fora da Europa pode ter uma única entrada na tabela para o intervalo de endereços 194.0.0.0 a 195.255.255.255. Isso faz com que ele envie todos os datagramas IP com destinos nesse intervalo, em uma mesma rota, para o roteador *gateway* europeu mais próximo. Mas, note que antes da data dessa decisão, os endereços IP eram alocados liberalmente, sem respeitar a topologia ou a geografia. Muitos desses endereços ainda estão em uso e a decisão de 1993 nada fez para reduzir a escala das entradas na tabela de roteamento para esses endereços.

A segunda solução para a explosão no tamanho da tabela de roteamento é mais simples e muito eficiente. Ela é baseada na observação de que a precisão da informação de roteamento pode ser relaxada para a maioria dos roteadores, desde que alguns roteadores-chave, aqueles mais próximos aos enlaces do *backbone*, tenham tabelas de roteamento relativamente completas. Esse relaxamento de precisão assume a forma de uma entrada de destino *padrão* nas tabelas de roteamento. A entrada padrão especifica uma rota a ser usada por todos os datagramas IP cujo destino não está incluído na tabela de roteamento. Para ilustrar isso, considere as Figuras 3.7 e 3.8, e suponha que a tabela de roteamento do nó C seja alterada para mostrar:

Tabela roteamento nó C

Para	*Enlace*	*Custo*
B	2	1
C	local	0
E	5	1
Padrão	5	-

Assim, o nó C não sabe da existência dos nós A e D. Ele direcionará todos os pacotes endereçados a eles para E, por meio do enlace 5. Qual é a conseqüência? Os pacotes endereçados a D atingirão seu destino sem perda de eficiência no roteamento, mas os pacotes endereçados a A farão um salto extra, passando por E e B no caminho. O uso de rotas padrão compensa a perda de eficiência com a redução do tamanho da tabela de roteamento. Mas, em alguns casos, especialmente naqueles em que um roteador é ponto de passagem obrigatória, não há perda de eficiência. O esquema de rota padrão é bastante usado na Internet; nela, nenhum roteador sozinho contém as rotas para todos os destinos.

Roteamento em uma sub-rede local ◊ Os pacotes endereçados para *hosts* que estão na mesma rede que o remetente são transmitidos para o *host* destino em um único salto usando a parte do endereço relativa ao identificador de *host* para obter o endereço físico de rede do *host* destino na rede subjacente. A camada IP simplesmente usa o ARP para obter o endereço físico de rede do destino e depois utiliza a rede subjacente para transmitir os pacotes.

Se a camada IP no computador remetente descobrir que o destino está em uma rede diferente, ela deve enviar a mensagem para o *gateway* da rede local. Ela usa o ARP para obter o endereço físico de rede do *gateway*, normalmente, um roteador, e depois usa a rede subjacente para transmitir o datagrama IP para ele. Os *gateways* são conectados em duas ou mais redes e têm vários endereços IP, um para cada rede em que estão ligados.

Roteamento interdomínios sem classes (CIDR – *Classless InterDomain Routing*) ◊ A falta de endereços IP, mencionada na Seção 3.4.1, levou à introdução, em 1996, desse novo esquema de alocação de endereços e gerenciamento das entradas nas tabelas de roteamento. O problema principal era a escassez de endereços de classe B – aqueles para sub-redes com mais de 255 *hosts* conectados. Muitos endereços de classe C estavam disponíveis. A solução CIDR para esse problema é permitir a alocação de um lote de endereços de classe C adjacentes para uma sub-rede que exija mais de 255 endereços. O esquema CIDR também torna possível subdividir um espaço de endereços de classe B para alocação para várias sub-redes.

Dispor de endereços de classe C em lotes parece uma medida simples, mas apenas se for acompanhada de uma alteração no formato da tabela de roteamento, senão haverá um impacto substancial no tamanho das tabelas e, portanto, na eficiência dos algoritmos que as gerenciam. A mudança adotada foi adicionar um campo de *máscara* nas tabelas de roteamento. A máscara é um padrão de bits usado para selecionar a parte de um endereço IP que é comparada com a entrada da tabela de roteamento. Isso permite efetivamente que o endereço de rede e de sub-rede ocupe qualquer porção de um endereço IP, proporcionando mais flexibilidade do que as classes A, B e C. Daí, o nome roteamento entre domínios *sem classes*. Mais uma vez, essas alterações nos roteadores são feitas paulatinamente, de modo que alguns roteadores executam o esquema CIDR e outros usam os antigos algoritmos baseados em classe.

Isso funciona porque os novos intervalos de endereços são sempre alocados em módulo de 256, portanto cada novo intervalo é sempre um múltiplo inteiro de uma rede classe C. Por outro lado, algumas sub-redes também fazem uso de CIDR para subdividir o intervalo de endereços em uma única rede de classe A, B ou C. Se um conjunto de sub-redes estiver conectado com o resto do mundo inteiramente através de roteadores CIDR, então os intervalos de endereços IP, usados dentro desse conjunto, poderão ser alocados em blocos de diferentes tamanhos. Isso é feito empregando-se máscaras de tamanho variável.

Por exemplo, um espaço de endereços de classe C pode ser subdividido em 32 grupos de 8 endereços. A Figura 3.10 contém um exemplo do uso do mecanismo CIDR para dividir a sub-rede 138.37.95 (classe C) em vários grupos de oito endereços que são roteados separadamente. Os grupos são designados pelas notações 138.37.95.232/29, 138.37.95.248/29 etc. A parte /29 desses endereços indica uma máscara binária de 32 bits composta pelos 29 bits mais significativos em 1 e os três últimos em zero.

Endereços não registrados e NAT (*Network Address Translation*) ◊ Nem todos os computadores e dispositivos que acessam a Internet precisam receber endereços IP globalmente exclusivos. Os computadores que estão ligados a uma rede local e acessam a Internet por meio de um roteador com suporte a NAT podem usá-lo para enviar e receber mensagens UDP e TCP. A Figura 3.18 ilustra uma rede doméstica típica, com computadores e outros dispositivos de rede ligados à Internet por meio de um roteador capaz de realizar NAT. A rede inclui computadores que estão diretamente conectados no roteador por meio de uma conexão Ethernet cabeada, assim como outros equipamentos que estão conectados por meio de um ponto de acesso WiFi. Para ser genérico, são mostrados alguns dispositivos compatíveis com o padrão Bluetooth, mas eles não estão conectados no roteador e, assim, não podem acessar a Internet diretamente. A rede doméstica recebeu um único endereço IP registrado (83.215.152.95) de seu provedor de serviços de Internet. A estratégia descrita aqui é conveniente para qualquer organização que queira conectar na Internet computadores sem endereços IP registrados.

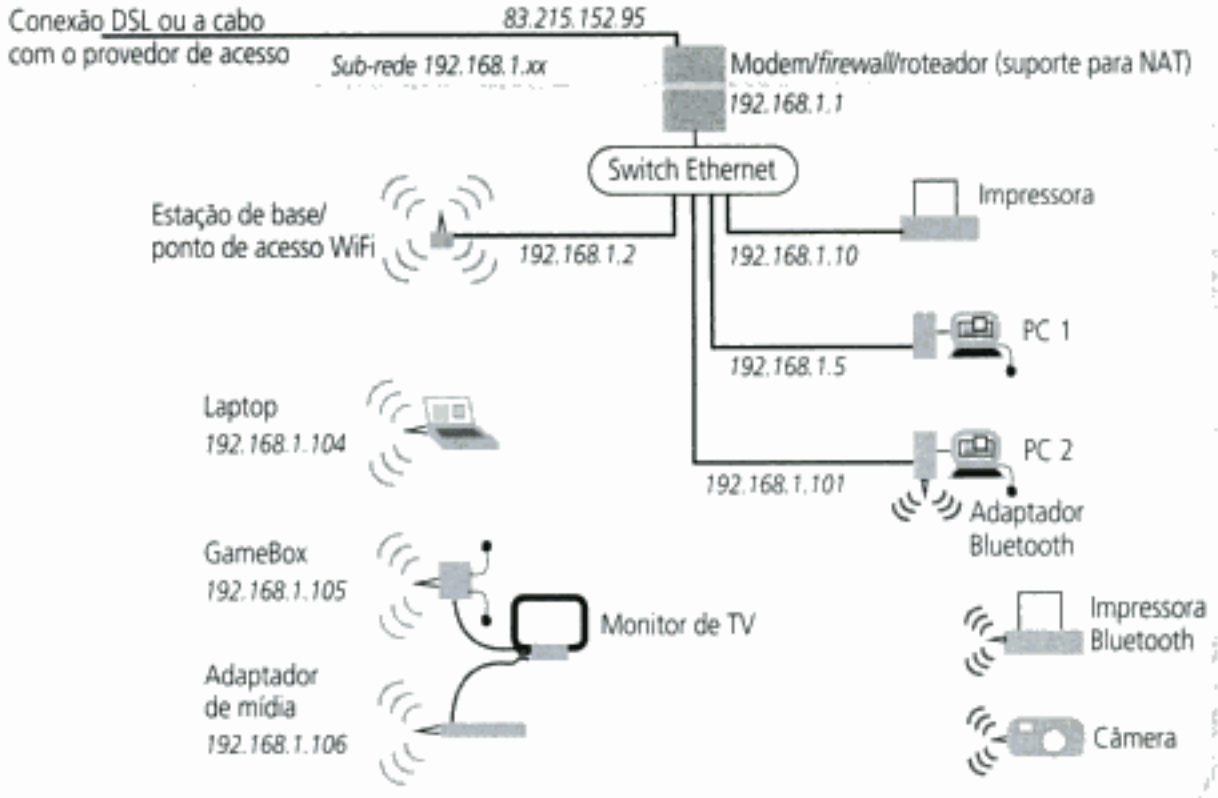


Figura 3.18 **Uma rede doméstica típica baseada em NAT.**

Todos os dispositivos da rede doméstica com conexão a Internet receberam endereços IP não registrados na sub-rede classe C 192.168.1.x. A maior parte dos computadores e dispositivos internos recebem endereços IP individuais, dinamicamente, por um serviço DHCP (*Dynamic Host Configuration Protocol*) executado no roteador. Em nossa ilustração, os números acima de 192.168.1.100 são fornecidos automaticamente pelo serviço DHCP e os endereços menores (como o PC 1) foram alocados manualmente, por um motivo explicado posteriormente nesta subseção. Embora todos esses endereços fiquem completamente ocultos do restante da Internet pelo roteador NAT, é convenção usar um intervalo de endereços de um de três blocos de endereços (10.z.y.x, 172.16.y.x ou 192.168.y.x) reservados pelo IANA para redes privadas, também denominadas de não registradas, não roteadas, ou ainda, falsas.

O NAT está descrito no RFC 1631 [Egevang e Francis 1994] e foi ampliado no RFC 2663 [Srisuresh e Holdrege 1999]. Os roteadores com suporte a NAT mantêm uma tabela de mapeamento de endereços, e emprega os campos de número de porta de origem e de destino das mensagens UDP e TCP para atribuir a cada mensagem de resposta recebida o computador interno que enviou a mensagem de solicitação correspondente. Note que a porta de origem fornecida em uma mensagem de solicitação é sempre usada como porta de destino na mensagem de resposta correspondente.

A variante mais comumente usada em NAT funciona da seguinte forma:

- Quando um computador da rede interna envia uma mensagem UDP ou TCP para um computador fora dela, o roteador recebe a mensagem e salva o endereço IP e o número de porta da origem em uma entrada disponível em sua tabela de mapeamento de endereços.
- O roteador substitui o endereço IP de origem pelo seu endereço IP, e a porta de origem por um número de porta virtual que indexa a entrada da tabela que contém a informação de endereço do computador remetente.
- O pacote com o endereço de origem e o número de porta modificados é então encaminhado para seu destino pelo roteador. Agora, a tabela de mapeamento de endereços contém uma associação do número de porta virtual para o endereço IP e número de porta internos reais para todas as mensagens enviadas pelos computadores da rede interna.

- Quando o roteador recebe uma mensagem UDP ou TCP de um computador externo, ele usa o número de porta de destino presente na mensagem para acessar uma entrada na tabela de mapeamento de endereços. Ele substitui o endereço de destino e a porta de destino, presentes na mensagem recebida, por aqueles armazenados na entrada, e encaminha a mensagem modificada para o computador interno identificado pelo endereço de destino.

O roteador mantém o mapeamento de porta enquanto ela estiver em uso. Um temporizador é zerado sempre que o roteador acessa uma entrada na tabela de mapeamento. Se a entrada não for acessada novamente antes que um certo período de tempo expire, a entrada é removida da tabela.

O esquema descrito anteriormente trata satisfatoriamente dos modos mais comuns de comunicação para computadores que possuem endereços IP não registrados que atuam como clientes para serviços externos, como servidores web. Mas isso não os permite atuar como servidores atendendo solicitações provenientes de clientes externos. Para tratar desse caso, os roteadores NAT podem ser manualmente configurados para encaminhar todos os pedidos recebidos em determinada porta para um computador interno em particular. Os computadores que atuam como servidores devem manter o mesmo endereço IP interno e isso é obtido por meio da configuração manual de seus endereços (como foi feito para PC 1). Essa solução para o problema do fornecimento de acesso externo a serviços é satisfatória, desde que não haja nenhum requisito para que mais de um computador interno ofereça um serviço em qualquer porta.

O NAT foi introduzido como uma solução a curto prazo para o problema da alocação de endereços IP para computadores pessoais e domésticos. Ele tem permitido que a expansão do uso da Internet ocorra de forma muito maior do que foi originalmente antecipado, mas também impõe algumas limitações, como a exemplificada anteriormente. O IPv6 deve ser visto como o próximo passo, permitindo participação total na Internet de todos os computadores e dispositivos portáteis.

3.4.4 IPv6

Uma solução permanente para as limitações do endereçamento do IPv4 foi investigada e isso levou ao desenvolvimento e à adoção de uma nova versão do protocolo IP com endereços substancialmente maiores. Já em 1990, o IETF percebeu os problemas em potencial provenientes dos endereços de 32 bits do IPv4 e iniciou um projeto para desenvolver uma nova versão do protocolo IP. O IPv6 foi adotado pelo IETF em 1994 e foi também recomendada uma estratégia de migração.

A Figura 3.19 mostra o formato do cabeçalho IPv6. Não nos propomos a abordar sua construção em detalhes aqui. Indicamos aos leitores os textos Tanenbaum [2003], Stallings [1998a] e Huitema [1998] para estudar sobre o processo de projeto e os planos de implementação do IPv6. Aqui, destacaremos os principais avanços dados pelo IPv6.

Espaço de endereçamento: os endereços do IPv6 têm 128 bits (16 bytes). Isso proporciona um número astronômico de entidades endereçáveis: 2^{128} ou, aproximadamente, 3×10^{38}. Tanenbaum

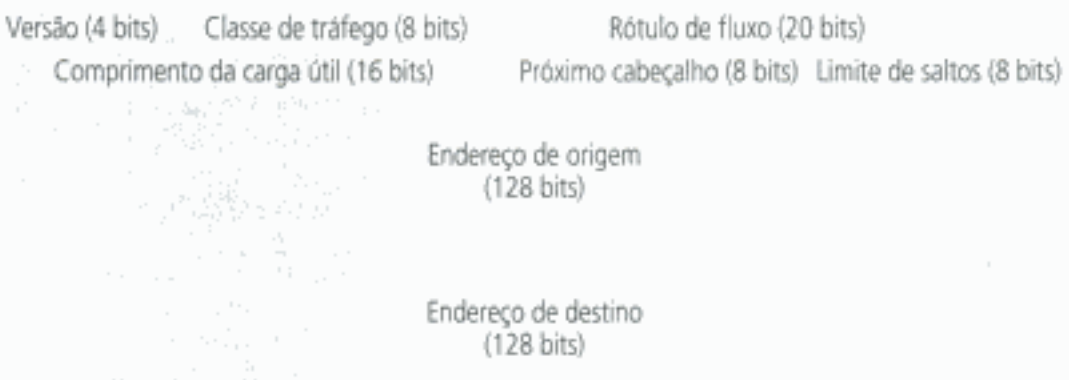


Figura 3.19 Formato do cabeçalho do IPv6 (obrigatório).

calcula que isso é suficiente para fornecer 7×10^{23} endereços IP por metro quadrado da superfície inteira da Terra. De forma mais conservadora, Huitema fez um cálculo pressupondo que os endereços IP são alocados de modo tão ineficiente como os números de telefone e obteve o valor de 1000 endereços IP por metro quadrado da superfície da Terra (terra e água).

O espaço de endereçamento do IPv6 é particionado. Não detalharemos aqui esse particionamento, mas mesmo as menores partições (uma das quais conterá o intervalo de endereços IPv4 inteiro, com mapeamento de um para um) são maiores do que o espaço total do IPv4. Muitas partições (representando 72% do total) são reservadas para propósitos ainda indefinidos. Duas partições grandes (cada uma compreendendo 1/8 do espaço de endereços) são alocadas para propósitos gerais e serão atribuídas aos nós de rede. Uma delas é baseada na localização geográfica dos nós e a outra, de acordo com uma localização organizacional. Isso possibilita duas estratégias alternativas para agregar endereços com vistas ao roteamento – o futuro dirá qual se mostrará mais eficiente ou popular.

Desempenho no roteamento: o processamento efetuado em cada nó para executar o roteamento IPv6 é reduzido em relação ao IPv4. Nenhuma soma de verificação é aplicada ao conteúdo do pacote (carga útil) e nenhuma fragmentação pode ocorrer, uma vez que um pacote tenha iniciado sua jornada. O primeiro é considerado aceitável, pois os erros podem ser detectados em níveis mais altos (o protocolo TCP inclui uma soma de verificação de conteúdo), e o último é obtido por meio do suporte de um mecanismo para determinação do menor MTU, antes que um pacote seja transmitido.

Suporte a tempo real e outros serviços especiais: os campos *classe de tráfego* e *rótulo de fluxo* estão relacionados a isso. Fluxos multimídia e outras seqüências de elementos de dados em tempo real podem ser transmitidos em um fluxo identificado. Os primeiros seis bits do campo *classe de tráfego* podem ser usados em conjunto, ou independentemente, com o *rótulo de fluxo* para permitir que pacotes específicos sejam manipulados mais rapidamente ou com confiabilidade maior do que outros. Os valores de classe de tráfego de 0 a 8 servem para transmissões que podem ser mais lentas sem causar efeitos não desejados para as aplicações. Os outros valores são reservados para pacotes cuja distribuição é dependente do tempo. Tais pacotes devem ser prontamente encaminhados ou eliminados – sua posterior distribuição não tem significado.

Os rótulos de fluxo permitem que recursos sejam reservados para atender requisitos de sincronização de fluxos de dados em tempo real específicos, como transmissões de áudio e vídeo ao vivo. O Capítulo 17 discutirá esses requisitos e os métodos para a alocação de recursos a eles. É claro que os roteadores e os enlaces de transmissão na Internet possuem limitações físicas e o conceito de reservá-los para usuários e aplicações específicas não foram considerados anteriormente. O uso dessas facilidades do IPv6 dependerá de maiores aprimoramentos na infra-estrutura e no desenvolvimento de métodos convenientes para ordenar e arbitrar a alocação de recursos.

Evolução futura: o segredo para a evolução futura é o campo *próximo cabeçalho*. Se for diferente de zero, ele define o tipo de um cabeçalho de extensão que é incluído no pacote. Atualmente, existem seis tipos de cabeçalho de extensão que fornecem dados adicionais para serviços especiais: informações para roteadores, definição de rota, tratamento de fragmentos, autenticação, criptografia e informações de tratamento no destino. Cada tipo de cabeçalho de extensão tem um tamanho específico e um formato definido. Novos tipos de cabeçalho de extensão poderão ser definidos quando surgirem novos requisitos de serviços. Um cabeçalho de extensão, se presente, vem após o cabeçalho obrigatório, precedendo a área de dados, e inclui um campo *próximo cabeçalho*, permitindo que vários cabeçalhos de extensão sejam encadeados.

Difusão seletiva (multicast) e não-seletiva (anycast): tanto o IPv4 como o IPv6 incluem suporte para a transmissão de datagramas/pacotes IP para vários *hosts* usando um único endereço (que está no intervalo reservado para isso). Os roteadores IP são então responsáveis pelo roteamento do pacote para todos os *hosts* que se inscreveram no grupo identificado pelo endereço relevante. Mais detalhes sobre comunicação *multicast* IP podem ser encontrados na Seção 4.5.1. Além disso, o IPv6 suporta um novo modo de transmissão, chamado *difusão não-seletiva (anycast)*. Esse serviço distribui um pacote para pelo menos um dos *hosts* que está inscrito no endereço de grupo relevante.

Segurança: até agora, os aplicativos Internet que exigem transmissão de dados autenticada, ou com privacidade, contam com o uso de técnicas de criptografia na camada de aplicação. O princípio fim-a-fim apóia a visão de que esse é o lugar certo para isso. Se a segurança fosse implementada em nível IP, os usuários e desenvolvedores de aplicações dependeriam do código implementado em cada roteador percorrido e deveriam confiar neles, e em outros nós intermediários, para a manipulação das chaves criptográficas.

A vantagem de implementar a segurança em nível IP é que ela pode ser aplicada sem a necessidade das aplicações serem implementadas com requisitos de segurança. Por exemplo, os administradores de rede podem configurá-la em um *firewall* e aplicá-la uniformemente a toda comunicação externa, sem incorrer no custo da criptografia para as comunicações realizadas na rede interna. Os roteadores também podem explorar um mecanismo de segurança em nível IP para dar garantias sobre as mensagens de atualização de tabelas de roteamento que trocam entre si.

A segurança no IPv6 é implementada por meio dos tipos de cabeçalho de extensão de *autenticação* e de *criptografia*. Eles implementam recursos equivalentes ao conceito de canal seguro apresentado na Seção 2.3.3. A carga útil (área de dados) é crifrada e/ou assinada digitalmente, conforme for exigido. Recursos de segurança semelhantes também estão disponíveis no IPv4, usando túneis IP entre roteadores e *hosts* que implementam a especificação IPSec (veja o RFC 2411 [Thayer 1998]).

Migração do IPv4 ◊ As conseqüências para a infra-estrutura existente na Internet de uma alteração em seu protocolo básico são profundas. O IP é processado na pilha de protocolos TCP/IP em cada *host* e no software de cada roteador. Os endereços IP são manipulados em muitos programas aplicativos e utilitários. Isso exige que todos sofram atualizações para suportar a nova versão de IP. Mas a alteração é inevitável por causa do futuro esgotamento do espaço de endereçamento fornecido pelo IPv4 e o grupo de trabalho da IETF, responsável pelo IPv6, definiu uma estratégia de migração – basicamente, ela envolve a implementação de "ilhas" de roteadores e *hosts* IPv6 se comunicando por meio de túneis com outras ilhas IPv6 e sua gradual agregação em ilhas maiores.

Conforme mencionamos, os roteadores e *hosts* IPv6 não devem ter nenhuma dificuldade no tratamento de tráfego misto, pois o espaço de endereçamento do IPv4 é incorporado ao espaço do IPv6. Todos os principais sistemas operacionais (Windows XP, Mac OS X, Linux e outras variantes do Unix) já incluem implementações de soquetes UDP e TCP (conforme descrito no Capítulo 4) sobre IPv6, permitindo que os aplicativos sejam migrados com uma atualização simples.

A teoria dessa estratégia é tecnicamente sólida, mas o andamento da implementação tem sido muito lento, talvez porque o CIDR e o NAT têm aliviado a pressão mais do que havia sido antecipado. Esse quadro começou a mudar com os mercados de telefonia móvel e de equipamentos portáteis. Todos esses dispositivos provavelmente serão habilitados para a Internet em um futuro próximo, e eles não podem ser facilmente ocultados atrás de roteadores com suporte a NAT. Por exemplo, estima-se que mais de um bilhão de dispositivos IP sejam distribuídos na Índia e na China em 2014. Apenas o IPv6 pode tratar de necessidades como essa.

3.4.5 MobileIP

Os computadores móveis, como os laptops e palmtops, são conectados na Internet a partir de diferentes locais, à medida que migram. O dono de um laptop, enquanto estiver em seu escritório, pode acessar a Internet através de uma conexão a uma rede local Ethernet, assim como também pode fazê-lo no trânsito, dentro de seu carro, a partir de um modem para telefone celular. É possível ainda que esse laptop seja conectado em uma rede local Ethernet em um outro local qualquer. Esse usuário pode desejar ter acesso a serviços como e-mail e web em cada um desses locais.

O simples acesso aos serviços não exige que um computador móvel mantenha um único endereço, e ele pode adquirir um novo endereço IP em cada local diferente; esse é o objetivo do DHCP, que permite a um computador recentemente conectado adquirir dinamicamente um endereço IP do intervalo de endereços da sub-rede local e descobrir os endereços de recursos locais, como um servidor de DNS. Ele também precisará descobrir quais serviços locais (como impressão, e-mail, etc.) estão disponíveis em cada instalação que visitar. Os serviços de descoberta são um tipo de serviço de atribuição de nomes que auxiliam nessa tarefa; eles serão descritos no Capítulo 16 (Seção 16.2).

No laptop podem existir arquivos, ou outros tipos de recursos, necessários para outros usuários, ou pode estar sendo executado um aplicativo distribuído, como um serviço que recebe notificações sobre o comportamento de ações da bolsa que atinjam um limite predefinido. Em situações como essa, um computador móvel necessita se manter acessível para clientes e aplicativos mesmo se locomovendo entre redes locais e redes sem fio. Para isso ele deve manter um único número IP, porém o roteamento IP é baseado na sub-rede. As sub-redes estão em locais fixos e o roteamento é dependente da localização na rede.

O MobileIP é uma solução para este último problema. A solução é implementada de forma transparente, de modo que a comunicação de IP continue normalmente, mesmo quando um *host* móvel se movimenta entre sub-redes de diferentes locais. Ela é baseada na alocação permanente de um endereço IP "convencional" para cada *host* móvel, em uma sub-rede de seu domínio doméstico.

Quando o *host* móvel está conectado em sua rede de domicílio, os pacotes são direcionados para ele da maneira habitual. Quando ele está conectado na Internet, em qualquer lugar, dois processos assumem a responsabilidade pelo redirecionamento. São eles: um *agente doméstico* (AD) e um *agente estrangeiro* (AE). Esses processos são executados em computadores fixos na rede doméstica e no local corrente do *host* móvel.

O AD é responsável por manter o conhecimento atualizado da localização corrente do *host* (o endereço IP por meio do qual ele pode ser encontrado). Isso é feito com a ajuda do próprio *host* móvel que quando deixa sua rede doméstica, informa ao AD e este anota a ausência. Durante a ausência, o AD se comporta como um *proxy*; para isso, ele diz aos roteadores locais que cancelem todos os registros colocados em cache relacionados ao endereço IP do *host* móvel. Enquanto estiver atuando como *proxy*, o AD responde as requisições ARP relativas ao endereço IP do *host* móvel, fornecendo seu próprio endereço físico de rede como endereço físico de rede do *host* móvel.

Quando o *host* móvel se instala em um novo local, ele informa ao AE sobre sua chegada. O AE aloca para ele um "endereço aos cuidados de" (COA – *care-of-address*) – que é um novo endereço IP, temporário, na sub-rede local. Então, o AE entra em contato com o AD e fornece o endereço IP doméstico do *host* móvel e o endereço aos cuidados de que foi alocado a ele.

A Figura 3.20 ilustra o mecanismo de roteamento do MobileIP. Quando um datagrama IP, endereçado para o endereço doméstico do *host* móvel, é recebido na rede doméstica, ele é direcionado para o AD. Então, o AD encapsula o datagrama IP em um pacote MobileIP e o envia para o AE. O AE desempacota o datagrama IP original e o distribui para o *host* móvel por meio da rede local na qual está atualmente ligado. Note que o método pelo qual o AD e o AE redirecionam o datagrama original até seu destino é um exemplo da técnica de túneis descrita na Seção 3.3.7.

O AD também envia o endereço aos cuidados de do *host* móvel para o remetente original. Se o remetente for compatível com o MobileIP, ele notará o novo endereço e o utilizará nas comunicação subseqüentes com esse *host* móvel, evitando as sobrecargas do redirecionamento por meio do AD. Caso não seja compatível, ele ignorará a mudança de endereço e a comunicação continuará a ser redirecionada por meio do AD.

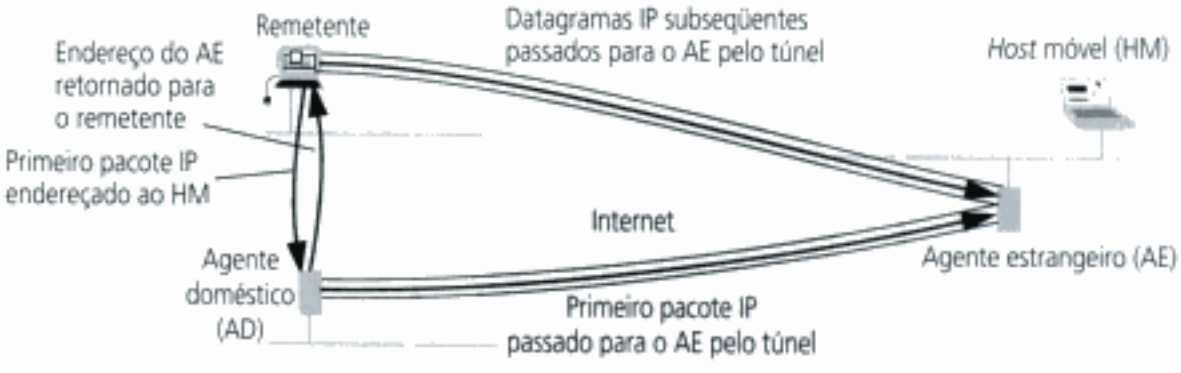


Figura 3.20 O mecanismo de roteamento MobileIP.

A solução MobileIP é eficaz, mas dificilmente eficiente. Uma solução que tratasse os *hosts* móveis como cidadão de primeira classe seria preferível, permitindo-os perambular sem aviso e direcionar datagramas a eles sem nenhuma utilização de túnel ou redirecionamento. Devemos notar que esse feito aparentemente difícil é exatamente o que ocorre na telefonia celular – os celulares não mudam de número ao se moverem entre as células ou mesmo entre países. Em vez disso, eles simplesmente notificam, de tempos em tempos, a estação de base de telefonia celular local sobre sua presença.

3.4.6 TCP e UDP

Os protocolos TCP e UDP disponibilizam os recursos de comunicação via Internet de uma forma bastante prática para programas aplicativos. No entanto, os desenvolvedores de aplicativos talvez queiram outros tipos de serviço de transporte, por exemplo, para fornecer garantias ou segurança em tempo real, porém tais serviços geralmente exigem mais suporte na camada de rede do que o fornecido pelo IPv4. Os protocolos TCP e UDP podem ser considerados um reflexo exato, em nível da programação de aplicativos, dos recursos de comunicação que o IPv4 tem a oferecer. O IPv6 é outra história; ele certamente continuará a suportar TCP e UDP, mas incluirá recursos que não podem ser convenientemente usados por meio de TCP e UDP. As capacidades do IPv6 serão úteis para se introduzir tipos adicionais de serviço de transporte quando a sua utilização for suficientemente ampla para justificar tal desenvolvimento.

O Capítulo 4 descreverá as características do TCP e do UDP, do ponto de vista dos programadores de aplicativos distribuídos. Aqui, precisamos ser muito sucintos, descrevendo apenas a funcionalidade que eles acrescentam ao IP.

Uso de portas ◊ A primeira característica a notar é que, enquanto o IP suporta comunicação entre pares de computadores (identificados pelos seus endereços IP), o TCP e o UDP, como protocolos de transporte, fornecem comunicação de processo para processo. Isso é feito pelo uso de portas. Os *números de porta* são utilizados para endereçar mensagens para processos em um computador em particular, e são válidos somente nesse computador. Um número de porta é um valor inteiro de 16 bits. Uma vez que um datagrama IP tenha sido entregue para o *host* destino, o software da camada TCP (ou UDP) o envia para um processo específico associado a uma dada porta nesse *host*.

Características do UDP ◊ O UDP é quase uma réplica, em nível de transporte, do IP. Um datagrama UDP é encapsulado dentro de um datagrama IP. Ele tem um cabeçalho curto que inclui os números de porta de origem e destino (os endereços de *host* correspondentes estão presentes no cabeçalho IP), um campo de comprimento e uma soma de verificação. O UDP não oferece nenhuma garantia de entrega. Já citamos que datagramas IP podem ser perdidos por causa de congestionamento ou erro na rede. O UDP não acrescenta nenhum mecanismo de confiabilidade adicional, exceto a soma de verificação, que é opcional. Se o campo de soma de verificação for diferente de zero, o *host* destino calculará um valor de verificação a partir do conteúdo do datagrama e o comparará com a soma de verificação recebida; se eles não corresponderem o datagrama é eliminado.

Assim, o UDP fornece uma maneira de transmitir mensagens de até 64 Kbytes de tamanho (o pacote máximo permitido pelo IP) entre pares de processos (ou de um processo para vários, no caso de datagramas enviados para endereços IP *multicast*) com custos adicionais ou atrasos mínimos se comparados àqueles esperados para transmissão IP. Ele não acarreta nenhum custo de configuração e não exige nenhuma mensagem de confirmação. Seu uso, no entanto, está restrito aos aplicativos e serviços que não exigem entrega confiável de uma ou várias mensagens.

Características do TCP ◊ O TCP fornece um serviço de transporte muito mais sofisticado. Ele oferece entrega confiável de seqüências de bytes arbitrariamente longas por meio de uma abstração denominada de fluxo de dados (*data stream*). A garantia da confiabilidade implica na recepção, por parte do processo destino, de todos os dados enviados, na mesma ordem. O TCP é orientado a conexão. Antes que qualquer dado seja transferido, o processo remetente e o processo destino devem estabelecer um canal de comunicação bidirecional: a conexão. A conexão é simplesmente um acordo, entre os processos participantes, que viabiliza uma transmissão de dados confiável. Os nós intermediários, como os roteadores, não têm conhecimento algum das conexões TCP, e nem todos os datagramas IP que transferem os dados em uma transmissão TCP seguem necessariamente a mesma rota.

A camada TCP inclui mecanismos adicionais (implementados sobre IP) para satisfazer as garantias de confiabilidade. São eles:

Sequenciamento: um processo remetente divide o fluxo TCP em uma seqüência de segmentos de dados e os transmite como datagramas IP. Um número de seqüência é anexado a cada segmento TCP fornecendo a ordem em que o primeiro byte desse segmento aparece no fluxo TCP. O destino usa os números de seqüência para ordenar os segmentos recebidos, antes de passá-los para o fluxo de entrada de dados do processo destino. Nenhum segmento pode ser colocado no fluxo de entrada até que todos os segmentos de número menor tenham sido recebidos e inseridos no fluxo; portanto, os segmentos que chegam fora de ordem devem ser mantidos em um buffer, até que seus predecessores cheguem.

Controle de fluxo: o remetente toma o cuidado de não sobrecarregar o destino ou os nós intermediários. Isso é obtido por meio de um sistema de confirmação de segmentos. Quando um destino recebe um segmento com êxito, ele grava seu número de seqüência. De tempos em tempos, o destino envia uma confirmação para o remetente, fornecendo o número de seqüência mais alto em seu fluxo de entrada, junto com um *tamanho de janela*. Se houver um fluxo de dados no sentido inverso, as confirmações são transportadas nos próprios segmentos de dados; caso contrário, elas são enviadas em segmentos específicos de confirmação. O campo tamanho de janela no segmento de confirmação especifica o volume de dados que o remetente pode enviar antes de receber a próxima confirmação.

Quando uma conexão TCP é usada para comunicação com um programa interativo remoto, os dados podem ser produzidos em pequenos volumes, mas na forma de picos. Por exemplo, a entrada do teclado pode resultar em apenas alguns caracteres por segundo, mas os caracteres devem ser enviados de maneira suficientemente rápida para que o usuário veja os resultados de sua digitação. Isso é tratado pela configuração de um tempo de espera T no buffer local – normalmente, 0,5 segundos. Com esse esquema simples, um segmento é enviado para o receptor quando os dados estão esperando no buffer de saída há T segundos ou quando o conteúdo do buffer atinge o limite de MTU. Esse esquema de bufferização não pode acrescentar mais do que T segundos no atraso interativo. Nagle [Nagle 1984] descreveu outro algoritmo, usado em muitas implementações de TCP, que produz menos tráfego e é mais eficiente para alguns aplicativos interativos. A maioria das implementações de TCP pode ser configurada, permitindo que os aplicativos alterem o valor de T ou selecionem um de vários algoritmos de bufferização.

Por causa da falta de confiabilidade das redes sem fio, e da resultante perda freqüente de pacotes, esses mecanismos de controle de fluxo não são apropriados para a comunicação sem fio. Esse é um dos motivos para a adoção de um mecanismo de transporte diferente na família de protocolos WAP, para comunicação móvel de longa distância. Mas a implementação de TCP para redes sem fio também é importante e foram propostas modificações no mecanismo TCP com esse objetivo [Balakrishnan *et al.* 1995, 1996]. A idéia é implementar um módulo de suporte a TCP no ponto de acesso sem fio (o *gateway* entre redes com e sem fio). O módulo de suporte espiona os segmentos TCP na rede sem fio, retransmitindo os segmentos que não são confirmados rapidamente pelo receptor móvel e solicitando retransmissões dos segmentos recebidos quando são notadas lacunas nos números de seqüência.

Retransmissão: o remetente registra os números de seqüência dos segmentos que envia. Quando recebe uma confirmação, ele nota que os segmentos foram recebidos com sucesso e pode então excluí-los de seus buffers de saída. Se qualquer segmento não for confirmado dentro de um tempo limite especificado, o remetente o retransmitirá.

Uso de buffers: o buffer de entrada do receptor é usado para balancear o fluxo entre o remetente e o destino. Se o processo destino executar operações de recepção mais lentamente do que o remetente faz operações de envio, o volume de dados no buffer crescerá. Normalmente, os dados são extraídos do buffer, antes que ele se torne cheio, mas o buffer pode esgotar e quando isso acontece, os segmentos recebidos são simplesmente eliminados, sem registro de sua chegada. Portanto, sua chegada não é confirmada e o remetente é obrigado a retransmiti-los.

Soma de verificação: cada segmento mantém uma soma de verificação abrangendo o cabeçalho e os dados do segmento. Se um segmento recebido não corresponde à sua soma de verificação, então ele é eliminado.

3.4.7 Sistema de nomes de domínio

O projeto e a implementação do DNS (*Domain Name System*) serão descritos em detalhes no Capítulo 9; fornecemos aqui um breve panorama para completar nossa discussão sobre os protocolos Internet. A Internet suporta um esquema para o uso de nomes simbólicos para *hosts* e redes, como *binkley.cs.mcgill.ca* ou *essex.ac.uk*. Os nomes são organizados de acordo com uma hierarquia de atribuição na forma de uma árvore. As entidades nomeadas são chamadas de *domínios* e os nomes simbólicos são chamados de *nomes de domínio*. Os domínios são organizados, hierarquicamente, de forma a refletir sua estrutura organizacional. A hierarquia de atribuição de nomes é totalmente independente da topologia física das redes que constituem a Internet. Os nomes de domínio são convenientes para os seres humanos, mas precisam ser transformados em endereços IP antes de serem usados como identificadores de uma comunicação. Isso é responsabilidade de um serviço específico, o DNS. Os aplicativos enviam requisições para o serviço de DNS, para converter os nomes de domínio fornecidos pelos usuários em endereços IP.

O DNS é implementado como um processo servidor que pode ser executado nos computadores *host* de qualquer parte da Internet. Existem pelo menos dois servidores de DNS em cada domínio e, freqüentemente, mais. Os servidores de cada domínio contêm um mapa parcial da hierarquia de nomes que está abaixo de seu domínio. Eles devem conter pelo menos a parte que consiste em todos os nomes de domínio e *host* dentro de seus domínios, mas freqüentemente contêm uma parte maior. Os servidores de DNS tratam as requisições de resolução de nomes de domínio que estão fora de sua hierarquia, emitindo requisições para os servidores de DNS nos domínios relevantes, prosseguindo recursivamente, da direita para a esquerda, transformando o nome em segmentos. A transformação resultante é então colocada na cache do servidor que está tratando a requisição original, para que futuras requisições de resolução de nomes que se refiram ao mesmo domínio sejam solucionadas sem referência a outros servidores. O DNS não funcionaria sem o uso extensivo da cache, pois os servidores de nomes-raiz seriam consultados em praticamente todos os casos, criando um gargalo no acesso ao serviço.

3.4.8 *Firewalls*

Quase todas as organizações precisam de conectividade à Internet para fornecer serviços para seus clientes, para usuários externos e para permitir que seus usuários internos acessem informações e serviços. Os computadores da maioria das organizações são bastante diversificados, executando uma variedade de sistemas operacionais e softwares aplicativos. Para piorar a situação, alguns aplicativos podem incluir tecnologias de segurança de ponta, porém, a maior parte deles tem pouca ou nenhuma capacidade para garantir a integridade dos dados recebidos ou, quando necessário, a privacidade no envio dos dados. Em resumo, em uma intranet com muitos computadores e uma ampla variedade de software, é inevitável que algumas partes do sistema tenham vulnerabilidades que o exponham a ataques contra a segurança. As formas de ataque serão melhor detalhadas no Capítulo 7.

O objetivo de um *firewall* é monitorar e controlar toda a comunicação para dentro e para fora de uma intranet. Um *firewall* é implementado por um conjunto de processos que atuam como um *gateway* para uma intranet (Figura 3.21 (a)), aplicando uma política de segurança determinada pela organização.

O objetivo de uma política de segurança com *firewall* pode incluir parte ou tudo do que segue:

Controle de serviço: para determinar quais serviços nos *hosts* internos estão disponíveis para acesso externo e para rejeitar quaisquer pedidos a outros serviços. Os pedidos de serviço enviados e as respostas também podem ser controlados. Essas ações de filtragem podem ser baseadas no conteúdo de datagramas IP e dos cabeçalhos TCP e UDP que eles contêm. Por exemplo, os pedidos de HTTP recebidos podem ser rejeitados, a não ser que sejam direcionados para um *host* servidor web considerado como "oficial".

Controle de comportamento: para evitar comportamentos que violem as políticas da organização, seja anti-social ou que não tenha nenhum propósito legítimo discernível e, portanto, seja suspeito de fazer parte de um ataque. Algumas dessas ações de filtragem podem ser aplicadas nas camadas de rede (IP) ou de transporte (TCP/UDP), mas outras podem exigir interpretação das mensagens

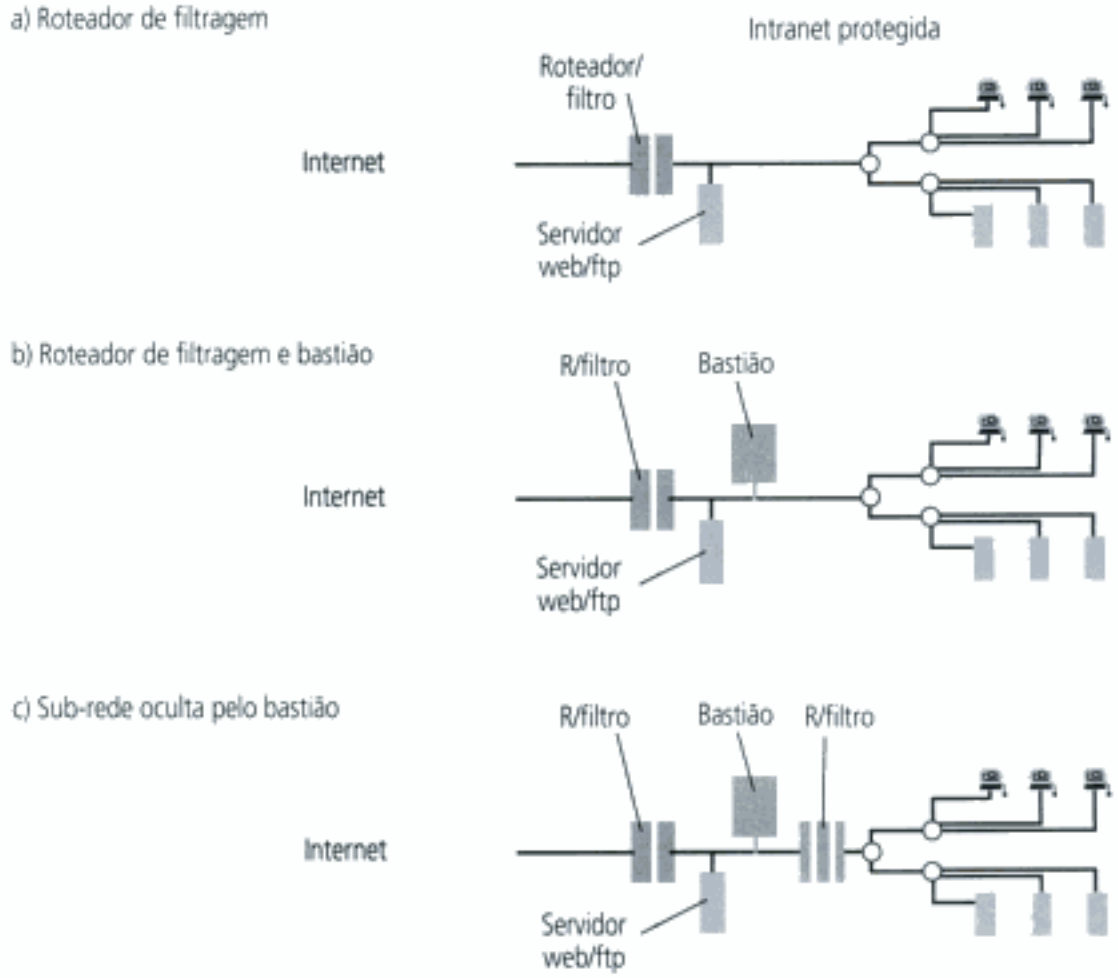


Figura 3.21 Configurações de *firewall*.

da camada de aplicação. Por exemplo, a filtragem de ataques por *spam* de e-mail pode exigir um exame do endereço de e-mail do remetente nos cabeçalhos da mensagem ou mesmo no seu conteúdo.

Controle de usuário: talvez a organização queira fazer discriminação entre seus usuários, permitindo que alguns acessem serviços externos, mas proibindo que outros o façam. Um exemplo de controle de usuário, que talvez seja mais socialmente aceitável, é impedir, exceto para os usuários que sejam membros da equipe de administração da rede, a instalação de software para evitar infecção por vírus ou para manter padrões de software. Na verdade, esse exemplo em particular seria difícil de implementar sem limitar o uso da web por usuários normais.

Outro exemplo de controle de usuário é o gerenciamento de conexões *dial-up* e outras fornecidas externamente para os usuários. Se o *firewall* também é o *host* para conexões via modem, ele pode autenticar o usuário no momento da conexão e exigir o uso de um canal seguro para toda comunicação (para evitar intromissão, mascaramento e outros ataques na conexão externa). Esse é o objetivo da tecnologia de VPN (*Virtual Private Network*), descrita na próxima sub-seção.

A política precisa ser expressa em termos das operações a serem executadas pelos processos de filtragem nos vários níveis diferentes:

Filtragem de datagramas IP: esse é um processo de filtragem que examina individualmente os datagramas IP. Ele pode tomar decisões com base nos endereços de destino e de origem. O processo também pode examinar o campo *tipo de serviço* dos datagramas IP e interpretar o conteúdo dos pacotes com base nele. Por exemplo, o processo pode filtrar pacotes TCP de acordo com o número da porta para a qual eles foram endereçados, e como os serviços estão geralmente localizados

em portas conhecidas, isso permite que os pacotes sejam filtrados com base no serviço solicitado. Por exemplo, muitos sites proíbem o uso de servidores de NFS por clientes externos.

Por razões de desempenho, a filtragem de datagramas IP é normalmente realizada por um processo dentro do núcleo do sistema operacional de um roteador. Se forem usados vários *firewalls*, o primeiro pode marcar certos pacotes para um exame mais exaustivo por parte de um *firewall* posterior, permitindo que pacotes "limpos" prossigam. É possível filtrar com base nas seqüências de datagramas IP, por exemplo, para impedir o acesso a um servidor FTP antes que um *login* tenha sido efetuado.

Gateway TCP: um processo *gateway* TCP verifica todos os pedidos de conexão TCP e transmissões de segmento. Quando um *gateway* TCP é instalado, a configuração das conexões TCP pode ser controlada e a correção dos segmentos TCP pode ser verificada (alguns ataques de negação de serviço utilizam segmentos TCP mal formados para entrar em sistemas operacionais de máquinas clientes). Quando desejado, eles podem ser direcionados por meio de um *gateway* em nível de aplicativo para verificação de conteúdo.

Gateway em nível de aplicativo: um processo *gateway* em nível de aplicativo atua como *proxy* para um processo aplicativo. Por exemplo, pode-se desejar uma política que permita conexões Telnet com certos *hosts* externos para determinados usuários internos. Quando um usuário executa um programa Telnet em seu computador local, ele tenta estabelecer uma conexão TCP com um *host* remoto. O pedido é interceptado pelo *gateway* TCP. O *gateway* TCP inicia um processo *proxy Telnet* e a conexão TCP original é direcionada para ele. Se o *proxy* aprovar a operação Telnet (o usuário está autorizado a usar o *host* solicitado), ele estabelece outra conexão com o *host* solicitado e, então, retransmite todos os segmentos TCP nas duas direções. Um processo *proxy* pode ser empregado para outros serviços, como por exemplo, para FTP.

Um *firewall* normalmente é composto por vários processos trabalhando em diferentes camadas de protocolo. É comum empregar mais de um computador em tarefas de *firewall*, por motivos de desempenho e tolerância a falhas. Em todas as configurações descritas a seguir, e ilustradas na Figura 3.21, mostramos um servidor web público e um servidor FTP sem proteção. Eles contêm apenas informações que não exigem proteção contra acesso público e seu software servidor garante que apenas usuários internos autorizados possam atualizá-las.

A filtragem de datagramas IP normalmente é feita por um roteador – um computador com pelo menos dois endereços de rede, em redes IP distintas, que executa um processo RIP, um processo de filtragem de datagramas IP e um mínimo possível de outros processos. O roteador/filtro só deve executar software confiável, de forma que se garanta a execução de suas políticas de filtragem. Isso envolve certificar-se de que nenhum processo do tipo cavalo de Tróia possa ser executado nele, e que os softwares de filtragem e de roteamento não tenham sido modificados, nem falsificados. A Figura 3.21(a) mostra uma configuração de *firewall* simples que conta apenas com a filtragem de datagramas IP e emprega um único roteador para esse fim. A configuração de rede da Figura 3.10, por motivos de desempenho e confiabilidade, possui dois roteadores/filtros atuando como *firewall*. Ambos obedecem a uma mesma política de filtragem e o segundo não aumenta a segurança do sistema.

Quando são exigidos processos TCP e *gateway* em nível de aplicativo, normalmente eles são executados em um computador separado, o qual é conhecido como *bastião*. (O termo se origina da construção de castelos fortificados; trata-se de uma torre de vigia saliente, a partir da qual o castelo pode ser defendido ou os defensores podem negociar com os que desejam entrar.) Um computador bastião é um *host* localizado dentro da intranet, protegido por um roteador/filtro IP, que executa o protocolo TCP e *gateways* em nível de aplicativo (Figura 3.21(b)). Assim como o roteador/filtro, o bastião deve executar software confiável. Em uma intranet bem segura, todos os acessos a serviços externos devem ser feitos via *proxy*. Os leitores podem estar familiarizados com o uso de *proxies* para acesso à web. Eles são um exemplo do uso de *proxy* como *firewall*; freqüentemente, são construídos de maneira a integrar um servidor de cache web (descrito no Capítulo 2). Esse e outros processos *proxy* provavelmente exigirão processamento e recursos de armazenamento substanciais.

A segurança pode ser melhorada pelo emprego de dois roteadores/filtros em série, com o bastião e os servidores públicos localizados em uma sub-rede separada ligando os roteadores/filtros (Figura 3.21(c)). Essa configuração tem diversas vantagens para a segurança:

- Se a política aplicada pelo bastião for restrita, os endereços IP dos *hosts* da intranet não precisarão nem mesmo ser divulgados para o mundo exterior, e os endereços do mundo exterior não precisarão ser conhecidos dos computadores internos, pois toda comunicação externa passará pelos processos *proxies* que estão no bastião.
- Se o primeiro roteador/filtro for invadido ou comprometido, o segundo, que é invisível fora da intranet e, portanto, menos vulnerável, continuará a selecionar e a rejeitar datagramas IP indesejáveis.

Redes privadas virtuais ◊ As redes privadas virtuais (VPNs – *Virtual Private Networks*) ampliam o limite de proteção para além de uma intranet local através do uso de canais seguros protegidos por criptografia em nível IP. Na Seção 3.4.4, delineamos as extensões de segurança IP disponíveis no IPv6 e no IPv4 com túneis IPSec [Thayer 1998]. Elas são a base para a criação de VPNs. As VPNs podem ser usadas tanto por usuários externos individuais para proteger suas conexões como para implementar um ambiente seguro entre intranets localizadas em diferentes sites, usando enlaces Internet públicos.

Por exemplo, talvez um membro de uma equipe precise se conectar à intranet de sua empresa por meio de um provedor de serviços de Internet. Uma vez conectado, ele deve ter acesso aos mesmos recursos que um usuário que está dentro da rede protegida por *firewall*. Isso pode ser obtido se seu *host* local implementar segurança IP. Nesse caso, o *host* local compartilha uma ou mais chaves criptográficas com o *firewall* e elas são usadas para estabelecer um canal seguro no momento da conexão. Os mecanismos de canal seguro serão descritos em detalhes no Capítulo 7.

3.5 Estudos de caso: Ethernet, WiFi, Bluetooth e ATM

Até este ponto, discutimos os princípios envolvidos na construção de redes de computadores e descrevemos o IP, a "camada de rede virtual" da Internet. Para concluirmos o capítulo, descreveremos nesta seção os princípios e as implementações de três redes físicas.

No início dos anos 80, o IEEE (Institute of Electrical and Electronic Engineers), dos EUA, instituiu um comitê para especificar uma série de padrões para redes locais (o 802 Committee [IEEE 1990]) e seus subcomitês produziram uma série de especificações que se tornaram os principais padrões para redes locais. Na maioria dos casos, os padrões são baseados em outros já existentes no setor, que surgiram a partir de pesquisas feitas nos anos 70. Os subcomitês relevantes e os padrões publicados são dados na Figura 3.22.

Os vários padrões diferem entre si no desempenho, na eficiência, na confiabilidade e no custo, mas todos fornecem recursos para a interligação em rede com uma largura de banda relativamente alta em distâncias curtas e médias. O padrão Ethernet IEEE 802.3 domina o mercado das redes locais cabeadas e o descreveremos na Seção 3.5.1 como nosso representante da tecnologia de rede local

IEEE No.	*Nome*	*Título*	*Referência*
802.3	Ethernet	CSMA/CD Networks (Ethernet)	[IEEE 1985a]
802.4		Token Bus Networks	[IEEE 1985b]
802.5		Token Ring Networks	[IEEE 1985c]
802.6		Metropolitan Area Networks	[IEEE 1994]
802.11	WiFi	*Wireless* Local Area Networks	[IEEE 1999]
802.15.1	Bluetooth	*Wireless* Personal Area Networks	[IEEE 2002]
802.15.4	ZigBee	*Wireless* Sensor Networks	[IEEE 2003]
802.16	WiMAX	*Wireless* Metropolitan Area Networks	[IEEE 2004a]

Figura 3.22 Padrões IEEE 802.

cabeada. Embora estejam disponíveis implementações de Ethernet para várias larguras de banda, os princípios de operação são idênticos em todas elas.

O padrão *Token Ring* IEEE 802.5 foi um concorrente importante em grande parte dos anos 90, oferecendo vantagens em relação ao padrão Ethernet na eficiência e em seu suporte para garantia de largura de banda, mas agora desapareceu do mercado. Os leitores que estiverem interessados nessa tecnologia de rede local podem encontrar uma breve descrição no endereço www.cdk4.net/networking. A popularização dos *switches* Ethernet (em oposição aos hubs) tem permitido que as redes Ethernet sejam configuradas de maneira a oferecer garantias de largura de banda e latência (conforme discutido na Seção 3.5.1, sub-seção *Ethernet para aplicativos em tempo real e qualidade de serviço crítica*), e esse é um dos motivos do desaparecimento da tecnologia *token ring*.

O padrão *Token Bus* IEEE 802.4 foi desenvolvido para aplicações industriais com requisitos de tempo real. O padrão *Metropolitan Area Network* IEEE 802.6 cobre distâncias de até 50 km e é destinado para uso em redes que abrangem municípios e cidades.

O padrão *Wireless* LAN IEEE 802.11 surgiu bem mais tarde, mas sob o nome comercial de WiFi, mantém uma posição de destaque no mercado com produtos de muitos fornecedores, se encontrando na maioria dos dispositivos de computação móveis e portáteis. O padrão IEEE 802.11 é projetado para suportar comunicação em velocidades de até 54 Mbps, em distâncias de no máximo 150 m entre dispositivos equipados com transmissores/receptores sem fio simples. Descreveremos seus princípios de operação na Seção 3.5.2. Mais detalhes sobre as redes IEEE 802.11 podem ser encontrados em Crow *et al.* [1997] e em Kurose e Ross [2000].

O padrão *Wireless Personal Area Network* (Bluetooth) IEEE 802.15.1 foi baseado em uma tecnologia desenvolvida em 1999 pela Ericsson, para transportar voz e dados digitais, com largura de banda baixa, entre dispositivos como PDAs, telefones móveis e fones de ouvido, e subseqüentemente, em 2002, foi padronizado como o padrão IEEE 802.15.1. A Seção 3.5.3 contém uma descrição do Bluetooth.

O IEEE 802.15.4 (ZigBee) é outro padrão de WPAN destinado a fornecer comunicação de dados para equipamentos domésticos de baixa energia e largura de banda muito baixa, como controles remotos, alarme contra roubo, sensores de sistema de aquecimento e dispositivos como crachás inteligentes e leitores de etiqueta. Tais redes são denominadas *redes de sensores sem fio* e suas aplicações e características de comunicação serão discutidas no Capítulo 16.

O padrão *Wireless* MAN IEEE 802.16 (nome comercial: WiMAX) foi ratificado em 2004 e em 2005. O padrão IEEE 802.16 foi projetado como uma alternativa para os enlaces a cabo e DSL para conexão de "última milha" com casas e escritórios. Uma variante do padrão (IEEE 802.20) se destina a superar as redes 802.11 WiFi como principal tecnologia de conexão para computadores laptop e dispositivos móveis em áreas públicas externas e internas.

A tecnologia ATM surgiu a partir de importantes trabalhos de pesquisa e padronização nos setores de telecomunicações e computação, no final dos anos 80, início dos 90 [CCITT 1990]. Seu objetivo é fornecer uma tecnologia para a interligação de redes de longa distância com uma alta largura de banda para aplicações nas áreas de telefonia, comunicação de dados e multimídia (áudio e vídeo de alta qualidade). Embora sua evolução tenha sido mais lenta do que o esperado, atualmente a tecnologia ATM é dominante na interligação de redes de longa distância de alto desempenho. O ATM também foi visto, em aplicações de rede local, como um substituto para redes Ethernet, mas acabou sendo relegado a um segundo plano devido à concorrência com as redes Ethernet de 100 Mbps e 1000 Mbps, que estão disponíveis a um custo muito menor. Delinearemos os princípios de operação da ATM na Seção 3.5.4. Mais detalhes sobre ATM e outras tecnologias de rede de alta velocidade podem ser encontrados em Tanenbaum [2003] e em Stallings [1998a].

3.5.1 Ethernet

A Ethernet foi desenvolvida no Xerox Palo Alto Research Center, em 1973 [Metcalfe e Boggs 1976; Shoch *et al.* 1982; 1985], como parte de um projeto de pesquisa sobre estações de trabalho pessoais e sistemas distribuídos. O protótipo da Ethernet foi a primeira rede local de alta velocidade (para a época), demonstrando sua exeqüibilidade e utilidade ao permitir que computadores de uma única instalação se comunicassem com baixas taxas de erro e sem atraso de comutação. A Ethernet original

funcionava em 3 Mbps e, atualmente, estão disponíveis redes Ethernet com larguras de banda variando de 10 Mbps a 1000 Mbps.

Vamos descrever os princípios de operação da Ethernet de 10 Mbps especificada no padrão IEEE 802.3 [IEEE 1985a]. Essa foi a primeira tecnologia de rede local amplamente usada. Nos dias atuais, a variante de 100 Mbps é a mais empregada, mas o princípio de operação é idêntico ao original. Concluímos esta seção com uma lista das variantes mais importantes da tecnologia de transmissão Ethernet e da largura de banda que disponibilizam. Para ver descrições mais abrangentes da Ethernet em todas as suas variações, consulte Spurgeon [2000].

Uma rede Ethernet é formada por um barramento simples que usa como meio de transmissão um ou mais segmentos contínuos de cabos ligados por hubs ou repetidores. Os hubs e repetidores são dispositivos de interligação de cabos simples, que permitem que um mesmo sinal se propague por todos eles. Várias redes Ethernet podem ser interligadas por meio de *switches* ou pontes Ethernet. Os *switches* e as pontes operam no nível dos quadros Ethernet, encaminhando-os, apropriadamente, para as redes Ethernet adjacentes de destino. Quando interligadas, as várias redes Ethernet aparecem como uma única rede para as camadas de protocolo mais altas, como a IP (veja a Figura 3.10, onde as sub-redes IP 138.37.88 e 138.37.94 são compostas cada uma de várias redes Ethernet ligadas por dispositivos rotulados como *Eswitch*). Em particular, uma requisição ARP (Seção 3.4.2), por ser feita em *broadcast* Ethernet, é confinada dentro dos limites de uma rede Ethernet.

O método de operação das redes Ethernet é definido pela frase "percepção de portadora, múltiplo acesso com detecção de colisão", abreviado por CSMA/CD (*Carrier Sensing, Multiple Access with Collision Detection*), e elas pertencem à classe das redes denominadas de *barramento de disputa* ou *contenção*. Esse tipo de barramento usa um único meio de transmissão para ligar todos os *hosts*. O protocolo que gerencia o acesso ao meio é chamado de MAC (*medium access control*). Como um único enlace conecta todos os *hosts*, o protocolo MAC combina as funções de protocolo da camada física (responsável pela transmissão dos sinais que representam os dados) e da camada de enlace (responsável pelo envio de quadros para os *hosts*) em uma única camada de protocolo.

Difusão (*broadcast*) de pacotes ◊ O método de comunicação em redes CSMA/CD é por difusão (*broadcast*) de pacotes de dados no meio de transmissão. Todas as estações estão continuamente "escutando" o meio para ver se há quadros endereçados a elas. Qualquer estação que queira enviar uma mensagem transmite um ou mais quadros (chamados de *frames* na especificação Ethernet) no meio. Cada quadro contém o endereço físico da estação de destino, o endereço físico da estação de origem e uma seqüência de bits de comprimento variável, representando a mensagem a ser transmitida. A transmissão de dados ocorre a 10 Mbps (ou em velocidades mais altas, especificadas para redes Ethernet de 100 e 1000 Mbps) e o comprimento dos pacotes varia entre 64 e 1518 bytes; portanto, o tempo para transmitir um pacote em uma rede Ethernet de 10 Mbps é de 50 a 1200 microssegundos, dependendo de seu comprimento. O MTU especificado no padrão IEEE é 1518 bytes, embora não exista motivo técnico algum para definir qualquer limite superior fixo em particular, exceto a necessidade de limitar os atrasos causados pela disputa de acesso ao meio.

O endereço físico da estação de destino normalmente se refere a uma única interface de rede. O hardware da interface de rede de cada estação recebe todos os quadros e compara o endereço de destino com seu endereço local. Os quadros endereçados para outras estações são ignorados e aqueles que correspondem ao endereço físico do *host* local são repassados para as camadas superiores. O endereço de destino também pode especificar um endereço de *broadcast* ou de *multicast*. Os endereços "normais", também denominados de *unicast*, são diferenciados dos endereços de *broadcast* e de *multicast* pelo seu bit de ordem mais alta (0 e 1, respectivamente). O endereço físico composto por todos seus bits em 1 é reservado para uso como endereço de *broadcast* e é empregado quando um quadro deve ser recebido por todas as estações da rede. Isso é utilizado, por exemplo, para implementar o protocolo ARP. Qualquer estação que receba um quadro com um endereço de *broadcast* o passará para as camadas superiores. Um endereço de *multicast* especifica uma forma limitada de *broadcast*, o qual é recebido apenas por um grupo de estações cujas interfaces de rede foram configuradas para receber quadros identificados com esse endereço. Nem todas as interfaces de rede Ethernet tratam endereços *multicast*.

O protocolo de rede Ethernet (que fornece a transmissão de quadros Ethernet entre pares de *hosts*) é implementado pelo hardware da própria interface Ethernet; já as camadas superiores são implementadas em software.

Formato do quadro Ethernet ◊ Os quadros (*frames*) transmitidos pelas estações na rede Ethernet têm o seguinte formato:

bytes: 7	*1*	*6*	*6*	*2*	*46 ≤ comprimento ≤ 1500*	*4*
Preâmbulo	S	Endereço de destino	Endereço de origem	Comprimento dos dados	Dados para transmissão	Soma de verificação

Fora os endereços de destino e origem já mencionados, os quadros incluem um prefixo de 8 bytes, um campo de comprimento, uma área de dados e uma soma de verificação. O prefixo é usado para propósitos de sincronização do hardware e consiste em um preâmbulo de sete bytes, cada um contendo o padrão de bits 10101010, seguido de um delimitador de início de quadro, um byte (S, no diagrama), com o padrão 10101011.

Apesar da especificação não permitir mais de 1024 estações em uma única rede Ethernet, os endereços de origem e destino ocupam, cada um, seis bytes, fornecendo 2^{48} endereços diferentes. Isso permite que cada interface de hardware Ethernet receba um endereço exclusivo de seu fabricante, garantindo que todas as estações em qualquer conjunto de redes Ethernet interconectadas tenham endereços exclusivos. O IEEE (Institute of Electrical and Electronic Engineers), dos EUA, atua como autoridade para distribuição e alocação de endereços Ethernet, reservando intervalos separados para os diversos fabricantes de interfaces de hardware Ethernet. Esses endereços são denominados de endereços MAC, pois são usados pela camada de controle de acesso ao meio. Na verdade, os endereços MAC, alocados dessa maneira, também têm sido adotados como endereços exclusivos para outros tipos de rede da família IEEE 802, incluindo 802.11 (WiFi) e 802.15.1 (Bluetooth).

A área de dados contém toda a mensagem que está sendo transmitida ou parte dela (se o comprimento da mensagem ultrapassar 1500 bytes). O limite inferior de 46 bytes para a área de dados garante um comprimento, no mínimo, de 64 bytes para o quadro, que é necessário para garantir que as colisões sejam detectadas por todas as estações da rede, conforme explicado a seguir.

A seqüência de verificação do quadro é uma soma de verificação gerada e inserida pelo remetente e usada pelo destino para validar quadros. Os quadros que contêm somas de verificação incorretas são simplesmente eliminados pela estação destino. Esse é outro exemplo do por quê, para garantir a transmissão de uma mensagem, um protocolo de camada de transporte, como o TCP, deve confirmar o recebimento de cada pacote e retransmitir todos que não foram confirmados. A incidência de erros em redes locais é tão pequena, que o uso desse método de recuperação é totalmente satisfatório quando é exigida uma garantia da entrega e, caso contrário, permite o uso de um protocolo de transporte menos dispendioso, como o UDP.

Colisões ◊ Mesmo no tempo relativamente curto que leva para transmitir quadros, existe uma probabilidade finita de que duas estações da rede tentem transmitir mensagens simultaneamente. Se uma estação tentar transmitir um quadro sem verificar se o meio está sendo usado por outras estações, poderá ocorrer uma colisão.

A Ethernet tem três mecanismos para tratar dessa possibilidade. O primeiro é chamado *percepção de portadora*; o hardware da interface de cada estação "sente" a presença de um sinal no meio, conhecido como *portadora*, em analogia com a transmissão de rádio. Quando uma estação deseja transmitir um quadro, ela espera até que nenhum sinal esteja presente no meio e depois começa a transmitir.

Infelizmente, "sentir" a presença da portadora não evita todas as colisões. A possibilidade de colisão permanece devido ao tempo finito τ para que um sinal inserido em um ponto no meio de transmissão (viajando a aproximadamente 2×10^8 metros por segundo) se propague a todos os outros pontos. Considere duas estações, A e B, que estão prontas para transmitir quadros quase ao mesmo tempo. Se A começa a transmitir primeiro, B pode verificar o meio e não detectar a portadora em dado momento $t < \tau$, após A ter começado a transmitir. Então, B começa a transmitir, *interferindo* na transmissão de A. Tanto o quadro de A, como o de B, serão danificados por essa interferência.

A técnica usada para se recuperar dessa interferência é chamada *detecção de colisão*. Ao mesmo tempo em que transmite, uma estação escuta o meio, e o sinal transmitido é comparado com o recebido, se eles diferirem, significa que houve uma colisão. Quando isso acontece, a estação pára de transmitir e produz um *sinal de reforço de colisão* (*jamming*) para garantir que todas as estações

reconheçam a colisão. Conforme já mencionamos, um comprimento de quadro mínimo é necessário para garantir que as colisões sejam sempre detectadas. Se duas estações transmitirem de forma aproximadamente simultânea, a partir de extremos opostos da rede, elas não identificarão a ocorrência de uma colisão por τ segundos (porque o primeiro remetente ainda deverá estar transmitindo ao receber o segundo sinal). Se os quadros demorarem menos do que τ para serem transmitidos, a colisão não será notada, pois cada estação de envio não verá o outro quadro até ter terminado de transmitir o seu próprio. Porém, isso não garante o pior caso: imediatamente antes de receber um quadro, uma estação "sente" o meio e, estando livre, inicia sua transmissão provocando uma colisão. Essa colisão só será sentida pelo remetente do primeiro quadro após 2τ segundos (τ segundos para o quadro ser transmitido e mais τ segundos para a colisão se propagar até ele). Portanto, para o CSMA/CD funcionar corretamente, a duração mínima de um quadro deve ser 2τ segundos. As estações presentes em pontos intermediários receberão os dois quadros simultaneamente, resultando em corrupção dos dados.

Após o sinal de *jamming*, todas as estações que estão transmitindo e captando cancelam o quadro corrente. As estações que estão transmitindo precisam então tentar retransmitir seus quadros. Agora, surge uma dificuldade maior. Se, após o sinal de *jamming*, todas as estações envolvidas na colisão tentarem retransmitir seus quadros imediatamente, outra colisão provavelmente ocorrerá. Para evitar isso, é usada uma técnica conhecida como *back-off*. Cada uma das estações envolvidas na colisão escolhe um tempo de espera $n\tau$, antes de retransmitir. O valor de n é um número inteiro, aleatório, escolhido separadamente em cada estação e limitado por uma constante L, definida no software de rede. Ao ocorrer uma nova colisão, o valor de L é duplicado e esse procedimento é repetido, se necessário, por até dez tentativas.

Finalmente, o hardware da interface na estação destino calcula a seqüência de verificação e a compara com a soma de verificação transmitida no quadro. Ao usar todas essas técnicas, as estações conectadas em uma rede Ethernet são capazes de gerenciar o uso do meio sem nenhum controle ou sincronismo centralizado.

Eficiência da Ethernet ◊ A eficiência de uma rede Ethernet é a razão entre o número de quadros transmitidos com sucesso e o número máximo teórico que poderia ser transmitido sem colisões. Isso é afetado pelo valor de τ, pois o intervalo de 2τ segundos após o início da transmissão de um quadro, é a "janela de oportunidade" para colisões – nenhuma colisão pode ocorrer depois de 2τ segundos após o início da transmissão de um quadro. Ela também é afetada pelo número de estações presentes na rede e por seu nível de atividade.

Para um cabo de 1 km, o valor de τ é menor do que 5 microssegundos e a probabilidade de colisões é pequena o suficiente para garantir uma alta eficiência. A Ethernet pode atingir uma utilização de canal entre 80 e 95%, embora os atrasos causados pela disputa se tornem significativos quando se ultrapassa 50% de utilização. Como a carga na rede é variável, é impossível *garantir* a entrega de uma determinada mensagem dentro de um tempo fixo qualquer, pois a rede poderia estar completamente carregada quando a mensagem estivesse pronta para transmissão. Mas a *probabilidade* de transferir a mensagem com determinado atraso é tão boa, ou melhor, do que em outras tecnologias de rede.

Medidas empíricas do desempenho de uma rede Ethernet no Xerox PARC são relatadas por Shoch e Hupp [1980] e confirmam essa análise. Na prática, a carga nas redes Ethernet usadas em sistemas distribuídos varia bastante. Muitas redes são empregadas principalmente para interações cliente-servidor assíncronas e essas interações operam, na maior parte do tempo, sem as estações estarem esperando oportunidade para transmitir. Essa situação é bastante diferente daquela encontrada por aplicações que exigem o acesso a um grande volume de dados ou aquelas que transportam fluxos multimídia. Nesses dois últimos casos, a rede tende a ficar sobrecarregada.

Implementações físicas ◊ A descrição anterior define o protocolo de camada MAC para as redes Ethernet. Sua ampla adoção pelo mercado resultou no barateamento do hardware necessário à execução dos algoritmos exigidos para sua implementação, de tal forma que ele é incluído, por padrão, em vários tipos de computadores.

Existe uma grande gama de variações para implementar fisicamente uma rede Ethernet, oferecendo uma série de contrapartidas custo e desempenho. Essas variações resultam do uso de diferentes mídias de transmissão – cabo coaxial, par de cobre trançado (semelhante à fiação telefônica) e fibra óptica – com diferentes limites de alcance de transmissão e de velocidades de sinalização, que resul-

tam em uma largura de banda maior. O IEEE adotou vários padrões para as implementações da camada física e um esquema de atribuição de nomes para distingui-los. São usados nomes como 10Base5 e 100BaseT, com a seguinte interpretação:

<*R*><*B*><*L*> Onde: *R* = taxa de dados, em Mbps
B = tipo de sinalização do meio (banda base ou banda larga)
L = comprimento do segmento máximo, em centenas de metros, ou T (hierarquia de cabos de par trançado)

Na Figura 3.23, são tabulados a largura de banda e a distância máxima de vários padrões atualmente disponíveis assim como e os tipos de cabo. As configurações que terminam com a designação T são implementadas com cabeamento UTP (*Unshielded Twisted Pair*) – pares de fios trançados não-blindados – comuns à fiação telefônica, e normalmente são organizadas em uma hierarquia de hubs onde os computadores são as folhas da árvore. Nesse caso, os comprimentos de segmento dados em nossa tabela são duas vezes a distância máxima permitida de um computador para um hub.

Ethernet para aplicações de tempo real e com exigências de qualidade de serviço ◊ Freqüentemente, é argumentado que o protocolo MAC Ethernet é inerentemente inconveniente para aplicativos de tempo real, ou que exigem qualidade de serviço, devido à incapacidade de garantir um atraso máximo para a entrega de quadros. Entretanto, deve-se salientar que a maioria das instalações de Ethernet agora são em *switches*, conforme ilustrado na Figura 3.10 e descrito na Seção 3.3.7 (em vez de hubs ou cabos com uma derivação para cada conexão, como era o caso anteriormente). O uso de *switches* resulta em um segmento separado para cada *host* sem quadros transmitidos nele, além daqueles endereçados a esse *host*. Portanto, se o tráfego para o *host* é proveniente de uma única origem, não há disputa pelo meio – a eficiência é de 100% e a latência é constante. A possibilidade de disputa surge apenas no interior dos *switches* e, estes, freqüentemente podem ser projetados para tratar simultaneamente vários quadros. Assim, uma instalação Ethernet comutada, aquela baseada em *switches*, pouco carregada, se aproxima dos 100% de eficiência com uma latência baixa constante e, portanto, podem ser usadas com sucesso nessas áreas de aplicação críticas.

Um passo a mais na direção do suporte em tempo real para protocolos MAC do estilo Ethernet é descrito em [Rether, Pradhan e Chiueh 1998], e um esquema semelhante é implementado em uma extensão de código-fonte aberto do Linux [RTnet]. Essas estratégias de software tratam do problema da disputa implementando, em nível aplicativo, um protocolo cooperativo para reservar fatias de tempo para uso do meio. Isso depende da colaboração de todos os *hosts* conectados a um segmento.

	10Base5	*10BaseT*	*100BaseT*	*1000BaseT*
Taxa de dados	10 Mbps	10 Mbps	100 Mbps	1000 Mbps
Comprimentos de segmento max.:				
Par trançado (UTP)	100 m	100 m	100 m	25 m
Cabo coaxial (STP)	500 m	500 m	500 m	25 m
Fibra multi-modo	2000 m	2000 m	500 m	500 m
Fibra mono-modo	25000 m	25000 m	20000 m	2000 m

Figura 3.23 Distâncias e taxas de transmissão de redes Ethernet.

3.5.2 Rede local sem fio IEEE 802.11 (WiFi)

Nesta seção, resumimos as características especiais da comunicação sem fio (*wireless*) que devem ser tratadas por uma tecnologia de rede local sem fio e explicamos como o padrão IEEE 802.11 trata delas. O padrão IEEE 802.11 estende o princípio da percepção de portadora e acesso múltiplo (CSMA), empregado pela tecnologia Ethernet (IEEE 802.3), para se adequar às características da comunicação

sem fio. O padrão 802.11 se destina a suportar comunicação entre computadores localizados dentro de cerca de 150 metros uns dos outros, em velocidades de até 54 Mbps.

A Figura 3.24 ilustra parte de uma intranet que inclui uma rede local sem fio. Vários equipamentos móveis sem fio se comunicam com o restante da intranet por meio de uma estação de base, que serve como *ponto de acesso* para a rede local cabeada. Uma rede sem fio que se conecta ao mundo exterior através de uma rede cabeada convencional, por meio de um ponto de acesso, é conhecida como *rede de infra-estrutura*.

Uma configuração alternativa para redes sem fio é conhecida como *rede ad hoc*. As redes *ad hoc* não contêm ponto de acesso, nem estação de base. Elas são construídas dinamicamente, como resultado da detecção mútua de dois ou mais equipamentos móveis com interfaces sem fio na mesma vizinhança. Uma rede *ad hoc* se estabelece quando, por exemplo, em uma sala, dois ou mais laptops, se detectam e estabelecem uma comunicação entre eles. Nesse caso, então, eles poderiam compartilhar arquivos, ativando um processo servidor de arquivos em uma das máquinas.

No nível físico, as redes IEEE 802.11 usam sinais de radiofreqüência (nas faixas livres de licença, 2,4 GHz e 5 GHz), ou sinalização infravermelha, como forma de transmissão. A versão baseada em radiofreqüência tem maior atenção comercial e vamos descrevê-la. O padrão IEEE 802.11b foi a primeira variante a ter seu uso bastante difundido. Ele opera na faixa de 2,4 GHz e suporta comunicação de dados em até 11 Mbps. Esse tipo de rede, a partir de 1999 em diante, se tornou bastante comum em muitos escritórios, casas e locais públicos, permitindo que computadores laptop e PDAs acessassem dispositivos interligados em rede local ou a Internet. O padrão IEEE 802.11g é um aprimoramento mais recente do padrão 802.11b, que usa a mesma faixa de 2,4 GHz, mas utiliza uma técnica de sinalização diferente para atingir velocidades de até 54 Mbps. Finalmente, a variante 802.11a, que opera na faixa de 5 GHz e oferece 54 Mbps de largura de banda, mas com um alcance menor. Todas as variantes usam diversas técnicas de seleção de freqüência e saltos de freqüência para evitar interferência externa, e mútua, entre redes locais sem fio independentes, o que não vamos detalhar aqui. Em vez disso, vamos focar nas alterações feitas no mecanismo de controle de acesso ao meio CSMA/CD para que, em todas as versões de 802.11, fosse possível a transmissão de dados via sinal de rádio.

Assim como o padrão Ethernet, o protocolo MAC 802.11 oferece oportunidades iguais a todas as estações para usar o canal de comunicação e uma estação transmitir diretamente para qualquer outra. Um protocolo MAC controla o uso do canal por várias estações. No que diz respeito à Ethernet, a camada MAC também executa as funções de camada física e de enlace de dados, distribuindo quadros de dados para os *hosts* de uma rede.

Vários problemas surgem do uso de ondas de rádio, em vez de fios, como meio de transmissão. Esses problemas se originam do fato de que os mecanismos de percepção da portadora e detecção de

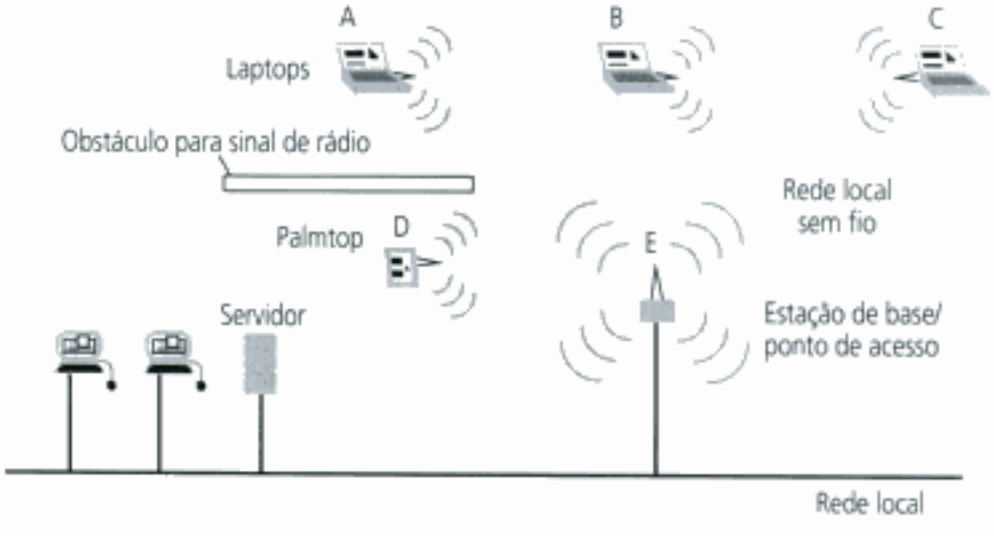


Figura 3.24 Configuração de rede local sem fio.

colisão empregados nas redes Ethernet são eficazes somente quando a intensidade dos sinais é aproximadamente igual em toda a rede.

Lembramos que o objetivo da percepção de portadora é verificar se o meio está livre para que uma transmissão seja iniciada, e a detecção de colisão serve para determinar se houve interferências durante uma transmissão. Como a intensidade do sinal não é uniforme em toda a área na qual as redes locais sem fio operam, a detecção da portadora e de colisão podem falhar das seguintes maneiras:

Estações ocultas: o sinal de portadora pode não ser detectado mesmo se houver uma outra estação transmitindo. Isso está ilustrado na Figura 3.24. Se o palmtop D estiver transmitindo para a estação de base E, o laptop A poderá não captar o sinal de D, devido ao obstáculo para o sinal de rádio mostrado. Então, A poderia começar a transmitir, causando uma colisão em E, a menos que medidas fossem tomadas para evitar isso.

Desvanecimento do sinal (fading): a lei do inverso do quadrado da propagação de ondas eletromagnéticas nos diz que a intensidade dos sinais de rádio diminui rapidamente com a distância do transmissor. As estações de uma rede local sem fio podem ficar fora do alcance do sinal de rádio de outras estações da mesma rede. Assim, na Figura 3.24, o laptop A talvez não possa detectar uma transmissão de C, embora cada uma delas possa transmitir com sucesso para B ou E. O desvanecimento anula tanto a percepção de portadora quanto a detecção de colisão.

Mascaramento de colisões: infelizmente, a técnica de "sentir" usada na Ethernet para detectar colisões não é muito eficiente em redes baseada em sinal de rádio. Devido à lei do inverso do quadrado, citada anteriormente, o sinal local gerado será sempre muito mais forte do que qualquer sinal originado em outro lugar, abafando a transmissão remota. Portanto, os laptops A e C poderiam transmitir simultaneamente para E. Nenhum deles detectaria essa colisão, mas E receberia apenas uma transmissão truncada.

Apesar de sua falibilidade, a percepção de portadora não é dispensada nas redes IEEE 802.11; ela é ampliada pela adição de um mecanismo de *reserva de slot* no protocolo MAC. O esquema resultante é chamado de *percepção de portadora de acesso múltiplo com evitação de colisão* (CSMA/CA).

Quando uma estação está pronta para transmitir, ela sente o meio. Se ela não detectar nenhum sinal de portadora, pode presumir que uma das seguintes condições é verdadeira:

1. o meio está disponível;
2. uma estação fora do alcance está no procedimento de solicitação de um *slot*;
3. uma estação fora do alcance está usando um *slot* que tinha reservado anteriormente.

O protocolo de reserva de *slot* envolve a troca de duas mensagens curtas (quadros) entre o emissor e o receptor. A primeira é um quadro *request to send* (RTS) do emissor para o receptor. A mensagem RTS especifica uma duração para o *slot* solicitado. O receptor responde com um quadro *clear to send* (CTS), repetindo a duração do *slot*. O efeito dessa troca de quadros é o seguinte:

- As estações dentro do alcance do emissor também receberão o quadro RTS e tomarão nota da duração.
- As estações dentro do alcance do receptor receberão o quadro CTS e tomarão nota da duração.

Como resultado, todas as estações dentro do alcance do emissor e do receptor não transmitirão pela duração do *slot* solicitado, deixando o canal livre para o emissor transmitir um quadro de dados de comprimento apropriado. Finalmente, a recepção bem-sucedida do quadro de dados é confirmada pelo receptor. Isso auxilia a tratar dos problemas de interferências externas no canal. O recurso de reserva de *slot* do protocolo MAC ajuda a evitar colisões das seguintes maneiras:

- Os quadros CTS ajudam a evitar os problemas de estação oculta e desvanecimento.
- Os quadros RTS e CTS possuem curta duração; portanto, o risco de colisões com eles é baixo. Se uma colisão ocorrer e for detectada, ou se um RTS não resultar em um CTS, o RTS será retransmitido usando um período de *back-off* aleatório, como na Ethernet.
- Quando os quadros RTS e CTS tiverem sido trocados corretamente, não deverão ocorrer colisões envolvendo os quadros de dados subseqüentes, nem nos quadros de confirmação, a não ser que um desvanecimento intermitente tenha impedido o outro participante de receber um deles.

Segurança ◊ A privacidade e a integridade da comunicação é uma preocupação óbvia nas redes sem fio. Qualquer estação que esteja dentro do alcance e equipada com um receptor/transmissor poderá fazer parte da rede ou, falhando isso, bisbilhotar as transmissões entre outras estações. A primeira tentativa para tratar dos problemas de segurança para o padrão 802.11 é intitulada WEP (*Wired Equivalent Privacy*). Infelizmente, o WEP é tudo, menos o que seu nome implica. Seu projeto era falho em vários aspectos, o que permitia uma invasão muito facilmente. Descreveremos suas vulnerabilidades e resumiremos o status atual relativo aos aprimoramentos, na Seção 7.6.4.

3.5.3 Redes pessoais sem fio – Bluetooth IEEE 802.15.1

Bluetooth é uma tecnologia de rede pessoal sem fio que surgiu a partir da necessidade de conectar telefones móveis à PDAs, computadores laptop e outros equipamentos pessoais sem fios. Um grupo de interesse especial (SIG – *Special Interest Group*) de fabricantes de telefones móveis e computadores, liderados por L.M. Ericsson, desenvolveu uma especificação para uma rede pessoal sem fio (WPAN – *Wireless Personal Area Network*) para a transmissão de fluxos de voz digitalizada como dados [Haartsen *et al.* 1998]. A versão 1.0 do padrão Bluetooth foi publicada em 1999, seu nome é devido a um rei viking. Então o grupo de trabalho do IEEE 802.15 o adotou como sendo o padrão 802.15.1 e publicou uma especificação para as camadas física e de enlace de dados [IEEE 2002].

As redes Bluetooth diferem substancialmente do IEEE 802.11 (WiFi), refletindo os requisitos de seu contexto de utilização e as metas de custo e consumo de energia para os quais foi projetado. O padrão Bluetooth foi projetado para permitir comunicação entre dispositivos pequenos e de baixo custo, como os fones de ouvido sem fio adaptados à orelha, que recebem fluxos de áudio de um telefone celular, ou interconexões entre computadores, telefones, PDAs e outros dispositivos móveis. A meta de custo era adicionar apenas cinco dólares ao preço final de um dispositivo portátil. A meta de consumo de energia era usar apenas uma pequena fração da energia total da bateria de um telefone celular ou PDA, e operar por várias horas, mesmo com baterias pequenas, empregadas tipicamente em aparelhos acoplados ao corpo, como fones de ouvido.

As aplicações alvo do Bluetooth exigem menos largura de banda e um alcance de transmissão menor do que as aplicações de rede local sem fio do padrão 802.11. Isso é bom, pois o Bluetooth opera na mesma faixa de freqüência livre de licença de 2,4 GHz que as redes WiFi, telefones sem fio e muitos outros sistemas de comunicação de serviços de emergência, sendo portanto muito ocupada. A transmissão se dá em baixa energia, saltando a uma taxa de 1600 vezes por segundo entre 79 sub-faixas de 1 MHz da faixa de freqüência permitida para minimizar os efeitos da interferência. A potência de saída dos dispositivos Bluetooth convencionais é de 1 miliwatt, fornecendo uma área de cobertura de apenas 10 metros; dispositivos de 100 miliwatts, com um alcance de até 100 metros, são permitidos para aplicações como redes domésticas. A eficiência no uso da energia é melhorada pela inclusão de um recurso de *alcance adaptativo*, que ajusta a potência da transmissão para um nível mais baixo, quando dispositivos que se comunicam estão próximos (conforme determinado pela intensidade dos sinais recebidos inicialmente).

Os nós Bluetooth se associam dinamicamente em pares, sem exigir nenhum conhecimento anterior da rede. O protocolo de associação será descrito a seguir. Após uma associação bem-sucedida, o nó iniciador tem a função de *mestre* e o outro, a de *escravo*. Uma *Piconet* é uma rede associada dinamicamente, composta de um mestre e até sete escravos ativos. O mestre controla o uso do canal de comunicação, alocando *slots* de tempo para cada escravo. Um nó que esteja em mais de uma Piconet pode atuar como ponte, permitindo que os mestres se comuniquem – várias Piconets ligadas dessa maneira são chamadas de *Scatternet*. Muitos tipos de dispositivo têm a capacidade de atuar como mestre ou como escravo.

Todos os nós Bluetooth também são equipados com um endereço MAC globalmente exclusivo de 48 bits (veja a Seção 3.5.1), embora apenas o endereço MAC do mestre seja usado no protocolo. Quando um escravo se torna ativo em uma Piconet, ele recebe um endereço local temporário no intervalo de 1 a 7. O objetivo disso é reduzir o comprimento dos cabeçalhos do quadro. Além dos sete escravos ativos, uma Piconet pode conter até 255 nós *estacionados* (*parked*) no modo de baixa energia, esperando um sinal de ativação do mestre.

Protocolo de associação ◊ Para economizar energia, os dispositivos permanecem no modo de suspensão ou *espera* (*standby*) antes que qualquer associação seja feita ou quando nenhuma comunicação

recente ocorreu. No modo de espera, eles são acionados para captar mensagens de ativação em intervalos que variam de 0,64 a 2,56 segundos. Para se associar a um nó próximo conhecido (estacionado), o nó iniciador transmite uma seqüência de 16 quadros de *página*, em 16 faixas de freqüência, o qual pode ser repetido várias vezes. Para entrar em contato com qualquer nó desconhecido dentro do intervalo, o iniciador deve primeiro transmitir uma seqüência de mensagens de *pergunta*. Essas seqüências de transmissão podem ocupar até cerca de 5 segundos, no pior caso, levando a um tempo de associação máximo de 7–10 segundos.

A associação é seguida por uma troca de autenticação opcional, baseada em "fichas" de autenticação (*tokens*) fornecidas pelo usuário ou recebidas anteriormente, para garantir que a associação se dê com o nó pretendido e não com um impostor. Um escravo permanece sincronizado com o mestre, observando os quadros transmitidos regularmente por este, mesmo quando eles não são endereçados para esse escravo. Um escravo que está inativo pode ser colocado no modo "estacionado" pelo mestre, liberando seu *slot* na Piconet para uso por outro nó.

Os requisitos de oferecer canais de comunicação síncronos, com qualidade de serviço adequada para a transmissão de áudio bidirecional em tempo real (por exemplo, entre um telefone e o fone de ouvido sem fio), assim como comunicação assíncrona para troca de dados, impuseram uma arquitetura de rede muito diferente do projeto de múltiplo acesso e melhor esforço das redes Ethernet e WiFi. A comunicação síncrona é obtida pelo uso de um protocolo de comunicação bidirecional simples entre um mestre e um de seus escravos, denominado enlace *orientado a conexão síncrona* (*SCO – Synchronous Connection Oriented*), no qual mestre e escravo enviam quadros alternadamente. A comunicação assíncrona é obtida por um enlace *sem conexão assíncrona* (*ACL – Asynchronous ConnectionLess*), no qual o mestre envia quadros de consulta periodicamente (*polling*) para os escravos e estes transmitem somente após receberem esses quadros.

Todas as variantes do protocolo Bluetooth usam quadros que respeitam a estrutura mostrada na Figura 3.25. Uma vez estabelecida uma Piconet, o *código de acesso* consiste em um preâmbulo fixo para sincronizar o emissor e o receptor e identificar o início de um slot, seguido de um código derivado do endereço MAC do mestre, que identifica, de forma exclusiva, a Piconet. Este último garante que os quadros sejam corretamente direcionados em situações onde existem várias Piconets sobrepostas. Como o meio provavelmente terá ruído e a comunicação em tempo real não pode contar com retransmissões, cada bit do cabeçalho é transmitido de forma triplicada. Isso provê uma redundância de sua informação, incluindo a soma de verificação.

O campo de endereço tem apenas três bits para permitir endereçamento a qualquer um dos sete escravos correntemente ativos. O endereço zero indica um *broadcast*. Existem campos para controle de fluxo, confirmação e número de seqüência, cada um deles com largura equivalente a 1 bit. O bit de controle de fluxo é usado por um escravo para indicar ao mestre que seus buffers estão cheios; nesse caso, o mestre deve esperar do escravo um quadro com um bit de confirmação diferente de zero. O bit de número de seqüência é alternado entre 0 e 1 para cada novo quadro enviado para o mesmo nó; isso permite a detecção de quadros duplicados (isto é, retransmitidos).

Os enlaces SCO são usados em aplicações críticas em relação a tempo, como a transmissão de uma conversa de voz bidirecional. Os quadros têm de ser curtos para manter a latência baixa e há pouco interesse em identificar, ou retransmitir, pacotes corrompidos em tais aplicações, pois os dados

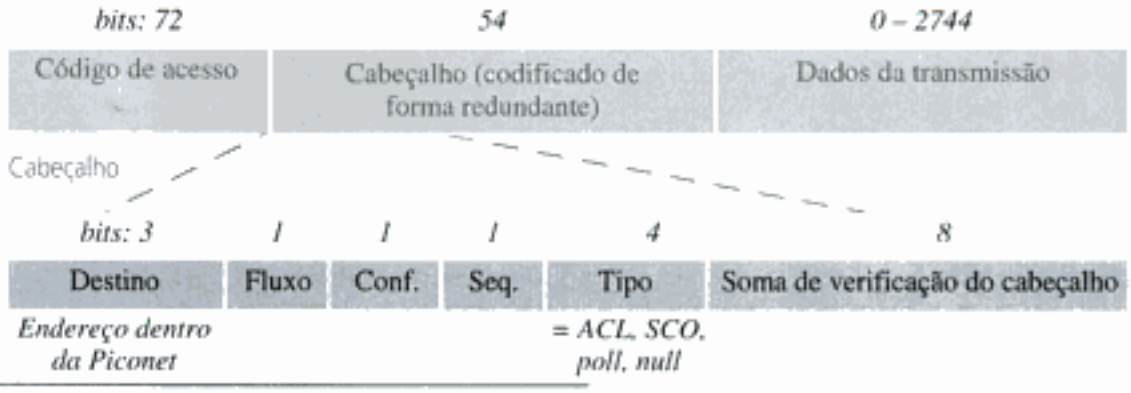


Figura 3.25 Estrutura de quadro Bluetooth.

retransmitidos chegariam tarde demais para serem úteis. Por isso, o SCO usa um protocolo simples e altamente redundante, no qual 80 bits de dados de voz são normalmente transmitidos de forma triplicada para produzir uma área de dados de 240 bits. Duas réplicas quaisquer de 80 bits que correspondam são aceitas como válidas.

Por outro lado, os enlaces ACL são usados para aplicações de transferência de dados, como a sincronização de agenda de endereços entre um computador e um telefone. A área de dados não é duplicada, mas pode conter uma soma de verificação interna que é verificada em nível aplicativo e, no caso de falha de retransmissão, pode ser solicitada.

Os dados são transmitidos em quadros que ocupam *slots* de tempo de duração de 625 microssegundos, cronometrados e alocados pelo nó mestre. Cada pacote é transmitido em uma freqüência diferente, em uma seqüência de saltos definida pelo nó mestre. Como esses *slots* não são grandes o bastante para permitir uma área de dados de tamanho significativo, os quadros podem ser estendidos para ocupar um, três ou cinco *slots*. Essas características, e o método de transmissão física subjacente, resultam em um desempenho máximo de 1 Mbps para uma Piconet, o que acomoda até três canais duplex síncronos de 64 Kbps entre um mestre e seus escravos ou um canal para transferência de dados assíncrona em velocidades de até 723 Kbps. Esses valores são calculados para a versão mais redundante do protocolo SCO, conforme descrito anteriormente. São definidas outras variantes do protocolo que alteram a relação entre a robusteza e a simplicidade (e, portanto, baixo custo computacional) dos dados triplicados para se obter um desempenho maior.

Ao contrário da maioria dos padrões de rede, o Bluetooth inclui especificações (chamadas de *perfis*) para vários protocolos em nível de aplicação, alguns dos quais são muito orientados para aplicativos em particular. O objetivo desses perfis é aumentar a probabilidade de que os dispositivos fabricados por diferentes fornecedores funcionem em conjunto. São incluídos 13 perfis de aplicativo: acesso genérico, descoberta de serviço, porta serial, troca de objeto genérica, acesso a rede local, interligação em rede *dial-up*, fax, telefonia sem fio, sistema de intercomunicação, fones de ouvido, *push* de objetos (trocas), transferência de arquivo e sincronização. Outros perfis estão sendo elaborados, incluindo tentativas ambiciosas de transmitir música em alta qualidade e até vídeo por meio de Bluetooth.

O padrão Bluetooth ocupa um nicho especial na classe das redes locais sem fio. Ele cumpre seu objetivo de projeto de oferecer comunicação síncrona de áudio em tempo real com qualidade de serviço satisfatória (consulte o Capítulo 17 para ver uma discussão melhor sobre os problemas de qualidade do serviço), assim como permitir a transferência de dados assíncrona, usando hardware de custo muito baixo, compacto e portátil, com baixo consumo de energia e largura de banda limitada.

Sua principal desvantagem é o tempo que leva (até 10 segundos) para efetuar a associação de novos dispositivos. Isso impede seu uso para certas aplicações, especialmente onde os dispositivos estão se movendo uns em relação aos outros, impedindo sua utilização, por exemplo, para pagamento automatizado de pedágios em estradas ou para transmitir informações promocionais para usuários de telefonia móvel quando eles passam em frente a uma loja. Outra referência útil sobre interligação em rede Bluetooth é o livro de Bray e Sturman [2002].

A versão 2.0 do padrão Bluetooth, com desempenho de até 3 Mbps – suficiente para enviar áudio com qualidade de CD – estava em processo de aprovação quando este livro estava sendo produzido e aprimoramentos, incluindo um mecanismo de associação mais rápido e endereços de Piconet maiores, estavam sendo desenvolvidos.

3.5.4 Redes ATM

A rede ATM (*Asynchronous Transfer Mode*) foi projetada para transportar uma ampla variedade de dados, incluindo dados multimídia como voz e vídeo. Trata-se de uma rede rápida para comutação de pacotes, baseada em um método de roteamento de pacotes conhecido como *revezamento de célula* (*cell relay*) que opera em velocidades maiores que a comutação de pacotes convencional. Ela melhora sua velocidade evitando controle de fluxo e verificação de erros nos nós intermediários em uma transmissão. Os enlaces de transmissão e os nós devem, portanto, ter baixa probabilidade de erros. Outro fator que afeta o desempenho são as pequenas unidades de dados de comprimento fixo transmitidas, as *células*, o que reduz o tamanho do buffer, a complexidade e o atraso de enfileiramento nos nós intermediários. A rede ATM opera em um modo conectado, mas uma conexão só pode ser estabelecida

se recursos necessários estiverem disponíveis. Uma vez estabelecida uma conexão, sua qualidade (isto é, suas características de largura de banda e latência) podem ser garantidas.

ATM é uma tecnologia de comutação de dados que pode ser implementada sobre as redes de telefonia digital existentes, as quais até agora eram síncronas. Quando uma rede ATM é implementada sobre uma rede de enlaces digitais síncronos de alta velocidade, como aquelas especificadas para a SONET (*Synchronous Optical NETwork*) [Omidyar e Aldridge 1993], ela produz uma rede de pacotes de alta velocidade muito mais flexível, com muitas conexões virtuais. Cada conexão virtual ATM fornece garantias de largura de banda e latência. Os circuitos virtuais resultantes podem ser usados para suportar uma ampla variedade de serviços com velocidades variadas. Isso inclui voz (32 kbps), fax, vídeo e televisão de alta definição (100–150 Mbps). O padrão ATM [CCITT 1990] recomenda o aprovisionamento de circuitos virtuais com taxas de transferência de dados de até 155 Mbps ou 622 Mbps.

As redes ATM também podem ser implementadas em *modo nativo*, diretamente sobre fibra óptica, cabos de cobre e outra mídia de transmissão, possibilitando larguras de banda de até vários gigabits por segundo com a atual tecnologia de fibra. Esse é o modo no qual ela é empregada em redes locais e metropolitanas.

O ATM é estruturado em três camadas, representadas pelas listras mais escuras na Figura 3.26. A *camada de adaptação ATM* é uma camada fim-a-fim implementada apenas nos *hosts* de envio e recepção. Ela se destina a suportar protocolos de mais alto nível, como TCP/IP e X25, sobre a camada ATM. Diferentes versões da camada de adaptação podem fornecer uma gama de funções de adaptação, de acordo com os requisitos dos diferentes protocolos de nível mais alto. Isso inclui algumas funções comuns, como montagem e desmontagem de pacotes para uso na construção de protocolos específicos de mais alto nível.

A *camada ATM* fornece um serviço orientado a conexão que transmite pacotes de comprimento fixo chamados *células*. Uma conexão consiste em uma seqüência de canais virtuais dentro de caminhos virtuais. Um *canal virtual* (VC – *Virtual Channel*) é uma associação lógica unidirecional entre dois pontos extremos de um enlace no caminho físico entre a origem e o destino. Um *caminho virtual* (VP – *Virtual Path*) é um feixe de canais virtuais associados a um caminho físico entre dois nós de comutação. Os caminhos virtuais se destinam a ser usados para suportar conexões semipermanentes entre pares de pontos extremos. Os canais virtuais são alocados dinamicamente quando as conexões são estabelecidas.

Os nós em uma rede ATM podem desempenhar três funções distintas:

- *hosts*, que enviam e recebem mensagens;

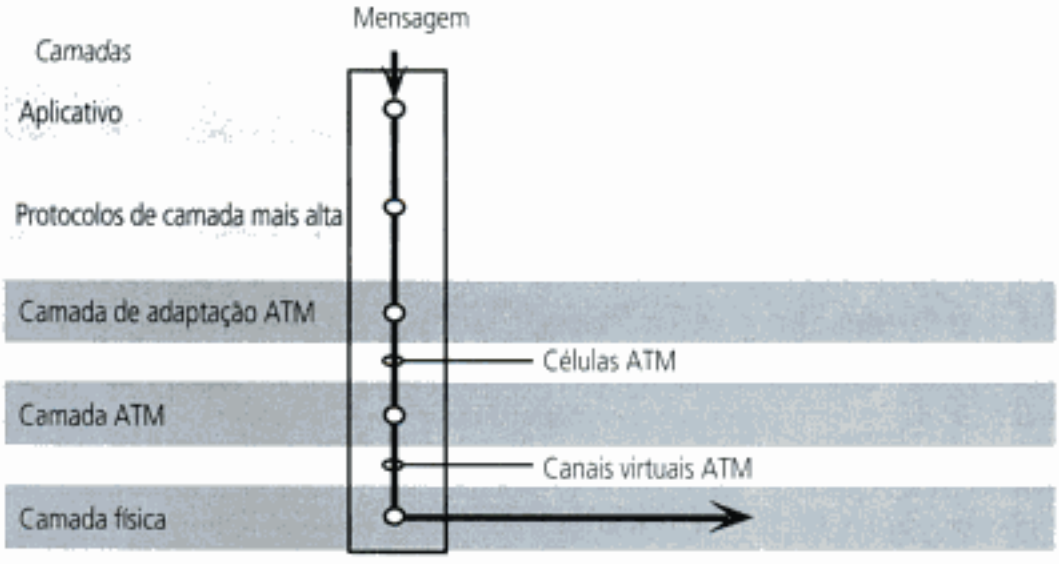


Figura 3.26 Camadas do protocolo ATM.

Figura 3.27 Formato de uma célula ATM.

- *comutadores VP*, que contêm tabelas mostrando a correspondência entre os caminhos virtuais de entrada e saída;
- *comutadores VP/VC*, que mantêm tabelas de correspondência entre caminhos virtuais e canais virtuais.

Uma célula ATM tem um cabeçalho de 5 bytes e um campo de dados de 48 bytes (Figura 3.27). Um campo de dados completo é sempre enviado, mesmo quando está parcialmente preenchido com dados. O cabeçalho contém um identificador de canal virtual (VCI – *Virtual Channel Identifier*) e um identificador de caminho virtual (VPI – *Virtual Path Identifier*) os quais, juntos, fornecem as informações necessárias para encaminhar uma célula na rede. O identificador de caminho virtual se refere a um caminho virtual em particular no enlace físico em que a célula é transmitida. O identificador de canal virtual se refere a um único canal virtual específico dentro do caminho virtual. Outros campos de cabeçalho são usados para indicar o tipo de célula, sua prioridade e o limite.

Quando uma célula chega em um comutador VP, o identificador de caminho virtual presente no cabeçalho é pesquisado em sua tabela de roteamento para encontrar o identificador de caminho virtual correspondente do caminho físico de saída; veja a Figura 3.28. Ele coloca o novo identificador de caminho virtual no cabeçalho e então transmite a célula no caminho físico de saída. Um comutador VP/VC pode realizar um roteamento semelhante, com base nos identificadores VP e VC.

Note que os identificadores VP e VC são definidos de forma local. Esse esquema tem a vantagem de não haver necessidade de identificadores globais para toda a rede, os quais precisariam ser números muito grandes. Um esquema de endereçamento global introduziria sobrecargas administrativas e exigiria que os cabeçalhos de célula e as tabelas dos comutadores contivessem mais informações.

A rede ATM fornece um serviço com baixa latência – o atraso na troca é de cerca de 25 microssegundos por comutador, proporcionando, por exemplo, uma latência de 250 microssegundos quando

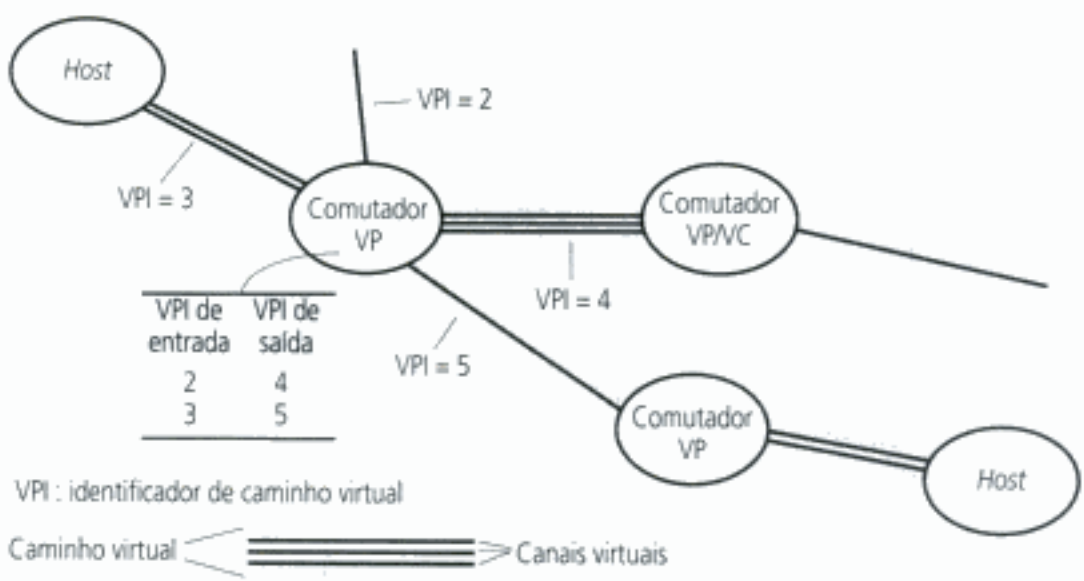


Figura 3.28 Comutação de caminhos virtuais em uma rede ATM.

uma mensagem passa por 10 comutadores. Isso fecha bem com os requisitos de desempenho estimados para sistemas distribuídos (Seção 3.2), implicando que uma rede ATM pode ser usada na comunicação entre processos e interações cliente-servidor com um desempenho semelhante, ou melhor, do que o agora disponível nas redes locais. Também disponibiliza canais de largura de banda muito alta, com qualidade de serviço garantida, convenientes para a transmissão de fluxos de dados multimídia em velocidades de até 600 Mbps. Velocidades de gigabits por segundo podem ser alcançadas em redes ATM puras.

3.6 Resumo

Focalizamos os conceitos e técnicas para a interligação de redes necessários à compreensão de sistemas distribuídos e os abordamos do ponto de vista de um projetista desse tipo de sistema. As redes de pacotes e os protocolos em camada fornecem a base da comunicação em sistemas distribuídos. As redes locais são baseadas na transmissão (difusão) de pacotes em um meio compartilhado; Ethernet é a tecnologia dominante. As redes de longa distância são baseadas na comutação de pacotes para direcionar pacotes para seus destinos em uma rede conectada. O roteamento é um mecanismo chave e, para tal, é usado uma variedade de algoritmos, dos quais vetor de distância é o mais simples, porém eficaz. O controle de congestionamento é necessário para evitar o estouro dos buffers no receptor e nos nós intermediários.

A interligação de redes é construída dispondo-se, em camada, um protocolo de rede virtual sobre conjuntos de redes interconectadas por roteadores. Os protocolos TCP/IP permitem que os computadores se comuniquem na Internet de maneira uniforme, independentemente de estarem na mesma rede local ou em países diferentes. Os padrões da Internet incluem muitos protocolos em nível de aplicação que são convenientes para uso em aplicativos distribuídos remotos. O padrão IPv6 tem um espaço de endereçamento muito maior e oferece suporte para novos requisitos de aplicativo, como qualidade do serviço e segurança, necessários para a evolução futura da Internet.

O MobileIP oferece uma infra-estrutura para os usuários móveis permitindo a migração entre redes de longa distância e redes locais sem fio baseadas nos padrões IEEE 802. A rede ATM oferece comunicação assíncrona com uma alta largura de banda. Por ser baseada em circuitos virtuais possibilita a garantia de qualidade de serviço.

Exercícios

3.1 Um cliente envia uma mensagem de requisição de 200 bytes para um serviço, o qual produz uma resposta contendo 5000 bytes. Estime o tempo total gasto para completar o pedido em cada um dos casos a seguir, com as considerações de desempenho listadas abaixo:

i) Usando comunicação sem conexão (datagrama) (por exemplo, UDP).

ii) Usando comunicação orientada a conexão (por exemplo, TCP).

iii) O processo servidor está na mesma máquina que o cliente.

[Latência por pacote (local ou remoto, acarretada no envio e na recepção): 5 ms
Tempo de estabelecimento da conexão (somente para TCP): 5 ms
Taxa de transferência de dados: 10 Mbps
MTU: 1000 bytes
Tempo de processamento da requisição no servidor: 2 ms
Suponha que a rede esteja pouco carregada.] *páginas 71, 104–105*

3.2 A Internet é grande demais para um roteador conter informações de roteamento para todos destinos. Como o esquema de roteamento da Internet trata desse problema? *páginas 89–90, 97–98*

3.3 Qual é a função de um *switch* Ethernet? Quais tabelas ele mantém? *páginas 89–90, 111–112*

3.4 Faça uma tabela semelhante à Figura 3.5, descrevendo o trabalho feito pelo software em cada camada de protocolo, quando aplicativos Internet e o conjunto TCP/IP são implementados sobre uma rede Ethernet. *páginas 80–82, 104–105, 111–112*

3.5 Como o princípio fim-a-fim [Saltzer *et al.* 1984] foi aplicado no projeto da Internet? Considere como o uso de um protocolo de rede de circuito virtual, no lugar do IP, teria impacto sobre a exeqüibilidade da World Wide web. *páginas 47–48, 82–83, 90–91,* [www.reed.com]

3.6 Podemos ter certeza de que dois computadores na Internet não têm os mesmos endereços IP? *páginas 93–94*

3.7 Compare a comunicação sem conexão (UDP) e orientada a conexão (TCP) para a implementação de cada um dos seguintes protocolos nas camadas de aplicação ou de apresentação:

i) acesso a terminal virtual (por exemplo, Telnet);

ii) transferência de arquivo (por exemplo, FTP);

iii) localização de usuário (por exemplo, rwho, finger);

iv) navegação em informações (por exemplo, HTTP);

v) chamada remota de procedimentos. *páginas 104–105*

3.8 Explique como é possível uma seqüência de pacotes transmitidos por meio de uma rede de longa distância chegarem ao seu destino em uma ordem diferente daquela em que foram enviados. Por que isso não pode acontecer em uma rede local? Isso pode acontecer em uma rede ATM? *páginas 83, 112, 120*

3.9 Um problema específico que deve ser resolvido nos protocolos de acesso a terminal remoto, como Telnet, é a necessidade de transmitir eventos excepcionais, como "sinais de encerramento" do "terminal" para o *host*, sem esperar a transmissão dos dados que está em andamento. O "sinal de encerramento" deve chegar ao destino antes de quaisquer outras transmissões. Discuta a solução desse problema com protocolos orientados a conexão e sem conexão. *páginas 104–105*

3.10 Quais são as desvantagens de usar *broadcast* em nível de rede para localizar recursos:

i) em uma única Ethernet?

ii) em uma intranet?

Até que ponto o *multicast* Ethernet é um aprimoramento em relação ao *broadcast*? *páginas 111–112*

3.11 Sugira um esquema que aprimore o MobileIP para fornecer acesso a um servidor web em um dispositivo móvel que às vezes é conectado à Internet pelo telefone móvel e outras vezes tem uma conexão com fio com a Internet em um de vários locais. *página 103–104*

3.12 Mostre a seqüência de alterações nas tabelas de roteamento da Figura 3.8 que ocorreriam (de acordo com o algoritmo RIP dado na Figura 3.9) após o enlace rotulado como 3 na Figura 3.7 ser desfeito. *páginas 83– 88*

3.13 Use o diagrama da Figura 3.13 como base para uma ilustração que mostre a fragmentação e o encapsulamento de um pedido HTTP para um servidor e a resposta resultante. Suponha que o pedido seja uma mensagem HTTP curta, mas que a resposta inclua pelo menos 2000 bytes de código HTML. *páginas 79–80, 92–93*

3.14 Considere o uso de TCP em um cliente de terminal remoto Telnet. Como a entrada do teclado deve ser colocada no buffer do cliente? Investigue os algoritmos de Nagle e de Clark [Nagle 1984, Clark 1982] para controle de fluxo e compare-os com o algoritmo simples descrito na página 103, quando TCP for usado por: (a) um servidor web, (b) um aplicativo Telnet, (c) um aplicativo gráfico remoto com entrada de mouse contínua. *páginas 87–88, 106*

3.15 Construa um diagrama de rede semelhante à Figura 3.10 para a rede local de sua instituição ou empresa. *página 89*

3.16 Descreva como você configuraria um *firewall* para proteger a rede local de sua instituição ou empresa. Quais pedidos de entrada e saída ele deve interceptar? *páginas 106–107*

3.17 Como um computador pessoal recentemente instalado, conectado a uma rede Ethernet, descobre os endereços IP dos servidores locais? Como ele os transforma em endereços Ethernet? *página 95*

3.18 Os *firewalls* podem evitar ataques de negação de serviço, como aquele descrito na página 96? Quais outros métodos estão disponíveis para tratar desses ataques? *páginas 96–97, 106–107*

Comunicação Entre Processos 4

Este capítulo considera as características dos protocolos para comunicação entre processos em um sistema distribuído e para realizar comunicação entre objetos distribuídos.

A API Java para comunicação entre processos na Internet fornece tanto comunicação por datagrama como por fluxo (*stream*). Suas funcionalidades fornecem os blocos básicos para a construção de protocolos de comunicação. Elas são apresentadas junto com uma discussão sobre seus modelos de falha.

Discutiremos os protocolos para a representação de conjuntos de objetos de dados em mensagens e referências a objetos remotos.

Discutiremos também a construção de protocolos para suportar os dois paradigmas de comunicação comumente usados em programas distribuídos:

- comunicação cliente-servidor – na qual mensagens de requisição e resposta fornecem a base para a invocação a método remoto ou chamada de procedimento remoto;
- comunicação em grupo – onde uma mesma mensagem é enviada para vários processos.

A comunicação entre processos no UNIX é tratada como um estudo de caso.

4.1 Introdução

Este capítulo e o próximo estão relacionados ao *middleware*. Aqui tratamos do projeto dos componentes mostrados na camada mais escura da Figura 4.1. A camada acima será discutida no Capítulo 5; que considera a integração da comunicação em um paradigma de linguagem de programação, fornecendo, por exemplo, invocação a método remoto (RMI – *Remote Method Invocation*) ou chamada de procedimento remoto (RPC – *Remote Procedure Call*). A *invocação a método remoto* permite que um objeto execute um método em um objeto de um processo remoto. Exemplos de sistemas de invocação remota são CORBA e RMI Java. De maneira semelhante, uma *chamada de procedimento remoto* permite que um cliente chame um procedimento em um servidor remoto.

O Capítulo 3 discutiu os protocolos em nível de transporte UDP e TCP da Internet, sem dizer como *middlewares* e aplicativos poderiam utilizar esses protocolos. A próxima seção deste capítulo apresentará as características da comunicação entre processos e depois discutirá os protocolos UDP e TCP do ponto de vista do programador, apresentando a interface Java para cada um desses dois protocolos, junto com uma discussão sobre seus modelos de falha. A última seção deste capítulo apresentará a interface de soquetes UNIX para UDP e TCP, como um estudo de caso.

A interface de programa aplicativo para UDP fornece uma abstração de *passagem de mensagem* – a forma mais simples de comunicação entre processos. Isso permite que um processo remetente transmita uma única mensagem para um processo destino. Os pacotes independentes contendo essas mensagens são chamados de *datagramas*. Nas APIs Java e UNIX, o remetente especifica o destino usando um soquete – uma referência indireta para uma porta em particular usada pelo processo de destino que executa um computador.

A interface de programa aplicativo para TCP fornece a abstração de um *fluxo* (*stream*) bidirecional entre pares de processos. A informação transmitida consiste em um fluxo contínuo de dados sem dar a noção de limites da mensagem, isto é, que ela tem um início e um fim. Os fluxos fornecem um bloco de construção para a comunicação produtor–consumidor [Bacon 2002]. Um produtor e um consumidor formam um par de processos no qual a função do primeiro é produzir itens de dados e do segundo é consumi-los. Os itens de dados enviados pelo produtor para o consumidor são enfileirados na chegada até que o consumidor esteja pronto para recebê-los. O consumidor deve esperar quando nenhum item de dados estiver disponível. O produtor deve esperar, caso o armazenamento usado para conter os itens de dados enfileirados esteja cheio.

A terceira seção deste capítulo se preocupa com o modo como os objetos e as estruturas de dados usados nos programas aplicativos podem ser transformados em uma forma conveniente para envio em mensagens pela rede, levando em consideração o fato de que diferentes computadores podem utilizar diferentes representações para tipos simples de dados. A seção também discutirá uma representação conveniente para referências a objeto em um sistema distribuído.

A quarta e a quinta seções deste capítulo tratarão do projeto de protocolos convenientes para suportar comunicação cliente-servidor e em grupo. Os protocolos de requisição e resposta são proje-

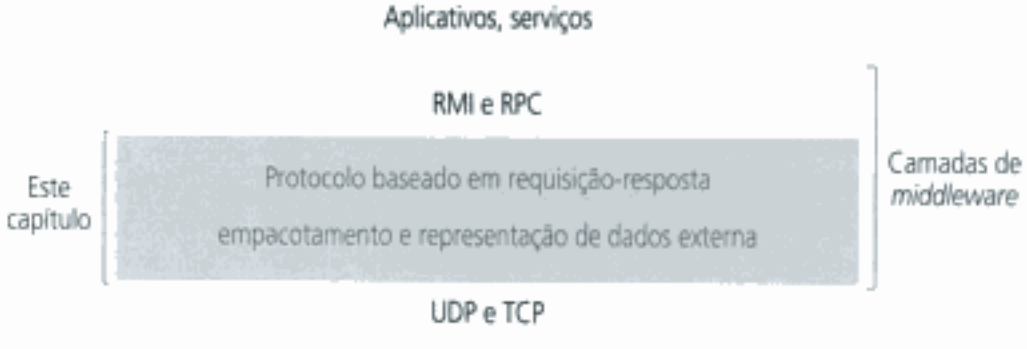


Figura 4.1 Camadas de *middleware*.

tados para suportar comunicação cliente-servidor na forma de RMI ou RPC. Os protocolos *multicast* são projetados para suportar comunicação em grupo. O *multicast* é uma forma de comunicação entre processos na qual um processo de um grupo transmite a mesma mensagem para todos os membros do grupo de processos.

Operações de passagem de mensagem podem ser usadas para construir protocolos para suportar funções de processo e padrões de comunicação em particular, por exemplo, invocações a métodos remotos. Examinando-se as funções e os padrões de comunicação, é possível projetar protocolos de comunicação adequados, baseados em trocas reais, e evitar a redundância. Em particular, esses protocolos especializados não devem incluir confirmações redundantes. Por exemplo, em uma comunicação requisição-resposta, geralmente é considerado redundante confirmar a mensagem de requisição, pois a mensagem de resposta serve automaticamente como uma confirmação de recebimento da primeira. Se um protocolo mais especializado exigir confirmações ou quaisquer outras características particulares, elas serão fornecidas com operações específicas. A idéia é adicionar funções apenas onde elas são necessárias, visando obter protocolos que usem um mínimo de trocas de mensagem.

4.2 A API para protocolos Internet

Nesta seção, discutiremos as características gerais da comunicação entre processos e depois veremos os protocolos Internet como um exemplo, explicando como os programadores podem utilizá-los por meio de mensagens UDP ou por fluxos TCP.

A Seção 4.2.1 revê as operações de comunicação *send* e *receive* apresentadas na Seção 2.3.2, junto com uma discussão sobre como elas são sincronizadas e como os destinos das mensagens são especificados em um sistema distribuído. A Seção 4.2.2 apresenta os *soquetes*, que são empregados na interface para programação de aplicativos baseados em UDP e TCP. A Seção 4.2.3 discute o UDP e sua API em Java. A Seção 4.2.4 discute o TCP e sua API em Java. As APIs Java são orientadas a objetos, mas são semelhantes àquelas projetadas originalmente no sistema operacional Berkeley BSD 4.x UNIX e que serão abordadas na Seção 4.6. Os leitores que estiverem estudando os exemplos de programação desta seção devem consultar a documentação Java on-line, ou Flanagan [2002], para ver a especificação completa das classes discutidas, que estão no pacote *java.net*.

4.2.1 As características da comunicação entre processos

A passagem de mensagens entre um par de processos pode ser suportada por duas operações de comunicação de mensagem: *send* e *receive*, definidas em termos de destinos e mensagens. Para que um processo se comunique com outro, um deles envia (*send*) uma mensagem (uma seqüência de bytes) para um destino e o outro processo, no destino, recebe (*receive*) a mensagem. Essa atividade envolve a comunicação de dados do processo remetente para o processo destino e pode implicar na sincronização dos dois processos. A Seção 4.2.3 dará as definições para as operações *send* e *receive* na API Java para os protocolos Internet.

Comunicação síncrona e assíncrona ◊ Uma fila é associada a cada destino de mensagem. Os processos remetentes fazem as mensagens serem adicionadas em filas remotas e os processos destino removem mensagens de suas filas locais. A comunicação entre os processos remetente e destino pode ser síncrona ou assíncrona. Na forma *síncrona* de comunicação, os processos remetente e destino são sincronizados a cada mensagem. Nesse caso, *send* e *receive* são operações que causam *bloqueio*. Quando um *envio* (*send*) é feito, o processo remetente (ou *thread*) é bloqueado até que a *recepção* (*receive*) correspondente seja realizada. Quando uma *recepção* é executada, o processo (ou *thread*) é bloqueado enquanto a mensagem não chegar.

Na forma *assíncrona* de comunicação, o uso da operação *send* é *não bloqueante*, no sentido de que o processo remetente pode prosseguir assim que a mensagem tenha sido copiada para um buffer local, e a transmissão da mensagem ocorre em paralelo com o processo remetente. A operação *receive* pode ter variantes com e sem bloqueio. Na variante *não bloqueante*, o processo destino prossegue sua execução após ter realizado a operação *receive*, a qual fornece um buffer para ser preenchido em *back-*

ground. Nesse caso, o processo deve receber separadamente uma notificação de que seu buffer possui dados a serem lidos, isso pode ser feito baseado em *polling* ou em interrupção.

Em um ambiente de sistema como o Java, que suporta múltiplas *threads* em um único processo, a *recepção* bloqueante não tem desvantagens, pois ela pode ser executada por uma *thread*, enquanto outras *threads* do processo permanecem ativas; e a simplicidade de sincronização das *threads* destinos com a mensagem recebida é uma vantagem significativa. A comunicação não bloqueante parece ser mais eficiente, mas ela envolve uma complexidade extra no processo destino: a necessidade de ler uma mensagem recebida fora de seu fluxo normal de execução. Por esses motivos, os sistemas atuais geralmente não fornecem a forma de *recepção* não bloqueante.

Destinos de mensagem ◊ O Capítulo 3 explicou que, nos protocolos Internet, as mensagens são enviadas para destinos identificados pelo par (*endereço IP, porta local*). Uma porta local é um destino de mensagem dentro de um computador, especificado como um valor inteiro. Uma porta tem exatamente um destino (as portas de *multicast* são uma exceção, veja a Seção 4.5.1), mas pode ter vários remetentes. Os processos podem usar várias portas para receber mensagens. Qualquer processo que saiba o número de uma porta pode enviar uma mensagem para ela. Geralmente, os servidores divulgam seus números de porta para os clientes acessarem.

Se o cliente usa um endereço IP fixo para se referir a um serviço, então esse serviço sempre deve ser executado no mesmo computador para que seu endereço permaneça válido. Para proporcionar transparência de localização, isso pode ser evitado com o uso de uma das seguintes estratégias:

- Os programas clientes se referem aos serviços pelo nome e usam um servidor de nomes ou de associação (*binder*), veja a Seção 5.2.5, para transformar seus nomes em localizações de servidor no momento da execução. Isso permite que os serviços sejam movidos enquanto o sistema está em execução.
- O sistema operacional, por exemplo, Mach (www.cdk4.net/oss), fornece identificadores independentes da localização para os destinos de mensagem, mapeando-os em um endereço de nível mais baixo para elas serem entregues nas portas, o que possibilita a migração e a movimentação do serviço.

Uma alternativa às portas é o fato de que as mensagens devem ser endereçadas para processos, como o que acontecia no sistema V [Cheriton 1984]. Entretanto, as portas têm a vantagem de fornecer vários pontos alternativos de entrada para um processo destino. Em algumas aplicações, é muito útil poder distribuir a mesma mensagem para os membros de um conjunto de processos. Portanto, alguns mecanismos de IPC (*InterProcess Communication*) têm a capacidade de enviar mensagens para grupos de destinos, sejam eles processos ou portas. Por exemplo, o Chorus [Rozier *et al.* 1990] fornecia grupos de portas.

Confiabilidade ◊ O Capítulo 2 definiu a comunicação confiável em termos de validade e integridade. No que diz respeito à propriedade da validade, um serviço de mensagem ponto a ponto pode ser descrito como confiável se houver garantia que as mensagens forem entregues, independente de um número razoável de pacotes que possam ter sido eliminados ou perdidos. Em contraste, um serviço de mensagem ponto a ponto pode ser descrito como não confiável se não houver garantia de que as mensagens sejam entregues. Quanto à integridade, as mensagens devem chegar não corrompidas e sem duplicação.

Ordenamento ◊ Algumas aplicações exigem que as mensagens sejam entregues na *ordem de emissão* – isto é, na ordem em que foram transmitidas pelo remetente. A entrega de mensagens fora da ordem do remetente é considerada uma falha por tais aplicações.

4.2.2 Soquetes

As duas formas de comunicação (UDP e TCP) usam a abstração de *soquete*, um ponto de destino para a comunicação entre processos. Os soquetes são originários do UNIX BSD, mas também estão presentes na maioria das versões do UNIX, incluindo o Linux, assim como no Windows e no Macintosh

Figura 4.2 **Soquetes e portas.**

OS. A comunicação entre processos consiste em transmitir uma mensagem entre um soquete de um processo e um soquete de outro processo, conforme ilustrado na Figura 4.2. Para que um processo receba mensagens, seu soquete deve estar vinculado a uma porta local e a um dos endereços IP do computador em que é executado. As mensagens enviadas para um endereço IP e um número de porta em particular só podem ser recebidas por um processo cujo soquete esteja associado a esse endereço IP e a esse número de porta. Os processos podem usar o mesmo soquete para enviar e receber mensagens. Cada computador tem 2^{16} números de portas disponíveis para serem usados pelos processos para o envio e recepção de mensagens. Qualquer processo pode fazer uso de várias portas para receber mensagens, mas um processo não pode compartilhar portas com outros processos no mesmo computador. Os processos que usam *multicast* IP são uma exceção, pois eles compartilham portas – veja a Seção 4.5.1. Entretanto, qualquer número de processos pode enviar mensagens para a mesma porta. Cada soquete é associado a um protocolo em particular – UDP ou TCP.

API Java para endereços Internet ◊ A linguagem Java fornece uma classe, *InetAddress*, que representa endereços IP, para permitir a utilização dos protocolos TCP e UDP. Os usuários dessa classe se referem aos computadores pelos nomes de *host* DNS (*Domain Name Service*) (veja a Seção 3.4.7). As instâncias de *InetAddress* que contêm endereços IP podem ser criadas pela chamada ao método estático *InetAddress*, fornecendo-se um nome de *host* DNS como argumento. O método usa o DNS para obter o endereço IP correspondente. Por exemplo, para obter um objeto representando o endereço IP do *host* cujo nome DNS é *bruno.dcs.qmul.ac.uk*, use:

InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk")

Esse método pode disparar a exceção *UnknownHostException*. Note que o usuário da classe não precisa informar o valor explícito de um endereço IP. Na verdade, a classe encapsula os detalhes da representação dos endereços IP. Assim, a interface dessa classe não depende do número de bytes necessários para representar o endereço IP – 4 bytes no IPv4 e 16 bytes no IPv6.

4.2.3 Comunicação por datagrama UDP

Um datagrama enviado pelo protocolo UDP é transmitido de um processo remetente para um processo destino sem a existência de confirmações ou novas tentativas de envio. Se ocorrer uma falha, a mensagem poderá não chegar. Um datagrama é transmitido entre processos quando um deles efetua um *send* e o outro um *receive*. Para enviar ou receber mensagens, um processo precisa primeiro criar uma associação entre um soquete com um endereço IP e com uma porta do *host* local. Um servidor associará seu soquete a uma *porta de serviço* – uma que ele torna conhecida dos clientes para que eles possam enviar mensagens a ela. Um cliente vincula seu soquete a qualquer porta local livre. O método *receive* retorna, além da mensagem, o endereço IP e a porta do remetente permitindo que o destinatário envie uma resposta a este.

Os parágrafos a seguir discutem alguns problemas relacionados à comunicação por datagrama:

Tamanho da mensagem: o processo destino precisa especificar um vetor de bytes de um tamanho em particular para receber as mensagens. Se a mensagem for grande demais para esse vetor, ela

será truncada na chegada. O protocolo IP permite datagramas de até 2^{16} bytes (64 KB), incluindo seu cabeçalho e a área de dados. Entretanto, a maioria dos ambientes impõem uma restrição de tamanho de 8 Kilobytes. Qualquer aplicativo que exija mensagens maiores do que o tamanho máximo deve fragmentá-las em porções desse tamanho. Geralmente, um aplicativo, por exemplo, o DNS, usará um tamanho que não seja excessivamente grande, mas que adequado para o uso pretendido.

Bloqueio: normalmente, os soquetes fornecem operações *send* não bloqueantes e *receive* bloqueantes para comunicação por datagrama (um *receive* não bloqueante é uma opção possível em algumas implementações). A operação *send* retornará quando tiver repassado a mensagem para as camadas UDP e IP subjacentes, que são responsáveis por transmiti-la para seu destino. Ao chegar, a mensagem é posta em uma fila de recepção vinculada ao soquete associado à porta de destino. A mensagem é recuperada dessa fila quando uma operação *receive* for realizada, ou estiver com sua execução pendente, nesse soquete. Se nenhum processo tiver um soquete associado à porta de destino, as mensagens serão descartadas.

O método *receive* bloqueia a execução do processo até que um datagrama seja recebido, a não ser que um tempo de espera limite tenha sido fornecido ao soquete. Se o processo que invoca o método *receive* tiver outra tarefa para fazer enquanto espera pela mensagem, ele deve tomar providências para usar *threads* separadas. As *threads* serão discutidas no Capítulo 6. Por exemplo, quando um servidor recebe uma mensagem de um cliente, normalmente uma tarefa é realizada; se o servidor for implementado usando várias *threads*, elas podem executar essas tarefas enquanto outra *thread* espera pelas mensagens de novos clientes.

Timeouts: a *recepção* bloqueante é conveniente para uso por um servidor que esteja esperando para receber requisições de seus clientes. Mas, em algumas situações, não é adequado que um processo espere indefinidamente para receber algo, pois o processo remetente pode ter falhado ou a mensagem esperada ter se perdido. Para atender a tais requisitos, limites temporais (*timeouts*) podem ser configurados nos soquetes. É difícil escolher um *timeout* apropriado, porém ele deve ser bem grande, em comparação com o tempo exigido para transmitir uma mensagem.

Recepção anônima: o método *receive* não especifica uma origem para as mensagens. A invocação ao método *receive* obtém uma mensagem endereçada para seu soquete, independente da origem. O método *receive* retorna o endereço IP e a porta local do remetente, permitindo que o destinatário verifique de onde ela veio. Entretanto, é possível associar um soquete de datagrama a uma porta remota e a um endereço IP em particular, no caso em que se deseje apenas enviar e receber mensagens desse endereço.

Modelo de falhas ◊ O Capítulo 2 apresentou um modelo de falhas para canais de comunicação e definiu a comunicação confiável em termos de duas propriedades: integridade e validade. A propriedade da integridade exige que as mensagens não devam estar corrompidas nem serem duplicadas. O uso de uma soma de verificação garante que haja uma probabilidade insignificante de que qualquer mensagem recebida esteja corrompida. O modelo de falhas pode ser usado em datagramas UDP, que sofrem das seguintes falhas:

Falhas por omissão: Ocasionalmente, mensagens podem ser descartadas devido a erros de soma de verificação ou porque não há espaço disponível no buffer, na origem ou no destino. Para simplificar a discussão, consideraremos as falhas por omissão de envio e por omissão de recepção (veja a Figura 2.11) como falhas por omissão no canal de comunicação.

Ordenamento: às vezes, as mensagens podem ser entregues em uma ordem diferente da que foram emitidas.

Os aplicativos que usam datagramas UDP podem efetuar seus próprios controles para atingir a qualidade de comunicação confiável que suas finalidades exigem. Um serviço de entrega confiável pode ser construído a partir de outro que sofra de falhas por omissão, pelo uso de confirmações. A Seção 4.4 discutirá como protocolos requisição-resposta confiáveis para comunicação cliente-servidor podem ser construídos sobre UDP.

Emprego de UDP ◊ Para algumas aplicações, é aceitável usar um serviço que esteja exposto a falhas por omissão ocasionais. Por exemplo, o *Domain Name Service*, que pesquisa nomes DNS na Internet, é implementado sobre UDP. O *Voice Over IP* (VOIP) também é executado sobre UDP. Às vezes, os datagramas UDP são uma escolha atraente, pois eles não sofrem as sobrecargas necessárias a entrega de mensagens garantida. Existem três fontes de sobrecarga principais:

1. a necessidade de armazenar informações de estado na origem e no destino;
2. a transmissão de mensagens extras; e
3. a latência do remetente.

Os motivos dessas sobrecargas serão discutidos na Seção 4.2.4.

API Java para datagramas UDP ◊ A API Java fornece comunicação por datagrama por meio de duas classes: *DatagramPacket* e *DatagramSocket*.

DatagramPacket: esta classe fornece um construtor para uma instância composta por um vetor de bytes (mensagem), o comprimento da mensagem, o endereço IP e o número da porta local do soquete de destino, como segue:

Pacote de datagrama

vetor de bytes contendo a mensagem	comprimento da mensagem	endereço IP	número da porta

Instâncias de *DatagramPacket* podem ser transmitidas entre processos quando um processo realiza uma operação de *send* e outro de *receive*.

Essa classe fornece outro construtor para ser usado na recepção de mensagens. Seus argumentos especificam um vetor de bytes, para armazenamento da mensagem a ser recebida, e seu comprimento. Uma mensagem recebida é colocada no *DatagramPacket*, junto com seu comprimento e o endereço IP e a porta do soquete remetente. A mensagem pode ser recuperada do *DatagramPacket* por meio do método *getData*. Os métodos *getPort* e *getAddress* acessam a porta e o endereço IP.

DatagramSocket: esta classe oferece mecanismos para criação de soquetes para envio e recepção de datagramas UDP. Ela fornece um construtor que recebe como argumento um número de porta, para os processos que precisam utilizar uma porta em particular, e um construtor sem argumentos que permite a obtenção dinâmica de um número de porta. Esses construtores podem provocar uma exceção *SocketException*, caso a porta já esteja em uso ou se, em ambientes UNIX, for especificada uma porta reservada (um número abaixo de 1024).

A classe *DatagramSocket* fornece métodos que incluem os seguintes:

send e *receive*: esses métodos servem para transmitir datagramas entre dois soquetes. O argumento de *send* é uma instância de *DatagramPacket* contendo uma mensagem e seu destino. O argumento de *receive* é um *DatagramPacket* vazio para se receber a mensagem, seu comprimento e origem. Os métodos *send* e *receive* podem causar exceções *IOException*.

setSoTimeout: este método permite o estabelecimento de um *timeout*. Com um *timeout* configurado, o método *receive* bloqueará pelo tempo especificado e depois causará uma exceção *InterruptedIOException*.

connect: este método é usado para contactar uma porta remota e um endereço IP em particular, no caso em que o soquete é capaz apenas de enviar e receber mensagens desse endereço.

A Figura 4.3 mostra o programa de um cliente que cria um soquete, envia uma mensagem para um servidor na porta 6789 e depois espera para receber uma resposta. Os argumentos do método *main* fornecem uma mensagem e o nome de *host* DNS do servidor. A mensagem é convertida em um vetor de bytes e o nome de *host* DNS é convertido em um endereço IP. A Figura 4.4 mostra o programa do servidor correspondente, o qual cria um soquete vinculado a porta de serviço (6789) e depois, em um laço, espera pelo recebimento de uma mensagem e responde enviando-a de volta.

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        // args fornece o conteúdo da mensagem e o nome de host do servidor
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                new DatagramPacket(m, args[0].length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        } catch (IOException e){System.out.println("IO: " + e.getMessage());
        } finally { if(aSocket != null) aSocket.close();}
    }
}
```

Figura 4.3 O cliente UDP envia uma mensagem para o servidor e obtém uma resposta.

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        } catch (IOException e) {System.out.println("IO: " + e.getMessage());
        } finally {if (aSocket != null) aSocket.close();}
        }
}
```

Figura 4.4 O servidor UDP recebe uma mensagem e a envia de volta para o cliente.

4.2.4 Comunicação por fluxo TCP

A API do protocolo TCP, que se originou do UNIX BSD 4.x, fornece a abstração de um fluxo de bytes no qual dados podem ser lidos (*receive*) e escritos (*send*). As seguintes características da rede são ocultas pela abstração de fluxo (*stream*):

Tamanho das mensagens: o aplicativo pode escolher o volume de dados que vai ser enviado ou recebido em um fluxo. Ele pode tratar com conjuntos de dados muito pequenos ou muito grandes. A implementação da camada TCP decide o volume de dados a coletar, antes de transmiti-los efetivamente como um ou mais pacotes IP. Ao chegar, os dados são entregues para o aplicativo, conforme solicitado. Se necessário, os aplicativos podem obrigar os dados a serem enviados imediatamente.

Mensagens perdidas: o protocolo TCP usa um esquema de confirmação. Como um simples exemplo desses esquemas (não é o usado no TCP), o lado remetente mantém um registro de cada datagrama IP enviado e o lado destino confirma todas as chegadas. Se o remetente não receber uma confirmação dentro de um tempo limite, ele retransmite a mensagem. O esquema de janela deslizante [Comer 2000a], mais sofisticado, reduz o número de mensagens de confirmação exigidas.

Controle de fluxo: o protocolo TCP tenta combinar a velocidade dos processos que lêem e escrevem em um fluxo. Se o processo que escreve (envia) for rápido demais para o que lê, então ele será bloqueado até que o leitor tenha consumido dados suficientes.

Duplicação e ordenamento de mensagens: identificadores de mensagem são associados a cada datagrama IP, o que permite ao destinatário detectar e rejeitar duplicatas ou reordenar as mensagens que chegam fora da ordem de emissão.

Destinos de mensagem: dois processos que estão em comunicação estabelecem uma conexão antes de poderem se comunicar por meio de um fluxo. Uma vez estabelecida a conexão, os processos simplesmente lêem ou escrevem no fluxo, sem necessidade de usar endereços IP e portas. O estabelecimento de uma conexão envolve uma requisição de *connect*, do cliente para o servidor, seguido de uma requisição de *accept*, do servidor para o cliente, antes que qualquer comunicação possa ocorrer. Em um modelo cliente-servidor, isso causa uma sobrecarga considerável para cada requisição-resposta.

A API para comunicação por fluxo pressupõe que, quando dois processos estão estabelecendo uma conexão, um deles desempenha o papel de cliente e o outro desempenha o papel de servidor, mas daí em diante eles poderiam ser iguais. O papel de cliente envolve a criação de um soquete de fluxo vinculado a qualquer porta e depois um pedido *connect* solicitando uma conexão a um servidor, em uma determinada porta. O papel de servidor envolve a criação de um soquete de "escuta", vinculado a porta de serviço, para esperar que os clientes solicitem conexões. O soquete de "escuta" mantém uma fila de pedidos de conexão recebidos. Na abstração de soquete, quando o servidor aceita uma conexão, um novo soquete de fluxo é criado para que o servidor se comunique com um cliente, mantendo nesse meio-tempo seu soquete na porta de serviço para receber os pedidos *connect* de outros clientes. Mais detalhes sobre as operações *connect* e *accept* serão descritos no estudo de caso do UNIX, no final deste capítulo.

O par de soquetes no cliente e no servidor são, na realidade, conectados por dois fluxos, um em cada direção. Assim, cada soquete tem um fluxo de entrada e um fluxo de saída. Um dos dois processos pode enviar informações para o outro, escrevendo em seu fluxo de saída, e o outro processo obtém as informações lendo seu fluxo de entrada.

Quando um aplicativo *encerra* (operação *close*) um soquete, isso indica que ele não escreverá mais nenhum dado em seu fluxo de saída. Os dados de seu buffer de saída são enviados para o outro lado do fluxo e colocados na fila de entrada do soquete de destino com uma indicação de que o fluxo está desfeito. O processo no destino pode ler os dados da fila, mas todas as outras leituras depois que a fila estiver vazia resultarão em uma indicação de fim de fluxo. Quando um processo termina, ou falha, todos os seus soquetes são encerrados e qualquer processo que tente se comunicar com ele descobrirá que sua conexão foi desfeita.

Os parágrafos a seguir tratam de alguns problemas importantes relacionados à comunicação por fluxo:

Correspondência de itens de dados: dois processos que estejam se comunicando precisam concordar quanto ao conteúdo dos dados transmitidos por um fluxo. Por exemplo, se um processo escreve (envia) um valor *int* em um fluxo, seguido de um valor *double*, então o outro lado deverá ler um valor *int*, seguido de um valor *double*. Quando dois processos não cooperam corretamente no uso de um fluxo, o processo leitor pode causar erros ao interpretar os dados, ou ser bloqueado, devido a dados insuficientes no fluxo.

Bloqueio: os dados gravados em um fluxo são mantidos em uma fila no soquete de destino. Quando um processo tentar ler dados de um canal de entrada, obterá dados da fila ou será bloqueado até que dados se tornem disponíveis. O processo que escreve dados em um fluxo pode ser bloqueado pelo mecanismo de controle de fluxo TCP, caso o soquete no outro lado já esteja armazenando o volume máximo de dados permitido pelo protocolo.

Threads: quando um servidor aceita uma conexão, geralmente ele cria uma nova *thread* para se comunicar com o novo cliente. A vantagem de usar uma *thread* separada para cada cliente é que o servidor pode bloquear quando estiver esperando por dados, sem atrasar os outros clientes. Em um ambiente em que *threads* não são suportadas, uma alternativa é testar, antes de tentar lê-lo, se a entrada está disponível; por exemplo, em um ambiente UNIX, a chamada de sistema *select* pode ser usada para esse propósito.

Modelo de falhas ◊ Para satisfazer a propriedade da integridade da comunicação confiável, os fluxos TCP usam somas de verificação para detectar e rejeitar pacotes corrompidos, assim como números de seqüência para detectar e rejeitar pacotes duplicados. Quanto à propriedade da validade, os fluxos TCP usam *timeout* e retransmissões para tratar com pacotes perdidos. Portanto, há garantia de que as mensagens sejam entregues, mesmo quando alguns dos pacotes das camadas inferiores são perdidos.

Mas se a perda de pacote em uma conexão ultrapassar um limite, ou se a rede que está conectando dois processos for rompida ou se tornar seriamente congestionada, o software TCP responsável pelo envio de mensagens não receberá nenhum tipo de confirmação e, após certo tempo, declarará que a conexão está desfeita. Assim, o protocolo TCP não fornece comunicação confiável, pois não garante a entrega de mensagens diante de todas as dificuldades possíveis.

Quando uma conexão é desfeita, um processo ao tentar ler ou escrever algo nela receberá uma notificação de erro. Isso tem os seguintes efeitos:

- os processos que estão usando a conexão não poderão distinguir entre falha de rede e falha do processo no outro lado da conexão;
- os processos que estão se comunicando não poderão identificar se as mensagens que enviaram recentemente foram recebidas ou não.

Emprego de TCP ◊ Muitos serviços freqüentemente usados são executados em conexões TCP com números de porta reservados. Eles incluem os seguintes:

HTTP: o protocolo de transferência de hipertexto é usado para comunicação entre navegadores e servidores web; ele será discutido posteriormente neste capítulo.

FTP: o protocolo de transferência de arquivos permite a navegação em diretórios em um computador remoto, e que arquivos sejam transferidos de um computador para outro por meio de uma conexão.

Telnet: o serviço telnet dá acesso a um computador remoto por meio de uma sessão de terminal.

SMTP: o protocolo de transferência de correio eletrônico usado para enviar correspondência entre computadores.

API Java para fluxos TCP ◊ A interface Java para fluxos TCP é fornecida pelas classes *ServerSocket* e *Socket*.

ServerSocket: esta classe se destina a ser usada por um servidor para criar um soquete em uma porta de serviço para receber requisições de *connect* dos clientes. Seu método *accept* recupera um pedido *connect* da fila ou, se a fila estiver vazia, bloqueia até que chegue um. O resultado da execução de *accept* é uma instância de *Socket* – um soquete para dar acesso aos fluxos para comunicação com o cliente.

Socket: esta classe é usada pelos dois processos de uma conexão. O cliente usa um construtor para criar um soquete, especificando o nome de *host* DNS e a porta do servidor. Esse construtor não apenas cria um soquete associado a uma porta local, mas também o conecta com o computador remoto e com o número de porta especificado. Ele pode causar uma exceção *UnknownHostException*, caso o nome de *host* esteja errado, ou uma exceção *IOException*, caso ocorra um erro de E/S.

A classe *Socket* fornece os métodos *getInputStream* e *getOutputStream* para acessar os dois fluxos associados a um soquete. Os tipos de retorno desses métodos são *InputStream* e *OutputStream*, respectivamente – classes abstratas que definem métodos para ler e escrever os bytes. Os valores de retorno podem ser usados como argumentos de construtores para fluxos de entrada e saída. Nosso exemplo usa *DataInputStream* e *DataOutputStream*, que permitem representações binárias de tipos de dados primitivos serem lidas e escritas de forma independente de máquina.

A Figura 4.5 mostra um programa cliente no qual os argumentos do método *main* fornecem uma mensagem e o nome de *host* DNS do servidor. O cliente cria um soquete vinculado ao nome de *host* e à porta de serviço 7896. Ele produz um *DataInputStream* e um *DataOutputStream* a partir dos fluxos de entrada e saída do soquete e, em seguida, escreve a mensagem em seu fluxo de saída e espera para ler uma resposta em seu fluxo de entrada. O programa servidor da Figura 4.6 abre um soquete em sua porta de "escuta" (7896) e recebe os pedidos *connect*. A cada pedido que chega, uma nova *thread* é criada para se comunicar com o cliente. A *thread* cria um *DataInputStream* e um *DataOutputStream* a partir dos fluxos de entrada e saída de seu soquete e, em seguida, espera para ler uma mensagem e enviá-la de volta.

Como nossa mensagem consiste em uma cadeia de caracteres (*string*), os processos cliente e servidor usam o método *writeUTF* de *DataOutputStream* para escrevê-la no fluxo de saída e o método *readUTF* de *DataInputStream* para lê-la do fluxo de entrada. UTF-8 é uma codificação que representa strings em um formato específico, que será descrito na Seção 4.3.

Quando um processo tiver encerrado (*close*) seu soquete, não poderá mais usar seus fluxos de entrada e saída. O processo para o qual ele tiver enviado dados ainda poderá lê-los em sua fila, mas

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // os argumentos fornecem a mensagem e o nome de host do destino
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =
                new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]); // UTF é uma codificação de string; veja a Seção 4.3
            String data = in.readUTF();
            System.out.println("Received: "+ data);
        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        } catch (IOException e){System.out.println("IO:"+e.getMessage());
        } finally {if(s!=null) try {s.close();}catch (IOException e){/*close falhou*/}}
    }
}
```

Figura 4.5 O cliente TCP estabelece uma conexão com o servidor, envia uma requisição e recebe uma resposta.

```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen:"+e.getMessage());}
    }
}
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream( clientSocket.getInputStream());
            out =new DataOutputStream( clientSocket.getOutputStream());
            this.start();
        } catch(IOException e) {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
        try {                    // Servidor de eco
            String data = in.readUTF();
            out.writeUTF(data);
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());
        } finally { try {clientSocket.close();}catch (IOException e){/*close falhou*/}}
    }
}
```

Figura 4.6 O servidor TCP estabelece uma conexão para cada cliente e, em seguida, ecoa o pedido do cliente.

as leituras feitas após essa fila ficar vazia resultarão em uma exceção *EOFException*. As tentativas de usar um soquete já encerrado, ou de escrever em um fluxo desfeito resultarão em uma exceção *IOException*.

4.3 Representação externa de dados e empacotamento

As informações armazenadas nos programas em execução são representadas como estruturas de dados – por exemplo, pela associação de um conjunto de objetos –, enquanto que as informações presentes nas mensagens são seqüências puras de bytes. Independente da forma de comunicação usada, as estruturas de dados devem ser simplificadas (convertidas em uma seqüência de bytes) antes da transmissão e reconstruídas na sua chegada. Os dados transmitidos nas mensagens podem corresponder a valores de tipos de dados primitivos diferentes e nem todos os computadores armazenam tipos de

dados primitivos, como os inteiros, na mesma ordem. A representação interna de números em ponto flutuante também difere entre as arquiteturas de processadores. Existem duas variantes para a ordenação de inteiros: ordem *big-endian*, na qual o byte mais significativo aparece na primeira posição, e a ordem *little-endian*, na qual ele aparece por último. Outro problema é o conjunto de códigos usado para representar caracteres: por exemplo, a maioria dos aplicativos em sistemas como o UNIX usa codificação de caracteres ASCII, com um byte por caractere, enquanto o padrão Unicode permite a representação de textos em muitos idiomas diferentes e usa dois bytes por caractere.

Um dos métodos a seguir pode ser usado para permitir que dois computadores troquem valores de dados binários:

- Os valores são convertidos para um formato externo, acordado antes da transmissão e convertidos para a forma local, na recepção; se for sabido que os dois computadores são do mesmo tipo, a conversão para o formato externo pode ser omitida.
- Os valores são transmitidos no formato do remetente, junto com uma indicação do formato usado, e o destinatário converte os valores, se necessário.

Note, entretanto, que os bytes em si nunca tem a ordem de seus bits alterados durante a transmissão. Para suportar RMI ou RPC, todo tipo de dados que possa ser passado como argumento, ou retornado como resultado, deve ser simplificado e os valores de dados primitivos individuais, representados em um formato comum. Um padrão aceito para a representação de estruturas de dados e valores primitivos é chamado de *representação externa de dados*.

Empacotamento (*marshalling*) é o procedimento de pegar um conjunto de itens de dados e montá-los em uma forma conveniente para transmissão em uma mensagem. *Desempacotamento* (*unmarshalling*) é o procedimento inverso de desmontá-los na chegada para produzir um conjunto de itens de dados equivalente no destino. Assim, o empacotamento consiste na transformação de itens de dados estruturados e valores primitivos em uma representação externa de dados. Analogamente, o desempacotamento consiste na geração de valores primitivos a partir de sua representação externa de dados e na reconstrução das estruturas de dados.

Serão discutidas três estratégias alternativas para representação externa de dados e empacotamento:

- a representação de dados comum do CORBA, que está relacionada a uma representação externa dos tipos estruturados e primitivos que podem ser passados como argumentos e resultados na invocação a métodos remotos no CORBA. Ela pode ser usada por diversas linguagens de programação (veja o Capítulo 20).
- serialização de objetos da linguagem Java, que está relacionada à simplificação e à representação externa de dados de um objeto, ou de uma árvore de objetos, que precise ser transmitida em uma mensagem ou armazenada em um disco. Isso é usado apenas pela linguagem Java.
- XML ou *Extensible Markup Language*, que define um formato textual para representar dados estruturados. Ela se destinava originalmente a documentos contendo dados estruturados textuais auto-descritivos; por exemplo, documentos web. Mas agora também é usada para representar dados enviados em mensagens trocadas por clientes e servidores em serviços web – veja o Capítulo 19.

Nos dois primeiros casos, as atividades de empacotamento e desempacotamento se destinam a serem executadas por uma camada de *middleware*, sem nenhum envolvimento por parte do programador de aplicativo. Mesmo no caso da XML, que é textual e, portanto, mais acessível para tratar de codificação, o software para empacotar e desempacotar está disponível para praticamente todas as plataformas e ambientes de programação comumente usados. Como o empacotamento exige a consideração de todos os mínimos detalhes da representação dos componentes primitivos de objetos compostos, esse procedimento é bastante propenso a erros se executado manualmente. A compactação é outro problema que pode ser tratado no projeto de procedimentos de empacotamento gerados automaticamente.

Nas duas primeiras estratégias, os tipos de dados primitivos são empacotados em uma forma binária. Na terceira estratégia (XML), os tipos de dados primitivos são representados textualmente. A representação textual de um valor de dados geralmente será maior do que a representação binária equivalente. O protocolo HTTP, que será descrito na Seção 4.4, é outro exemplo de estratégia textual.

Outro problema com relação ao projeto de métodos de empacotamento é se os dados empacotados devem incluir informações relativas ao tipo de seu conteúdo. Por exemplo, a representação usada pelo CORBA inclui apenas os valores dos objetos transmitidos – nada a respeito de seus tipos. Por outro lado, tanto a serialização Java, como a do XML, incluem informações sobre o tipo, mas de maneiras diferentes. A linguagem Java coloca todas as informações de tipo exigidas na forma serializada. Mas os documentos XML podem se referir a conjuntos de nomes (com tipos) definidos externamente, chamados *espaços de nomes*.

Embora estejamos interessados no uso de representação externa de dados para os argumentos e resultados de RMIs e de RPCs, ela tem um uso mais genérico quando é empregada para representar estruturas de dados, objetos ou documentos estruturados em uma forma conveniente para transmissão em mensagens ou armazenamento em arquivos.

4.3.1 Representação comum de dados (CDR) do CORBA

O CDR do CORBA é a representação externa de dados definida no CORBA 2.0 [OMG 2004a]. O CDR pode representar todos os tipos de dados que são como argumentos e valores de retorno em invocações a métodos remotos no CORBA. Eles consistem em 15 tipos primitivos, os quais incluem *short* (16 bits), *long* (32 bits), *unsigned short*, *unsigned long*, *float* (32 bits), *double* (64 bits), *char*, *boolean* (TRUE, FALSE), *octet* (8 bits) e *any* (que pode representar qualquer tipo primitivo ou construído), junto com uma variedade de tipos compostos. Eles estão descritos na Figura 4.7. Cada argumento ou resultado em uma invocação remota é representado por uma seqüência de bytes na mensagem de invocação ou resultado.

Tipos primitivos: o CDR define uma representação para as ordens *big-endian* e *little-endian*. Os valores são transmitidos na ordem do remetente, que é especificada em cada mensagem. Se exigir uma ordem diferente, o destinatário a transforma. Por exemplo, um valor *short* de 16 bits ocupa dois bytes na mensagem e, para a ordem *big-endian*, os bits mais significativos ocupam o primeiro byte e os bits menos significativos ocupam o segundo byte. Cada valor primitivo é colocado em um índice na seqüência de bytes, de acordo com seu tamanho. Suponha que a seqüência de bytes seja indexada a partir de zero. Então, um valor primitivo com tamanho de n bytes (onde $n = 1, 2, 4$ ou 8) é anexado à seqüência, em um índice que é um múltiplo de n no fluxo de bytes. Os valores em ponto flutuante seguem o padrão IEEE – no qual o sinal, o expoente e a parte fracionária estão nos bytes 0–n para a ordem *big-endian* e ao contrário na ordem *little-endian*. Os caracteres são representados por um código acordado entre cliente e servidor.

Tipos construídos ou *compostos*: os valores primitivos que compreendem cada tipo construído são adicionados a uma seqüência de bytes, em uma ordem específica, como se vê na Figura 4.7.

Tipo	*Representação*
sequence	comprimento (*unsigned long*) seguido de seus elementos, em ordem
string	comprimento (*unsigned long*) seguido pelos caracteres que o compõe (um caractere pode ocupar mais de um byte)
array	elementos de vetor, fornecidos em ordem (nenhum comprimento especificado, pois é fixo)
struct	na ordem da declaração dos componentes
enumerated	*unsigned long* (os valores são especificados pela ordem declarada)
union	identificador de tipo seguido do membro selecionado

Figura 4.7 CDR do CORBA para tipos construídos.

A Figura 4.8 mostra uma mensagem no CDR do CORBA contendo três campos de um *struct* cujos tipos respectivos são *string*, *string* e *unsigned long*. A figura mostra a seqüência de bytes, com quatro bytes em cada linha. A representação de cada *string* consiste em um valor *unsigned long*, dando seu comprimento, seguido dos caracteres do *string*. Por simplicidade, presumimos que cada caractere ocupa apenas um byte. Os dados de comprimento variável são preenchidos com zero para que tenha uma forma padrão para permitir a comparação de dados empacotados ou de sua soma de verificação. Note que cada valor *unsigned long*, que ocupa quatro bytes, começa em um índice que é múltiplo de quatro. A figura não distingue entre as ordens *big-endian* e *little-endian*. Embora o exemplo da Figura 4.8 seja simples, o CDR do CORBA pode representar qualquer estrutura de dados composta por tipos primitivos e construídos, mas sem usar ponteiros.

Outro exemplo de representação de dados externa é o padrão XDR da Sun, que está especificado no RFC 1832 [Srinivasan 1995b] e é descrito em www.cdk4.net/ipc. Ele foi desenvolvido pela Sun para uso nas mensagens trocadas entre clientes e servidores NFS (veja o Capítulo 8).

O tipo de um item de dados não é fornecido com a representação de dados na mensagem, seja no CDR do CORBA ou no padrão XDR da Sun. Isso porque pressupõe-se que o remetente e o destinatário tenham conhecimento comum da ordem e dos tipos dos itens de dados de uma mensagem. Em particular para RMI, ou para RPC, cada invocação de método passa argumentos de tipos específicos e o resultado é um valor de um tipo em particular.

Empacotamento no CORBA ◊ As operações de empacotamento (*marshalling*) podem ser geradas automaticamente a partir da especificação dos tipos dos itens de dados a serem transmitidos em uma mensagem. Os tipos das estruturas de dados e os tipos dos itens de dados básicos estão descritos no IDL (*Interface Definition Language*) do CORBA (veja a Seção 20.2.3), que fornece uma notação para descrever os tipos dos argumentos e resultados dos métodos RMI. Por exemplo, poderíamos usar o IDL do CORBA para descrever a estrutura de dados na mensagem da Figura 4.8, como segue:

```
struct Person{
    string name;
    string place;
    unsigned long year;
};
```

O compilador da interface CORBA (veja o Capítulo 5) gera as operações de empacotamento e desempacotamento apropriadas para os argumentos e resultados dos métodos remotos, a partir das definições dos tipos de seus parâmetros e resultados.

Índice na seqüência de bytes	← 4 bytes →	*Observações sobre a representação*
0–3	5	*Comprimento do string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*Comprimento do string*
16–19	"Lond"	*'London'*
20–23	"on__"	
24–27	1934	*Unsigned long*

A forma simplificada representa uma *struct Person* com o valor: {'Smith', 'London', 1934}

Figura 4.8 **Mensagem no CDR do CORBA.**

4.3.2 Serialização de objeto Java

No Java RMI, tanto objetos como valores de dados primitivos podem ser passados como argumentos e resultados de invocações de método. Um objeto é uma instância de uma classe Java. Por exemplo, a classe Java equivalente a *struct Person* definida no IDL do CORBA poderia ser:

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }
    // seguido dos métodos para acessar as variáveis de instância
}
```

Essa classe diz que implementa a interface *Serializable*, a qual não tem métodos. Dizer que uma classe implementa a interface *Serializable* (que é fornecida no pacote *java.io*) tem o efeito de permitir que suas instâncias sejam serializadas.

Em Java, o termo *serialização* se refere à atividade de simplificar um objeto, ou um conjunto de objetos conectados, em uma forma seqüencial conveniente para ser armazenada em disco ou transmitida em uma mensagem; por exemplo, como um argumento ou resultado de uma RMI. A *desserialização* consiste em restaurar o estado de um objeto ou conjunto de objetos a partir de sua forma serializada. Pressupõe-se que o processo que faz a desserialização não tenha nenhum conhecimento anterior dos tipos dos objetos na forma serializada. Portanto, qualquer informação sobre a classe de cada objeto é incluída na forma serializada. Essa informação permite que o destinatário carregue a classe apropriada quando um objeto é desserializado.

A informação sobre uma classe consiste em seu nome e em um número de versão. O número da versão deve mudar quando forem feitas alterações na classe. Ele pode ser estabelecido pelo programador, ou calculado automaticamente como uma mistura do nome da classe, suas variáveis de instância, métodos e interfaces. O processo que desserializa um objeto pode verificar se ele tem a versão correta da classe.

Os objetos Java podem conter referências para outros objetos. Quando um objeto é serializado, todos os objetos que ele referencia são serializados junto, para garantir que, quando o objeto for reconstruído, todas as suas referências possam ser completadas no destino. As referências são serializadas através de *identificadores* (*handlers*) – neste caso, o *identificador* é uma referência a um objeto dentro da forma serializada; por exemplo, o próximo número em uma seqüência de valores inteiros positivos. O procedimento de serialização deve garantir que exista uma correspondência biunívoca entre referências de objeto e seus identificadores. Ele também deve garantir que cada objeto seja gravado apenas uma vez – na segunda ocorrência de um objeto, ou em ocorrências subseqüentes, é gravado o identificador, em vez do objeto.

Para serializar um objeto, a informação de sua classe é escrita por extenso, seguida dos tipos e nomes de suas variáveis de instância. Se as variáveis de instância pertencerem a novas classes, então suas informações de classe também deverão ser escritas por extenso, seguidas dos tipos e nomes de suas variáveis de instância. Esse procedimento recursivo continua até que a informação da classe, e os tipos e nomes das variáveis de instância de todas as classes necessárias, tenham sido escritas por extenso. Cada classe recebe um identificador (*handle*) e nenhuma classe é gravada mais do que uma vez no fluxo de bytes – os identificadores são gravados em seu lugar, onde for necessário.

O conteúdo das variáveis de instância que são tipos primitivos, como inteiros, caracteres, booleanos, bytes e longos, são gravados em um formato binário portável, usando métodos da classe *ObjectOutputStream*. Os strings e os caracteres são gravados pelo método *writeUTF*, usando o formato Universal Transfer Format (UTF-8), o qual permite que caracteres ASCII sejam representados de forma inalterada (em um byte), enquanto os caracteres Unicode são representados por vários bytes. Os strings são precedidos pelo número de bytes que ocupam no fluxo.

Como exemplo, considere a serialização do objeto a seguir:

Person p = new Person("Smith", "London", 1934);

A forma serializada está ilustrada na Figura 4.9, que omite os valores dos identificadores (*handlers*) e das informações de tipo que indicam objetos, classes, strings e outros recursos na forma serializada completa. A primeira variável de instância (1934) é um valor inteiro de comprimento fixo; a segunda e a terceira variáveis de instância são strings e são precedidas por seus comprimentos.

Para fazer uso de serialização Java, por exemplo, para serializar o objeto *Person*, é necessário criar uma instância da classe *ObjectOutputStream* e invocar seu método *writeObject*, passando o objeto *Person* como argumento. Para desserializar um objeto de um fluxo de dados, é necessário abrir o fluxo como *ObjectInputStream* e utilizar o método *readObject* para reconstruir o objeto original. O uso dessas duas classes é semelhante ao uso de *DataOutputStream* e *DataInputStream*, ilustrado nas Figuras 4.5 e 4.6.

A serialização e desserialização dos argumentos e resultados de invocações remotas geralmente são executados automaticamente pela camada de middeware, sem nenhuma participação do programador do aplicativo. Se necessário, os programadores que tiverem requisitos especiais podem fazer sua própria versão dos métodos que lêem e escrevem objetos. Para descobrir como fazer isso e para obter mais informações sobre serialização em Java, leia o exercício dirigido sobre serialização de objetos [java.sun.com II]. Outra maneira pela qual um programador pode modificar os efeitos da serialização é declarando as variáveis que não devem ser serializadas como *transientes*. Exemplos de variáveis que não devem ser serializadas são referências a recursos locais, como arquivos e soquetes.

O uso de reflexão ◊ A linguagem Java suporta *reflexão* – a capacidade de fazer perguntas sobre as propriedades de uma classe, como os nomes e tipos de suas variáveis de instância e métodos. Isso também permite que classes sejam criadas a partir de seus nomes e que seja criado para uma determinada classe um construtor com argumentos de determinados tipos de dados. A reflexão torna possível fazer serialização e desserialização de maneira completamente genérica. Isso significa que não há necessidade de gerar funções de empacotamento especiais para cada tipo de objeto, conforme descrito anteriormente para CORBA. Para saber mais sobre reflexão, veja Flanagan [2002].

A serialização de objetos Java usa reflexão para descobrir o nome da classe do objeto a ser serializado e os nomes, tipos e valores de suas variáveis de instância. Isso é tudo que é necessário para a forma serializada.

Para a desserialização, o nome da classe na forma serializada é usado para criar uma classe. Isso é usado então para criar um novo construtor, com tipos de argumento correspondentes àqueles especificados na forma serializada. Finalmente, o novo construtor é usado para criar um novo objeto, com variáveis de instância cujos valores são lidos da forma serializada.

	Valores serializados			*Explicação*
Person	Número da versão de 8 bytes		h0	*Nome da classe, número da versão*
3	int year	java.lang.String name	java.lang.String place	*Número, tipo de nome das variáveis de instância*
1934	5 Smith	6 London	h1	*Valores das variáveis de instância*

Na realidade, a forma serializada inclui marcas adicionais de tipos; h0 e h1 são identificadores.

Figura 4.9 Indicação da forma serializada Java.

4.3.3 XML (*Extensible Markup Language*)

A XML é uma linguagem de marcação que foi definida pelo World Wide Web Consortium (W3C) para uso na web. Em geral, o termo *linguagem de marcação* se refere a uma codificação textual que representa um texto e os detalhes de sua estrutura ou de sua aparência. Tanto a XML como a HTML foram derivadas da SGML (*Standardized Generalized Markup Language*) [ISO 8879], uma linguagem de marcação muito complexa. A HTML (veja a Seção 1.3.1) foi projetada para definir a aparência de páginas web. A XML foi projetada para elaborar documentos estruturados para a web.

Os itens de dados XML são rotulados com strings de marcação (*tags*). As *tags* são usadas para descrever a estrutura lógica dos dados e para associar pares atributo-valor às estruturas lógicas. Isto é, na XML, as *tags* estão relacionadas à estrutura do texto que englobam, em contraste com a HTML, na qual as *tags* especificam como um navegador poderia exibir o texto. Para ver uma especificação da XML, consulte as páginas sobre XML fornecidas pelo W3C, no endereço [www.w3.org VI].

A XML é usada para permitir que clientes se comuniquem com serviços web e para definir as interfaces e outras propriedades desses mesmos serviços. Entretanto, a XML também é usada de muitas outras maneiras. Ela é utilizada no arquivamento e na recuperação de sistemas – embora um repositório de arquivos XML possa ser maior do que seu equivalente binário, ele tem a vantagem de poder ser lido em qualquer computador. Outros exemplos do uso da XML incluem a especificação de interfaces com o usuário e a codificação de arquivos de configuração em sistemas operacionais.

A XML é *extensível*, pois os usuários podem definir suas próprias *tags*, em contraste com a HTML, que usa um conjunto fixo de *tags*. Entretanto, se um documento XML se destina a ser usado por mais de um aplicativo, os nomes das *tags* devem ser combinados entre eles. Por exemplo, os clientes normalmente usam mensagens SOAP para se comunicar com serviços web. O SOAP (veja a Seção 19.2.1) é um formato XML cujas *tags* são publicadas para serem usadas pelos serviços web e seus clientes.

Algumas representações externas de dados (como o CDR do CORBA) não precisam ser autodescritivas, pois pressupõe-se que o cliente e o servidor que estão trocando uma mensagem têm conhecimento anterior da ordem e dos tipos das informações que ela contém. Entretanto, a XML foi projetada para ser usada por vários aplicativos, para diferentes propósitos. A capacidade de prover *tags*, junto com o uso de espaços de nomes para definir o significado das próprias *tags*, tornou isso possível. Além disso, o uso de *tags* permite que os aplicativos selecionem apenas as partes de um documento que precisam processar, e isso não será afetado pela adição de informações que são relevantes para outros aplicativos.

Os documentos XML, sendo textuais, podem ser lidos por seres humanos. Na prática, a maioria dos documentos XML é gerada e lida por software de processamento de XML, mas a capacidade de ler código XML pode ser útil quando as coisas dão errado. Além disso, o uso de texto torna a XML independente de qualquer plataforma específica. O uso de uma representação textual, em vez de binária, junto com o uso de *tags*, torna as mensagens muito maiores, o que faz com que elas exijam tempos de processamento e transmissão maiores, assim como mais espaço de armazenamento. Uma comparação da eficiência das mensagens usando o formato XML SOAP e o CDR do CORBA é dada na Seção 19.2.4. Entretanto, os arquivos e as mensagens podem ser compactados – a HTTP versão 1.1 permite que os dados sejam compactados, o que economiza largura de banda durante a transmissão.

Elementos e atributos XML ◊ A Figura 4.10 mostra a definição XML da estrutura *Person* que foi usada para ilustrar o empacotamento no CDR do CORBA e em Java. Ela mostra que a XML consiste em *tags* e dados do tipo caractere. Os dados do tipo caractere, por exemplo, *Smith* ou *1934*, são os dados reais. Assim como na HTML, a estrutura de um documento XML é definida por pares de *tags* incluídas entre sinais de menor e maior. No exemplo anterior, *<name>* e *<place>* são *tags*. Assim como na HTML, o leiaute geralmente pode ser usado para melhorar a legibilidade. Na XML, os comentários são denotados da mesma maneira que na HTML.

Elementos: um elemento na XML consiste em um conjunto de dados do tipo caractere delimitados por *tags* de início e fim correspondentes. Por exemplo, um dos elementos na Figura 4.10 consiste no dado *Smith*, contido dentro do par de *tags <name> ... </name>*. Note que o elemento com a *tag <name>* é incluído no elemento com o par de *tags <person id="123456789"> ... </person >*. A capacidade de um elemento de incluir outro permite a representação de dados hierárquicos – um aspecto muito importante

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- um comentário -->
</person >
```

Figura 4.10 Definição em XML da estrutura *Person*.

da XML. Uma *tag* vazia não tem nenhum conteúdo e é terminada com />, em vez de >. Por exemplo, a tag vazia <*european*/> poderia ser incluída dentro da *tag* <*person*>...</*person*>.

Atributos: opcionalmente, uma *tag* de início pode incluir pares de nomes e valores de atributo associados, como em *id="123456789"*, conforme mostrado anteriormente. A sintaxe é igual à da HTML, onde um nome de atributo é seguido de um sinal de igualdade e um valor de atributo entre aspas. Múltiplos valores de atributo são separados por espaços.

É uma questão de escolha definir quais itens serão representados como elementos e quais serão representados como atributos. Um elemento geralmente é um contêiner para dados, enquanto um atributo é usado para rotular esses dados. Em nosso exemplo, *123456789*, poderia ser um identificador usado pelo aplicativo, enquanto *name*, *place* e *year* poderiam ser exibidos. Além disso, se os dados contêm subestruturas ou várias linhas, eles devem ser definidos como um elemento. Os atributos servem para valores simples.

Nomes: os nomes de *tags* e atributos na XML geralmente começam com uma letra, mas também podem começar com um sublinhado ou com dois-pontos. Os nomes continuam com letras, dígitos, hífens, sublinhados, dois-pontos ou pontos-finais. Letras maiúsculas e minúsculas são levadas em consideração, isto é, os nomes em XML são *case-sensitive*. Os nomes que começam com *xml* são reservados.

Dados binários: todas as informações nos elementos XML devem ser expressas com dados do tipo caractere. Mas a questão é: como representamos elementos criptografados ou *hashing* de códigos de segurança – os quais, como veremos na Seção 19.5, são usados em XML. A resposta é que eles podem ser representados na notação *base64* [Freed e Borenstein 1996], que utiliza apenas os caracteres alfanuméricos, junto com +, / e =, que têm significado especial.

Análise (*parsing*) e documentos bem formados ◊ Um documento XML deve ser bem formado – isto é, ele deve obedecer às regras sobre sua estrutura. Uma regra básica é que toda *tag* de início tem uma *tag* de fim correspondente. Outra regra básica é que todas as *tags* devem ser corretamente aninhadas, por exemplo <*x*>..<*y*>..</*y*>..</*x*> está correto, enquanto <*x*>..<*y*>....</*x*>.. </*y*>, não. Finalmente, todo documento XML deve ter um único elemento-raiz que englobe todos os outros elementos. Essas regras tornam muito simples implementar analisadores gramaticais (*parsers*) de documentos XML. Um analisador, ao ler um documento XML que não está bem formado, relata um erro fatal.

CDATA: normalmente, os analisadores de XML verificam o conteúdo dos elementos, pois ele pode conter mais estruturas aninhadas. Mas se o texto precisa conter um sinal de maior (ou menor) ou aspas, ele deve ser representado de uma maneira especial; por exemplo, *&lt* representa o sinal de menor. Entretanto, se uma seção não deve ser analisada por qualquer motivo, por exemplo, se contiver caracteres especiais, ela pode ser denotada como *CDATA*. Por exemplo, se um nome de lugar precisasse incluir um apóstrofo, ele poderia ser especificado de uma das duas maneiras a seguir:

```
<place> King&apos Cross </place >
<place> <![CDATA [King's Cross]]></place >
```

Prólogo XML: todo documento XML deve ter um prólogo como sua primeira linha. O prólogo deve especificar pelo menos a versão de XML que está sendo usada (que atualmente é a 1.0). Por exemplo:

```
<?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>
```

Um terceiro atributo pode ser usado para informar se o documento é único ou se é dependente de definições externas.

Codificações: o prólogo também pode especificar a codificação (UTF-8 é o padrão e foi explicado na Seção 4.3.2). O termo *codificação* se refere ao conjunto de códigos usados para representar caracteres – sendo o código ASCII o melhor exemplo conhecido. Note que, no prólogo XML, o código ASCII é especificado como *us-ascii*. Outras codificações possíveis incluem ISO-8859-1 (ou Latin-1), uma codificação de oito bits cujos primeiros 128 valores são ASCII, sendo o restante usado para representar os caracteres dos idiomas da Europa Ocidental. Outras codificações de oito bits estão disponíveis para representar outros alfabetos; por exemplo, grego ou cirílico.

Espaços de nomes na XML ◊ Tradicionalmente, os espaços de nomes fornecem uma maneira para dar escopo aos nomes. Um espaço de nomes XML é um conjunto de nomes para uma coleção de tipos e atributos de elemento, que é referenciado por um URL. Um espaço de nomes da XML pode ser usado por qualquer outro documento XML, referindo-se ao seu URL.

Qualquer elemento que utilize um espaço de nomes XML pode especificar esse espaço como um atributo chamado *xmlns*, cujo valor é um URL que faz referência a um arquivo que contém as definições do espaço de nomes. Por exemplo:

xmlns:pers = "http://www.cdk4.net/person"

O nome que aparece após *xmlns*, neste caso, *pers*, pode ser usado como prefixo para se referir aos elementos de um espaço de nomes em particular, como mostrado na Figura 4.11. O prefixo *pers* está vinculado a *http://www.cdk4.net/person* para o elemento *person*. Um espaço de nomes se aplica dentro do contexto do par de *tags* de início e fim que o engloba, a não ser que seja sobrescrita por uma nova declaração de espaço de nomes interna a si. Um documento XML pode ser definido em termos de vários espaços de nomes diferentes, cada um dos quais seria referenciado por um prefixo exclusivo.

A convenção de espaço de nomes permite que um aplicativo utilize vários conjuntos de definições externas em diferentes espaços de nomes, sem o risco de conflito de nomes.

Esquemas XML ◊ Um esquema XML [www.w3.org VIII] define os elementos e atributos que podem aparecer em um documento, o modo como os elementos são aninhados, a ordem, o número de elementos, se um elemento está vazio ou se pode conter texto. Para cada elemento, ele define o tipo e o valor padrão. A Figura 4.12 fornece um exemplo de esquema que define os tipos de dados e a estrutura da definição XML da estrutura *person* da Figura 4.10.

A intenção é que uma definição de esquema possa ser compartilhada por muitos documentos diferentes. Um documento XML, definido de forma a obedecer um esquema em particular, também pode ser validado por meio desse esquema. Por exemplo, o remetente de uma mensagem SOAP pode usar um esquema XML para codificá-la e o destinatário usará o mesmo esquema XML para validá-la e decodificá-la.

Definições de tipo de documento: As definições de tipo de documento (DTDs – *Document Type Definitions*) [www.w3.org VI] foram fornecidas como parte da especificação XML 1.0 para definir a estrutura de documentos XML e ainda são amplamente usadas com esse propósito. A sintaxe das DTDs é diferente do restante da XML e é bastante limitada no sentido do que pode especificar; por exemplo, ela não pode descrever tipos de dados, e suas definições são globais, impedindo que nomes de elemento sejam duplicados. As DTDs não são usadas para definir serviços web, embora possam ser usadas para definir documentos que são transmitidos por esses.

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk4.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1934 </pers:year>
</person>
```

Figura 4.11 Ilustração do uso de um espaço de nomes na estrutura *Person*.

```
<xsd:scheme xmlns:xsd = URL das definições de esquema XML >
    <xsd:element name= "person" type ="personType" />
        <xsd:complexType name="personType">
            <xsd:sequence>
                <xsd:element name = "name" type="xs:string"/>
                <xsd:element name = "place" type="xs:string"/>
                <xsd:element name = "year" type="xs:positiveInteger"/>
            </xsd:sequence>
            <xsd:attribute name= "id" type = "xs:positiveInteger"/>
        </xsd:complexType>
</xsd:scheme>
```

Figura 4.12 Um esquema XML para a estrutura *Person*.

APIs para acessar XML ◊ Analisadores e geradores de XML estão disponíveis para as linguagens de programação mais comumente usadas. Por exemplo, existe software Java para escrever objetos Java como XML (empacotamento) e para criar objetos Java a partir de tais estruturas (desempacotamento). Software semelhante está disponível em Python para tipos de dados e objetos Python.

4.3.4 Referências a objetos remotos

Esta seção se aplica apenas às linguagens que suportam o modelo de objeto distribuído, como Java e CORBA. Ela não é relevante para XML.

Quando um cliente invoca um método em um objeto remoto, uma mensagem de invocação é enviada para o processo servidor que contém o objeto remoto. Essa mensagem precisa especificar qual o objeto em particular deve ter seu método executado. Uma *referência de objeto remoto* é o identificador de um objeto remoto, e é válido em todo um sistema distribuído. A referência de objeto remoto é passada na mensagem de invocação para especificar qual objeto deve ser ativado. O Capítulo 5 explicará que as referências de objeto remoto também são passadas como argumentos e retornadas como resultados de invocações a métodos remotos, que cada objeto remoto tem uma única referência e que essas referências podem ser comparadas para ver se elas dizem respeito ao mesmo objeto remoto. Agora, discutiremos a representação externa das referências de objeto remoto.

As referências de objeto remoto devem ser geradas de uma forma que garanta a sua exclusividade no espaço e no tempo. Em geral, podem existir muitos processos contendo objetos remotos; portanto, as referências de objeto remoto devem ser únicas entre todos os processos, nos vários computadores de um sistema distribuído. Mesmo após um objeto remoto, associado a uma determinada referência, ter sido excluído, é importante que a referência de objeto remoto não seja reutilizada, pois seus ativadores em potencial podem manter referências obsoletas. Qualquer tentativa de ativar um objeto excluído deve produzir um erro, em vez de permitir o acesso a um objeto diferente.

Existem várias maneiras de garantir a exclusividade de uma referência de objeto remoto. Uma delas é construir uma referência concatenando o endereço IP de seu computador e o número de porta do processo que a criou, com a hora de sua criação e um número de objeto local. O número de objeto local é incrementado sempre que um objeto é criado nesse processo.

Juntos, o número de porta e a hora produzem um identificador de processo exclusivo nesse computador. Com essa estratégia, as referências de objeto remoto podem ser representadas com um formato como o que aparece na Figura 4.13. Nas implementações mais simples de RMI, os objetos remotos residem no processo que os criou e existem apenas enquanto esse processo continua a ser executado. Nesses casos, a referência de objeto remoto pode ser usada como endereço para objeto remoto. Em outras palavras, as mensagens de invocação são enviadas para o endereço IP, processo e porta fornecidos pela referência remota.

Para permitir que os objetos remotos sejam migrados para um processo diferente em outro computador, a referência de objeto remoto não deve ser usada como endereço do objeto remoto. A Seção

Figura 4.13 Representação de uma referência de objeto remoto.

20.2.4 discutirá uma forma de referência de objeto remoto que permite aos objetos serem invocados em diferentes servidores enquanto existirem.

Os sistemas *peer-to-peer*, Pastry e Tapestry, descritos no Capítulo 10, usam uma forma de objeto remoto que é completamente independente da localização. As mensagens são direcionadas para recursos por meio de um algoritmo de roteamento distribuído.

O último campo da referência de objeto remoto mostrada na Figura 4.13 contém informações sobre a interface do objeto remoto; por exemplo, o nome da interface. Essas informações são importantes para todo processo que recebe uma referência de objeto remoto como argumento ou resultado de uma invocação remota, pois ele precisa conhecer os métodos oferecidos pelo objeto remoto. Esse ponto será explicado novamente na Seção 5.2.5.

4.4 Comunicação cliente-servidor

Esta forma de comunicação é projetada para suportar as funções e trocas de mensagem em interações cliente-servidor típicas. No caso mais comum, a comunicação requisição e resposta é síncrona, pois o processo cliente bloqueia até que a resposta do servidor chegue. Ela também é confiável, pois a resposta é efetivamente a confirmação de que a requisição chegou ao servidor. A comunicação de requisição e resposta assíncrona é uma alternativa que pode ser útil em situações onde os clientes podem recuperar as respostas posteriormente – veja a Seção 6.5.2.

As interações cliente-servidor serão descritas nos próximos parágrafos em termos das operações *send* e *receive* na API Java, para datagramas UDP, embora muitas implementações atuais utilizem TCP. Um protocolo construído sobre datagramas evita as sobrecargas desnecessárias associadas ao protocolo TCP. Em particular:

- as confirmações são redundantes, pois as requisições são seguidas pelas respostas;
- o estabelecimento de uma conexão envolve mensagens extras, além do par exigido para a requisição e sua resposta;
- o controle de fluxo é redundante para a maioria das invocações, que passam apenas pequenos argumentos e resultados.

O protocolo requisição-resposta ◊ O protocolo a seguir é baseado em três primitivas de comunicação: *doOperation*, *getRequest* e *sendReply*, como mostrado na Figura 4.14. A maioria dos sistemas de RMI e RPC é suportada por um protocolo semelhante. O que descrevemos aqui é personalizado para suportar RMI, pois ele passa, na mensagem de pedido, uma referência de objeto remoto do objeto cujo método deve ser invocado.

O protocolo requisição-resposta é projetado justamente para associar respostas as suas correspondentes requisições, podendo ainda ser projetado de forma a fornecer certas garantias quanto à entrega das mensagens. Se forem usados datagramas UDP, as garantias de entrega deverão ser fornecidas pelo protocolo requisição-resposta, o qual pode usar a mensagem de resposta do servidor como confirmação da mensagem de requisição do cliente. A Figura 4.15 esboça as três primitivas de comunicação.

O método *doOperation* é usado pelos clientes para invocar operações remotas. Seus argumentos especificam o objeto remoto e o método a ser executado, junto com informações adicionais (argumentos) exigidas pelo método. Seu resultado é uma resposta RMI. Pressupõe-se que o cliente que chama *doOperation* empacota os argumentos em um array de bytes e desempacota os resultados do array de bytes recebido como retorno. O primeiro argumento de *doOperation* é uma instância da classe *Remo-*

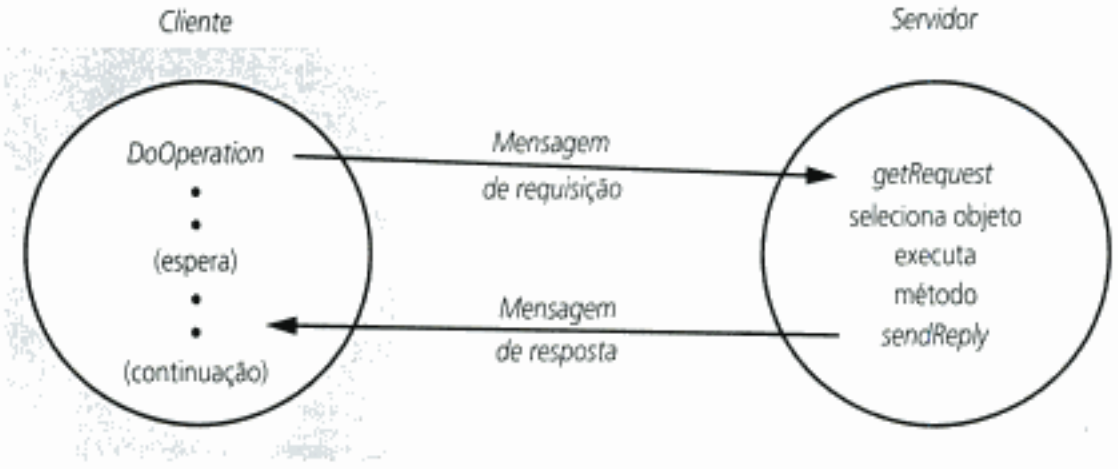


Figura 4.14 Comunicação requisição-resposta.

teObjectRef, que representa referências de objetos remotos, por exemplo, como mostrado na Figura 4.13. Essa classe fornece métodos para obter o endereço IP e a porta do servidor do objeto remoto. O método *doOperation* envia uma mensagem de requisição para o servidor cujo endereço IP e porta são especificados na referência de objeto remoto passada como argumento. Após enviar a mensagem de requisição, *doOperation* faz um *receive* para obter uma mensagem de resposta, a partir da qual extrai o resultado e o retorna para o chamador. O chamador de *doOperation* é bloqueado até que o objeto remoto no servidor execute a operação solicitada e transmita uma mensagem de resposta para o processo cliente.

O *getRequest* é usado por um processo servidor para ler as requisições de serviço, como mostrado na Figura 4.14. Quando o servidor tiver executado o método no objeto especificado, usará *sendReply* para enviar a mensagem de resposta para o cliente. O cliente, ao receber a mensagem de resposta, fará com que o método *doOperation* original seja desbloqueado e a execução do programa cliente continue.

As informações a serem transmitidas em uma mensagem de requisição ou resposta aparecem na Figura 4.16. O primeiro campo indica se a mensagem é *Request* (requisição) ou *Reply* (resposta). O segundo campo *requestId* contém um identificador de mensagem. O método *doOperation* do cliente gera um campo *requestId* para cada mensagem de requisição e o servidor o copia nas mensagens de resposta correspondentes. Isso permite que *doOperation* verifique se a mensagem de resposta é resultado da requisição corrente e não de uma chamada anterior atrasada. O terceiro campo é uma referência de objeto remoto empacotada na forma mostrada na Figura 4.13. O quarto campo é um identificador do método a ser invocado; por exemplo, os métodos de uma interface poderiam ser numerados como 1, 2, 3, ...; caso o cliente e o servidor utilizem uma linguagem comum que suporte reflexão, então uma representação do método em si pode ser colocada nesse campo – em Java, uma instância de *Method* pode ser colocada nesse campo.

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
envia uma mensagem de requisição para um objeto remoto e retorna a resposta.
Os argumentos especificam o objeto remoto, o método a ser invocado e os argumentos desse método.

public byte[] getRequest ();
lê uma requisição de cliente por meio da porta de servidor.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
envia a mensagem de resposta para o cliente em seu endereço IP e porta, respectivos.

Figura 4.15 Operações do protocolo requisição-resposta.

Tipo da mensagem	*int (0=Request, 1= Reply)*
Identificador Requisição (RequestId)	*int*
Referência do Objeto	*RemoteObjectRef*
Identificador de método (methodId)	*int ou Method*
Argumentos	*// array de bytes*

Figura 4.16 **Estrutura da mensagem de requisição e resposta.**

Identificadores de mensagem ◊ Todo esquema que envolve o gerenciamento de mensagens para fornecer propriedades adicionais, como a entrega confiável de mensagens ou a comunicação requisição-resposta, exige que cada mensagem tenha um identificador exclusivo, por meio do qual ela possa ser referenciada. Um identificador de mensagem consiste em duas partes:

1. um *identificador da requisição*, que é criado pelo processo remetente a partir de uma seqüência ascendente de valores inteiros; e
2. um identificador do processo remetente; por exemplo, sua porta e endereço IP.

A primeira parte gera o identificador exclusivo para o remetente e a segunda o torna único no sistema distribuído. (A segunda parte pode ser obtida diretamente da mensagem recebida – por exemplo, se UDP estiver em uso.)

Quando o valor do *identificador da requisição* atingir o máximo para um inteiro sem sinal (por exemplo, $2^{32} - 1$) ele voltará a zero. A única restrição aqui é que o tempo de vida de um identificador de mensagem deve ser muito menor do que o tempo que leva para esgotar os valores da seqüência de inteiros.

Modelo de falhas do protocolo requisição-resposta ◊ Se as três primitivas *doOperation*, *getRequest* e *sendReply* forem implementadas em datagramas UDP, então elas sofrerão das mesmas falhas de comunicação. Isto é:

- elas sofrerão de falhas por omissão;
- não haverá garantia de que as mensagens sejam entregues na ordem da emissão.

Além disso, o protocolo poderá sofrer com as falhas dos processos (veja a Seção 2.3.2). Pressupomos falhas do tipo colapso, isto é, quando eles falham, permanecem parados. Em outros termos, não há um comportamento bizantino.

Para considerar as ocasiões em que um servidor tiver falhado, ou que uma mensagem de requisição ou resposta tenha sido descartada, *doOperation* utiliza um *timeout* para limitar a espera de uma mensagem de resposta do servidor. A ação executada quando o *timeout* ocorre dependerá das garantias de entrega a serem oferecidas.

Timeouts ◊ Existem várias opções para o que *doOperation* deva fazer após o *timeout*. A opção mais simples é retornar imediatamente de *doOperation*, com uma indicação para o cliente de que o método *doOperation* falhou. Essa não é a estratégia usual – o *timeout* pode ter sido atingido devido à perda da mensagem de requisição ou da resposta – e, neste último caso, a operação terá sido executada. Para compensar a possibilidade de perda de mensagens, *doOperation* envia a mensagem de requisição repetidamente até que receba uma resposta, ou esteja razoavelmente seguro de que o atraso é devido à falta de resposta do servidor e não por causa de mensagens perdidas. Finalmente, quando *doOperation* retornar, indicará isso para o cliente por meio de uma exceção, dizendo que nenhum resultado foi recebido.

Descartando mensagens de requisição duplicadas ◊ Nos casos em que a mensagem de requisição é retransmitida, o servidor pode recebê-la mais de uma vez. Por exemplo, o servidor pode receber

a primeira mensagem de requisição, mas demorar mais do que o *timeout* do cliente para executar o comando e retornar a resposta. Isso pode levar à execução de uma operação mais de uma vez para uma mesma requisição por parte do servidor. Para evitar isso, o protocolo é projetado de modo a reconhecer mensagens sucessivas (do mesmo cliente) com o mesmo identificador de requisição e filtrar as duplicatas. Se o servidor ainda não tiver enviado a resposta, ele não precisará executar nenhuma ação especial – ele transmitirá a resposta quando tiver terminado de executar a operação.

Perda de mensagens de resposta ◊ Se o servidor já tiver enviado a resposta quando receber uma requisição duplicada, ele precisará executar a operação novamente para obter o resultado, a não ser que tenha armazenado o resultado da execução original. Alguns servidores podem executar suas operações mais de uma vez e obter o mesmo resultado a cada vez. Uma *operação idempotente* é aquela que pode ser executada repetidamente, com o mesmo efeito de que se tivesse sido executada apenas uma vez. Por exemplo, uma operação para pôr um elemento em um conjunto é idempotente, pois ela sempre terá o mesmo efeito no conjunto, toda vez que for executada, enquanto uma operação para incluir um elemento em uma seqüência não é idempotente, pois ela amplia a seqüência sempre que é executada. Um servidor cujas operações são todas idempotentes não precisa adotar medidas especiais para evitar as suas execuções mais de uma vez.

Histórico ◊ Para os servidores que exigem retransmissão das respostas sem executar novamente as operações, pode ser usado um histórico. O termo "histórico" é usado para se referir a uma estrutura que contém um registro das mensagens que foram transmitidas. Uma entrada em um histórico contém um identificador de requisição, uma mensagem e um identificador do cliente para o qual ela foi enviada. Seu objetivo é permitir que o servidor retransmita as mensagens de resposta quando os processos clientes as solicitarem. Um problema associado ao uso de um histórico é seu consumo de memória. Um histórico pode se tornar muito grande, a menos que o servidor possa identificar quando não há mais necessidade de retransmissão das mensagens.

Como, normalmente, os clientes só podem fazer uma requisição por vez, o servidor pode interpretar cada nova requisição como uma confirmação de sua resposta anterior. Portanto, o histórico precisa conter apenas a última mensagem de resposta enviada a cada cliente. Entretanto, o volume de mensagens de resposta no histórico de um servidor pode ser um problema quando ele tiver um grande número de clientes. Em particular, quando um processo cliente termina, ele não confirma a última resposta recebida – portanto, as mensagens no histórico normalmente são descartadas após um determinado período de tempo.

Protocolos RPC ◊ Três protocolos, que produzem diferentes comportamentos na presença de falhas de comunicação, são usados para implementar vários tipos de RPC. Eles foram identificados originalmente por Spector [1982]:

- o protocolo *request (R)*;
- o protocolo *request-reply (RR)*;
- o protocolo *request-reply-acknowledge reply (RRA)*.

As mensagens passadas nesses protocolos estão resumidas na Figura 4.17. No protocolo R, uma única mensagem *Request* é enviada pelo cliente para o servidor. O protocolo R pode ser usado quando não existe nenhum valor a ser retornado do método remoto e o cliente não exige confirmação de que

Nome	*Mensagens enviadas pelo*		
	Cliente	*Servidor*	*Cliente*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Figura 4.17 Protocolos RPC.

o método foi executado. O cliente pode prosseguir imediatamente após a mensagem de requisição ter sido enviada, pois não há necessidade de esperar por uma mensagem de resposta. Esse protocolo é implementado sobre datagramas UDP e, portanto, sofre das mesmas falhas de comunicação.

O protocolo RR é útil para a maioria das trocas cliente-servidor, pois é baseado no protocolo requisição-resposta. Não são exigidas mensagens de confirmação especiais, pois uma mensagem de resposta do servidor é considerada como confirmação do recebimento da mensagem de requisição do cliente. Analogamente, uma requisição subseqüente de um cliente pode ser considerada como uma confirmação da mensagem de resposta de um servidor à requisição anterior. Conforme vimos anteriormente, as falhas de comunicação ocasionadas pelos datagramas UDP podem ser mascaradas pela retransmissão das requisições com filtragem duplicada e o registro das respostas em um histórico.

O protocolo RRA é baseado na troca de três mensagens: requisição, resposta e confirmação. A mensagem de *confirmação* contém a *requestId* da mensagem de resposta que está sendo confirmada. Isso permitirá que o servidor descarte entradas de seu histórico. A chegada de uma *requestId* em uma mensagem de confirmação será interpretada como a acusação do recebimento de todas as mensagens de resposta com valores de *requestId* menores; portanto, a perda de uma mensagem de confirmação não é muito prejudicial ao sistema. Embora o protocolo RRA envolva uma mensagem adicional, ela não precisa bloquear o cliente, pois a confirmação pode ser transmitida após a resposta ter sido entregue ao cliente, mas ela utiliza recursos de processamento e rede. O Exercício 4.23 sugere uma otimização para o protocolo RRA.

Uso de TCP para implementar o protocolo requisição-resposta ◊ A seção sobre datagramas mencionou que freqüentemente é difícil decidir-se sobre um tamanho apropriado para o buffer de recebimento de datagramas. No protocolo requisição-resposta, isso se aplica aos buffers usados pelo servidor para receber mensagens de requisição e pelo cliente para receber respostas. O comprimento limitado dos datagramas (normalmente, 8 quilobytes) pode não ser considerado adequado para uso em sistemas transparentes de RMI, pois os argumentos ou resultados dos procedimentos podem ser de qualquer tamanho.

O desejo de evitar a implementação de protocolos que tratem de "quebrar" requisições e respostas em múltiplos pacotes é um dos motivos para escolher a implementação de protocolos requisição-resposta em TCP. Isso permite que tanto argumentos como resultados de qualquer tamanho sejam transmitidos. Em particular, a serialização de objeto Java é um protocolo que permite que argumentos e resultados sejam enviados por meio de fluxos entre cliente e servidor, tornando possível que conjuntos de objetos de qualquer tamanho sejam transmitidos de maneira confiável. Se o protocolo TCP for usado, isso garantirá que as mensagens de requisição e de resposta sejam entregues de modo confiável; portanto, não há necessidade de o protocolo requisição-resposta tratar da retransmissão de mensagens, filtrar duplicatas ou de tratar com históricos. Além disso, o mecanismo de controle de fluxo permite que argumentos e resultados de grandes tamanhos sejam passados, sem se tomar medidas especiais no destinatário. Assim, a escolha do protocolo TCP simplifica a implementação de protocolos requisição-resposta. Se sucessivas requisições e respostas entre o mesmo par cliente-servidor são enviadas por um mesmo fluxo TCP, a sobrecarga de conexão necessária não se aplica a toda invocação remota. Além disso, a sobrecarga devido às mensagens de confirmação TCP é reduzida quando uma mensagem de resposta é gerada logo após a mensagem de requisição.

Às vezes, o aplicativo não exige todos os recursos oferecidos pelo TCP e um protocolo mais eficiente, especialmente customizado, pode ser implementado sobre UDP. Por exemplo, conforme mencionamos anteriormente, o NFS da Sun não exige mensagens de tamanho ilimitado, pois transmite blocos de arquivo de tamanho fixo entre o cliente e o servidor. Além disso, suas operações são projetadas para serem idempotentes, de modo que não importa se as operações são executadas mais de uma vez para retransmitir mensagens de resposta perdidas, tornando desnecessário manter um histórico.

HTTP: um exemplo de protocolo requisição-resposta ◊ O Capítulo 1 apresentou o protocolo HTTP (*HyperText Transfer Protocol*), usado pelos navegadores web para fazer pedidos para servidores web e receber suas respostas. Para recapitular, os servidores web gerenciam recursos implementados de diferentes maneiras:

- dados; por exemplo, o texto de uma página em HTML, uma imagem ou a classe de um *applet*;

- programa; por exemplo, programas *cgi* e *servlets* (veja [java.sun.com III]), que podem ser executados no servidor web.

As requisições do cliente especificam um URL que inclui o nome de *host* DNS de um servidor web e um número de porta opcional no servidor web, assim como o identificador de um recurso nesse servidor.

O protocolo HTTP especifica as mensagens envolvidas em uma troca requisição e resposta, os métodos, os argumentos, os resultados e as regras para representá-los (empacotamento) nas mensagens. Ele suporta um conjunto fixo de métodos (*GET, PUT, POST, etc.*) que são aplicáveis a todos os seus recursos. Ele é diferente dos protocolos acima, onde cada objeto tem seus próprios métodos. Além de invocar métodos sobre recursos web, o protocolo permite negociação de conteúdo e autenticação com senha.

Negociação de conteúdo: as requisições dos clientes podem incluir informações sobre qual representação de dados elas podem aceitar (por exemplo, linguagem ou tipo de mídia), permitindo que o servidor escolha a representação mais apropriada para o usuário.

Autenticação: credenciais e desafios (*challenges*) são usados para suportar autenticação com senha. Na primeira tentativa de acessar uma área protegida com senha, a resposta do servidor contém um desafio aplicável ao recurso. O Capítulo 7 explicará os desafios. Quando um cliente recebe um desafio, faz o usuário digitar um nome e uma senha, e envia as credenciais associadas às requisições subseqüentes. O protocolo HTTP é implementado sobre TCP. Na versão original do protocolo, cada interação cliente-servidor consiste nas seguintes etapas:

- o cliente solicita uma conexão com o servidor na porta HTTP padrão (80) ou em uma porta especificada no URL;
- o cliente envia uma mensagem de requisição para o servidor;
- o servidor envia uma mensagem de resposta para o cliente;
- a conexão é fechada.

Entretanto, a necessidade de estabelecer e fechar uma conexão para cada troca requisição-resposta é dispendiosa, tanto pela sobrecarga no servidor como no envio de mensagens pela rede. Lembrando que os navegadores geralmente fazem vários pedidos para o mesmo servidor, uma versão posterior do protocolo (HTTP 1.1, veja RFC 2616 [Fielding *et al.* 1999]) usa *conexões persistentes* – conexões que permanecem abertas durante uma seqüência de trocas requisição-resposta entre cliente e servidor. Uma conexão persistente pode ser encerrada a qualquer momento, tanto pelo cliente como pelo servidor, através do envio de uma indicação especial para o outro participante. Os servidores encerrarão uma conexão persistente quando ela estiver ociosa por um determinado período de tempo. É possível que um cliente receba uma mensagem do servidor dizendo que a conexão está encerrada, no meio do envio de outra requisição (ou requisições). Nesses casos, o navegador enviará novamente as requisições, sem envolvimento do usuário, desde que as operações sejam idempotentes. Por exemplo, o método GET descrito a seguir é idempotente. Onde estão envolvidas operações não idempotentes, o navegador deve consultar o usuário para saber o que fazer em seguida.

As requisições e respostas são empacotadas nas mensagens como strings de texto ASCII, mas os recursos podem ser representados como seqüências de bytes e podem ser compactados. A utilização de texto na representação externa de dados simplificou o uso de HTTP pelo programador de aplicativos que trabalha diretamente com o protocolo. Nesse contexto, uma representação textual não aumenta muito o comprimento das mensagens.

Os recursos implementados como dados são fornecidos como estruturas do tipo MIME em argumentos e resultados. O MIME (*Multipurpose Internet Mail Extensions*) é um padrão para envio de dados de múltiplas partes, especificado no RFC 2045 [Freed e Borenstein 1996], contendo, por exemplo, texto, imagens e som em mensagens de e-mail. Os dados são prefixados com seu *tipo Mime* para que o destinatário saiba como manipulá-los. Um *tipo Mime* especifica um tipo e um subtipo, por exemplo, *text/plain*, *text/html*, *image/gif*, *image/jpeg*. Os clientes também podem especificar os tipos Mime que desejam aceitar.

Métodos HTTP ◊ Cada requisição especifica o nome de um método a ser aplicado em um recurso no servidor e o URL desse recurso. A resposta fornece o status da requisição. As mensagens de requisição e resposta também podem conter dados de recurso, como o conteúdo de um formulário ou a saída de programa executado no servidor web. Os métodos incluem os seguintes:

GET: solicita o recurso cujo URL é dado como argumento. Se o URL se referir a dados, então o servidor web responderá retornando os dados identificados por esse URL. Se o URL se referir a um programa, então o servidor web executará o programa e retornará sua saída para o cliente. Argumentos podem ser adicionados no URL; por exemplo, o método GET pode ser usado para enviar o conteúdo de um formulário como argumento para um programa *cgi*. A operação *GET* pode ser condicionada à data em que um recurso foi modificado pela última vez. *GET* também pode ser configurado para obter partes dos dados.

HEAD: esta requisição é idêntica a GET, mas não retorna nenhum dado. Entretanto, retorna todas as informações sobre os dados, como a hora da última modificação, seu tipo ou seu tamanho.

POST: especifica o URL de um recurso (por exemplo, um programa) que pode tratar dos dados fornecidos com o pedido. O processamento executado nos dados depende da função do programa especificado no URL. Esse método é feito para tratar:

- do fornecimento de um bloco de dados (por exemplo, um formulário) para um processo de manipulação de dados, como um *servlet* ou um programa *cgi*;
- do envio de uma mensagem para um *bulletin board*, lista de distribuição ou *newsgroup*;
- da ampliação de um banco de dados com uma operação *append*.

PUT: armazena os dados fornecidos na requisição no URL, como uma modificação em um recurso já existente ou como um novo recurso.

DELETE: o servidor exclui o recurso identificado pelo URL dado. Nem sempre os servidores permitem essa operação, nesse caso, a resposta indicará a falha.

OPTIONS: o servidor fornece ao cliente uma lista de métodos que podem ser aplicados no URL dado (por exemplo, *GET*, *HEAD*, *PUT*) e seus requisitos especiais.

TRACE: o servidor envia de volta a mensagem de requisição. Usado para propósitos de diagnóstico.

As requisições descritas anteriormente podem ser interceptadas por um servidor *proxy* (veja a Seção 2.2.3) e as respostas de *GET* e *HEAD* podem ser armazenadas em sua cache.

Conteúdo da mensagem ◊ A mensagem *Request* especifica o nome de um método, o URL de um recurso, a versão do protocolo, uma área de cabeçalhos e um corpo de mensagem opcional. A Figura 4.18 mostra o conteúdo de uma mensagem *Request* HTTP cujo método é GET. Quando o URL especifica um recurso de dados, o método GET não tem corpo de mensagem.

As requisições para servidores *proxies* precisam do URL absoluto, como mostrado na Figura 4.18. As requisições para os servidores de origem (aquele onde reside o recurso) especificam um nome de caminho e fornecem o nome DNS do servidor no campo de cabeçalho *Host*. Por exemplo,

GET /index.html HTTP/1.1
Host: www.dcs.qmul.ac.uk

Em geral, os campos de cabeçalho contêm modificadores de requisição e informações do cliente, como as condições do recurso na data de modificação mais recente ou o tipo de conteúdo aceitável (por exemplo, texto HTML, áudio ou imagens JPEG). Um campo de autorização pode ser usado para fornecer as credenciais do cliente, na forma de um certificado, definindo seus direitos de acesso a um recurso.

método	*URL ou nome de caminho*	*versão de HTTP*	*cabeçalhos*	*corpo da mensagem*
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/ 1.1		

Figura 4.18 Mensagem *request* HTTP.

Uma mensagem *Reply* possui a versão do protocolo, um código de status e um "motivo", uma área de cabeçalhos e um corpo de mensagem opcional, como mostrado na Figura 4.19. O *código de status* e o *motivo* fornecem um relato sobre o sucesso ou não na execução da requisição: o primeiro é um valor inteiro de três dígitos para interpretação por um programa e o último é uma frase textual que pode ser entendida por uma pessoa. Os campos de cabeçalho são usados para passar informações adicionais sobre o servidor ou dados relativos ao acesso ao recurso. Por exemplo, se o pedido exige autenticação, o status da resposta indica isso e um campo de cabeçalho contém uma interpelação. Alguns *status* de retorno têm efeitos bastante complexos. Em particular, a resposta de código de status 303 diz ao navegador para que examine um URL diferente, o qual é fornecido em um campo de cabeçalho na resposta. Ela é usada em uma resposta de um programa executado por meio de uma requisição POST, quando esse programa precisar redirecionar o navegador para um recurso selecionado.

O corpo das mensagens de requisição ou de resposta contém os dados associados ao URL especificado na requisição. O corpo da mensagem possui seus próprios cabeçalhos especificando informações sobre os dados, seu comprimento, tipo Mime, conjunto de caracteres, codificação do conteúdo e a data da última modificação. O campo de tipo Mime determina o tipo dos dados, por exemplo *image/jpeg* ou *text/plain*. O campo de codificação do conteúdo especifica o algoritmo de compactação a ser usado.

versão de HTTP	*código de status*	*motivo*	*cabeçalhos*	*corpo da mensagem*
HTTP/ 1.1	200	OK		dados de recurso

Figura 4.19 Mensagem *reply* HTTP.

4.5 Comunicação em grupo

A troca de mensagens aos pares não é o melhor modelo para a comunicação de um processo com um grupo de outros processos, como o que ocorre, por exemplo, quando um serviço é implementado através de diversos processos em computadores diferentes para fornecer tolerância a falhas ou melhorar a disponibilidade. Nesses casos, o emprego de *difusão seletiva* (*multicast*) é mais apropriado – trata-se de uma operação que permite o envio de uma única mensagem para cada um dos membros de um grupo de processos de tal forma que membros participantes do grupo ficam totalmente transparentes para o remetente. Existem diversas possibilidades para o comportamento desejado de *multicast*. A mais simples não fornece garantias a respeito da entrega ou do ordenamento das mensagens.

As mensagens *multicast* fornecem uma infra-estrutura útil para a construção de sistemas distribuídos com as seguintes características:

1. *Tolerância à falha baseada em serviços replicados*: um serviço replicado consiste em um grupo de servidores. As requisições do cliente são difundidas para todos os membros do grupo, cada um dos quais executando uma operação idêntica. Mesmo quando alguns dos membros falham, os clientes ainda podem ser atendidos.
2. *Localização dos servidores de descoberta na interligação em rede espontânea*: a Seção 16.2.1 discutirá os serviços de descoberta para interligação em rede espontânea. Mensagens *multicast* podem ser usadas por servidores e clientes para localizar os serviços de descoberta disponíveis, para registrar suas interfaces ou para pesquisar as interfaces de outros serviços no sistema distribuído.
3. *Melhor desempenho através da replicação de dados*: os dados são replicados para aumentar o desempenho de um serviço – em alguns casos, as réplicas são postas nos computadores dos usuários. Sempre que os dados mudam, o novo valor é enviado por *multicast* para os processos que gerenciam as réplicas.

4. *Propagação de notificações de evento*: o *multicast* para um grupo pode ser usado para notificar os processos de quando algo acontece. Por exemplo, um sistema de notícias poderia avisar os usuários interessados de quando uma nova mensagem tivesse sido divulgada em um *newsgroup* específico.

Apresentamos o *multicast* IP para depois examinarmos as necessidades do uso de comunicação em grupo para ver quais delas são atendidas pelo *multicast* IP. Para as que não são, propomos mais algumas propriedades dos protocolos de comunicação em grupo, além daquelas fornecidas pela difusão seletiva de IP.

4.5.1 Multicast IP – uma implementação de comunicação em grupo

Esta seção discute o *multicast* IP e apresenta a API Java que suporta essa funcionalidade por meio da classe *MulticastSocket*.

***Multicast* IP** ◊ O *multicast* IP permite que o remetente transmita um único datagrama IP para um conjunto de computadores que formam um grupo de *multicast*. O remetente não conhece as identidades dos destinatários individuais nem do tamanho do grupo. Um *grupo multicast* é especificado por um endereço IP classe D (veja a Figura 3.15) – isto é, um endereço cujos primeiros 4 bits são 1110 no protocolo IPv4. Note que os datagramas IP são endereçados para computadores – as portas pertencem aos níveis TCP e UDP.

O fato de ser membro de um grupo *multicast* permite a um computador receber datagramas IP enviados para o grupo. A participação como membro de grupos *multicast* é dinâmica, permitindo aos computadores entrarem ou saírem a qualquer momento e participarem de um número arbitrário de grupos. É possível enviar datagramas para um grupo *multicast* sem ser membro.

Para a programação de aplicativos, o *multicast* IP está disponível apenas por meio de UDP. Um programa aplicativo faz uso de *multicast* enviando datagramas UDP com endereços *multicast* e números de porta normais. Um aplicativo se junta a um grupo *multicast* fazendo seu soquete se unir ao grupo. Em nível IP, um computador pertence a um grupo *multicast* quando um ou mais de seus processos tem soquetes pertencentes a esse grupo. Quando uma mensagem *multicast* chega em um computador, cópias são encaminhadas para todos os soquetes locais que tiverem se juntado ao endereço de *multicast* especificado e estejam vinculados ao número de porta especificado. Os detalhes a seguir são específicos do protocolo IPv4:

Roteadores multicast: os datagramas IP podem ser enviados em *multicast* em uma rede local e na Internet. As transmissões locais exploram a capacidade de *multicast* da rede local, por exemplo, de uma Ethernet. Na Internet fazem uso de roteadores com suporte a *multicast*, os quais encaminham um único datagrama para roteadores membros em outras redes, onde são novamente transmitidos para os membros locais. Para limitar a distância da propagação de um datagrama *multicast*, o remetente pode especificar o número de roteadores pelos quais pode passar – o que é chamado tempo de vida (*time to live*) ou, abreviadamente, TTL. Para entender como os roteadores sabem quais outros roteadores possuem membros de um grupo *multicast*, veja Comer [2000b].

Alocação de endereços multicast: os endereços *multicast* podem ser permanentes ou temporários. Existem grupos permanentes mesmo quando não existem membros – seus endereços são atribuídos pela autoridade da Internet, no intervalo de 224.0.0.1 a 224.0.0.255. Por exemplo, o primeiro endereço se refere ao grupo formado por todos os *hosts* com suporte *multicast*.

O restante dos endereços *multicast* está disponível para os grupos temporários, os quais devem ser criados antes de serem usados e que deixam de existir quando todos os membros tiverem saído. Quando um grupo temporário é criado, ele exige um endereço *multicast* livre para evitar participação acidental em um grupo já existente. O *multicast* provido pelo protocolo IP não trata desse problema. Mas quando seus usuários só precisam se comunicar de forma local, eles configuram o TTL com um valor pequeno, tornando improvável a escolha do mesmo endereço de outro grupo. Entretanto, os programas que usam *multicast* IP em toda a Internet exigem uma solução para o problema. O programa *session directory* (*sd*) pode ser usado para iniciar ou se unir a uma sessão *mul-*

ticast [Handley 1998, session directory]. Ele fornece uma ferramenta com uma interface interativa que permite às pessoas navegarem em sessões *multicast* existentes e anunciar suas próprias sessões, especificando o tempo e a duração – ele escolhe um endereço *multicast* para novas sessões.

Modelo de falhas para datagramas *multicast* ◊ No IP, o envio de datagramas *multicast* tem as mesmas características de falhas dos datagramas UDP – isto é, sofre de falhas por omissão. O efeito sobre um *multicast* é que não há garantia de que as mensagens sejam entregues para um membro do grupo em particular, mesmo em face de uma única falha por omissão; ou seja, alguns dos membros do grupo, mas não todos, podem recebê-las. Como não há garantias que uma mensagem será entregue para um membro de um grupo, esse tipo de comunicação é denominado de *multicast não confiável*. O *multicast confiável* será discutido no Capítulo 12.

API Java para *multicast* IP ◊ A API Java fornece uma interface de datagrama para *multicast* IP por meio da classe *MulticastSocket*, que é uma subclasse de *DatagramSocket* com a capacidade adicional de se unir a grupos *multicast*. A classe *MulticastSocket* fornece dois construtores alternativos, permitindo que soquetes sejam criados de forma a usar uma porta local especificada (como, por exemplo, a 6789, ilustrada na Figura 4.20) ou qualquer porta local livre. Um processo pode se unir a um grupo *multicast*, invocando o método *joinGroup* em seu soquete. Efetivamente, o soquete se junta a um grupo *multicast* em determinada porta e receberá nessa porta os datagramas enviados para esse grupo por processos existentes em outros computadores. Um processo pode sair de um grupo especificado invocando o método *leaveGroup* em seu soquete *multicast*.

No exemplo da Figura 4.20, os argumentos do método *main* especificam uma mensagem e o endereço *multicast* de um grupo (por exemplo, *"228.5.6.7"*). Após se unir a esse grupo *multicast*, o

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
        // args fornece o conteúdo da mensagem e o grupo multicast de destino
          (por exemplo, "228.5.6.7")
        MulticastSocket s =null;
        try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut =
                    new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);
            byte[] buffer = new byte[1000];
            for(int i=0; i< 3; i++) { // obtém mensagens de outros participantes do grupo
                    DatagramPacket messageIn =
                        new DatagramPacket(buffer, buffer.length);
                s.receive(messageIn);
                System.out.println("Received:" + new String(messageIn.getData()));
            }
            s.leaveGroup(group);
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        } catch (IOException e){System.out.println("IO: " + e.getMessage());
        } finally { if(s != null) s.close();}
    }
}
```

Figura 4.20 Um processo se une a um grupo *multicast* para enviar e receber datagramas.

processo cria uma instância de *DatagramPacket* contendo a mensagem e a envia por intermédio de seu soquete *multicast* para o grupo *multicast* na porta 6789. Depois disso, por meio desse mesmo soquete, o processo recebe três mensagens *multicast* destinadas a esse grupo e a essa porta. Quando várias instâncias desse programa são executadas simultaneamente em diferentes computadores, todas elas se unem ao mesmo grupo e cada uma deve receber sua própria mensagem e as mensagens daqueles que se uniram depois dela.

A API Java permite que o TTL seja configurado para um soquete *multicast* por meio do método *setTimeToLive*. O padrão é 1, permitindo que o *multicast* se propague apenas na rede local.

Um aplicativo implementado sobre *multicast* IP pode usar mais de uma porta. Por exemplo, o aplicativo MultiTalk [mbone], que permite a grupos de usuários manterem conversas baseadas em texto, possui uma porta para enviar e receber dados e outra para trocar dados de controle.

4.5.2 Confiabilidade e ordenamento

A seção anterior expôs o modelo de falhas do *multicast* IP, isto é, ele sofre de falhas por omissão. Para os envios *multicast* feitos em uma rede local que possui suporte nativo para que um único datagrama chegue a vários destinos, qualquer um deles pode, individualmente, descartar a mensagem porque seu buffer está cheio. Além disso, um datagrama enviado de um roteador *multicast* para outro pode ser perdido, impedindo assim que todos os destinos que estejam além desse roteador recebam a mensagem.

Outro fator é que qualquer processo pode falhar. Se um roteador *multicast* falhar, os membros do grupo que estiverem além desse roteador não receberão a mensagem, embora os membros locais possam receber.

O ordenamento é outro problema. Os datagramas IP enviados por várias redes interligadas não chegam necessariamente na ordem em que foram emitidos, com o possível efeito de que alguns membros do grupo recebam os datagramas de um único remetente em uma ordem diferente dos outros membros. Além disso, as mensagens enviadas por dois processos diferentes não chegarão necessariamente na mesma ordem em todos os membros do grupo.

Alguns exemplos dos efeitos da confiabilidade e do ordenamento ◊ Agora, vamos considerar o efeito da semântica da falha no *multicast* IP nos quatro exemplos de uso de replicação da introdução da Seção 4.5.

1. *Tolerância a falhas baseada em serviços replicados*: considere um serviço replicado que consiste nos membros de um grupo de servidores que começam no mesmo estado inicial e sempre executam as mesmas operações, na mesma ordem, de modo a permanecerem consistentes uns com os outros. Essa aplicação *multicast* impõe que todas as réplicas, ou nenhuma delas, devam receber cada pedido para executar uma operação – se uma perder um pedido, se tornará inconsistente com relação às outras. Na maioria dos casos, esse serviço exige que todos os membros recebam as mensagens de requisição na mesma ordem dos outros.
2. *Localização dos servidores de descoberta na interligação em rede espontânea*: desde que qualquer processo que queira localizar os servidores de descoberta faça, após sua inicialização, o envio periódico de requisições em *multicast*, uma requisição ocasionalmente perdida não será um problema na localização de um servidor de descoberta. Na verdade, o Jini usa *multicast* IP em seu protocolo de localização dos servidores de descoberta. Isso será descrito na Seção 16.2.1.
3. *Melhor desempenho através de dados replicados*: considere o caso onde os próprios dados replicados, em vez das operações sobre eles, são distribuídos por meio de mensagens de *multicast*. O efeito de mensagens perdidas e ordem inconsistente dependeria do método de replicação e da importância de todas as réplicas estarem totalmente atualizadas. Por exemplo, as réplicas de *newsgroups* não são necessariamente consistentes umas com as outras em dado momento – as mensagens aparecem até em ordens diferentes, mas os usuários podem tolerar isso.
4. *Propagação de notificações de evento*: o aplicativo em particular determina a qualidade exigida do *multicast*. Por exemplo, os serviços de pesquisa Jini utilizam *multicast* IP para anunciar sua existência (veja a Seção 16.2.1).

Esses exemplos sugerem que algumas aplicações exigem um protocolo *multicast* que seja mais confiável do que o oferecido pelo IP. Portanto, há necessidade de um *multicast confiável* – no qual qualquer mensagem transmitida ou é recebida por todos os membros de um grupo ou não é recebida por nenhum deles. Os exemplos também sugerem que algumas aplicações têm forte necessidade de ordem, cujo máximo rigor é chamado de *multicast totalmente ordenado*, na qual todas as mensagens transmitidas para um grupo chegam a todos os membros na mesma ordem.

O Capítulo 12 definirá e mostrará como implementar *multicast confiável* e várias garantias de ordenamento, incluindo o totalmente ordenado.

4.6 Estudo de caso: comunicação entre processos no UNIX

As primitivas IPC nas versões BSD 4.x do UNIX são fornecidas como chamadas de sistema implementadas como uma camada sobre os protocolos TCP e UDP. Os destinos das mensagens são especificados como *endereços de soquete* – um endereço de soquete consiste em um endereço IP e em um número de porta local.

As operações de comunicação entre processos são baseadas na abstração de soquete, descrita na Seção 4.2.2: as mensagens são enfileiradas no soquete remetente até serem transmitidas pelo protocolo de rede e, se o protocolo exigir, uma confirmação de entrega seja recebida. Quando as mensagens chegam no destino, elas são enfileiradas no soquete receptor até que o processo destino faça uma chamada de sistema apropriada para recebê-las.

Qualquer processo pode criar um soquete para usar na comunicação com outro processo. Isso é feito com a chamada de sistema *socket*, cujos argumentos especificam o domínio da comunicação (normalmente, a Internet), o tipo (datagrama ou fluxo) e, às vezes, um protocolo específico. Normalmente, o protocolo (por exemplo, UDP ou TCP) é selecionado pelo sistema, dependendo da comunicação ser por datagrama ou fluxo.

A chamada do soquete retorna um descritor, por meio do qual o soquete pode ser referenciado nas chamadas de sistema subseqüentes. O soquete dura até que seja *fechado*, ou até que todos os processos que tenham o descritor terminem. Dois soquetes podem ser usados para comunicação bidirecional (ou em uma das duas) entre os processos situados no mesmo computador ou em diferentes computadores.

Antes que dois processos possam se comunicar, o destinatário deve *associar* seu descritor de soquete a um endereço de soquete. O remetente também deve associar seu descritor a um endereço de soquete, caso exija uma resposta. A chamada de sistema *bind* é usada para isso; seus argumentos são um descritor de soquete e uma referência para uma estrutura contendo o endereço de soquete com o qual o soquete deve ser vinculado. Uma vez que um soquete tenha sido associado, seu endereço não poderá ser alterado.

Poderia parecer mais razoável ter uma única chamada de sistema para a criação do soquete e para associar um nome a um soquete, como, por exemplo, o que API Java faz. A suposta vantagem de ter duas chamadas separadas é que os soquetes podem ser úteis sem endereços.

Os endereços de soquete são públicos, no sentido de que podem ser usados como destinos por qualquer processo. Depois que um processo tiver associado seu soquete a um endereço, ele poderá ser contatado por outro processo através desse soquete. Todo processo que pretenda receber mensagens por meio de um soquete, deve primeiro vinculá-lo a um endereço e posteriormente torná-lo conhecido para os demais processos.

4.6.1 Comunicação por datagrama

Para enviar datagramas é necessário o envolvimento de um par de soquetes a cada comunicação feita. O processo remetente faz isso usando seu descritor local de soquete e o endereço do soquete destino, sempre que ele enviar uma mensagem.

Esse cenário está ilustrado na Figura 4.21, na qual os detalhes dos argumentos estão simplificados.

- Os dois processos usam a chamada *socket* para criar um soquete e obter um descritor para ele. O primeiro argumento de *socket* especifica o domínio da comunicação como sendo a Internet e o segundo argumento indica que é exigida comunicação por datagrama. O último argumento da

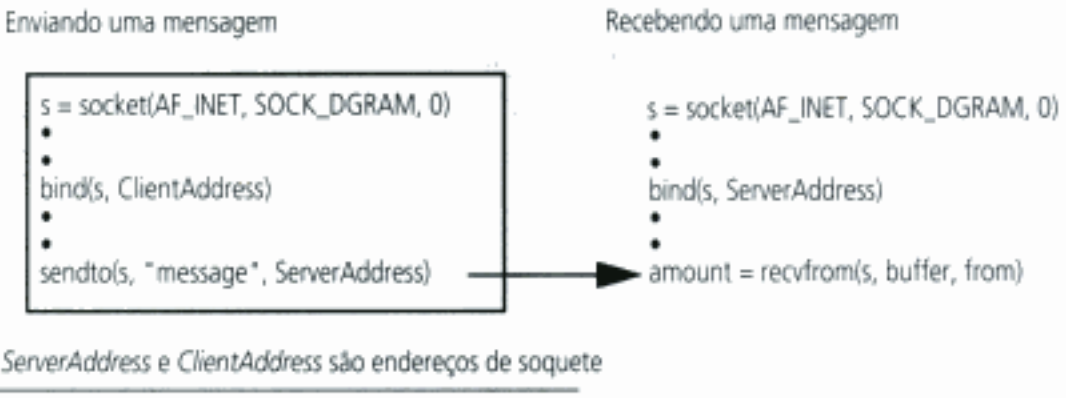


Figura 4.21 **Usando soquetes para comunicação por datagramas.**

chamada de soquete pode ser usado para especificar um protocolo em particular, mas configurá-lo como zero faz com que o sistema selecione um protocolo conveniente – neste caso, UDP.

- Os dois processos usam então a chamada *bind* para vincular seus soquetes a endereços de soquete. O processo remetente vincula seu soquete a um endereço que se refere a qualquer número de porta local disponível. O processo destino vincula seu soquete a um endereço que contém a porta do serviço e deve torná-lo conhecido do remetente.
- O processo remetente usa a chamada *sendto* com argumentos especificando o soquete por meio do qual a mensagem deve ser enviada, a mensagem em si e o endereço de soquete do destino (na realidade, uma referência para uma estrutura que o contém). A chamada *sendto* envia a mensagem para os protocolos UDP e IP e retorna o número de bytes enviados. Como solicitamos o serviço de datagrama, a mensagem é transmitida para seu destino sem recebermos uma confirmação. Se a mensagem for longa demais para ser enviada, haverá o retorno de um erro (e a mensagem não será transmitida).
- O processo destino usa a chamada *recvfrom* com argumentos especificando o soquete local no qual vai receber uma mensagem e a área de memória na qual vai armazenar a mensagem e o endereço do soquete remetente (novamente, uma referência para uma estrutura que o contém). A chamada *recvfrom* coleta a primeira mensagem da fila para o soquete ou, se a fila estiver vazia, espera até que uma mensagem chegue.

A comunicação ocorre apenas quando uma operação *sendto* em um processo endereça sua mensagem para o soquete usado por uma operação *recvfrom* em outro processo. Na comunicação cliente-servidor, não há necessidade de que os servidores tenham conhecimento anterior dos endereços de soquete dos clientes, pois a operação *recvfrom* fornece o endereço do remetente junto com cada mensagem que distribui. As propriedades da comunicação por datagrama no UNIX são iguais às descritas na Seção 4.2.3.

4.6.2 Comunicação por fluxo

Para usar o protocolo TCP, dois processos devem primeiro estabelecer uma conexão entre seus pares de soquetes. A organização é assimétrica, pois um dos soquetes estará captando uma requisição de conexão e o outro estará solicitando uma conexão, conforme descrito na Seção 4.2.4. Quando dois soquetes estiverem conectados, eles poderão ser usados para transmitir dados em ambas as direções (ou em uma delas). Isto é, eles se comportarão como fluxos, no sentido de que os dados disponíveis são lidos imediatamente, na mesma ordem em que foram gravados e que não há informação sobre o tamanho da mensagem. Entretanto, existe uma fila vinculada no soquete receptor e o processo destino bloqueia, caso a fila esteja vazia; se ela estiver cheia é o processo remetente que é bloqueado.

Para a comunicação entre clientes e servidores, os clientes solicitam conexões e o servidor as aceita. Quando uma conexão é aceita, o UNIX cria automaticamente um novo soquete e o une com o soquete do cliente para que o servidor possa continuar "escutando" os pedidos de conexão de outros clientes pelo soquete original. Dois soquetes de fluxo conectados podem ser usados na comunicação subseqüente, até que a conexão seja fechada.

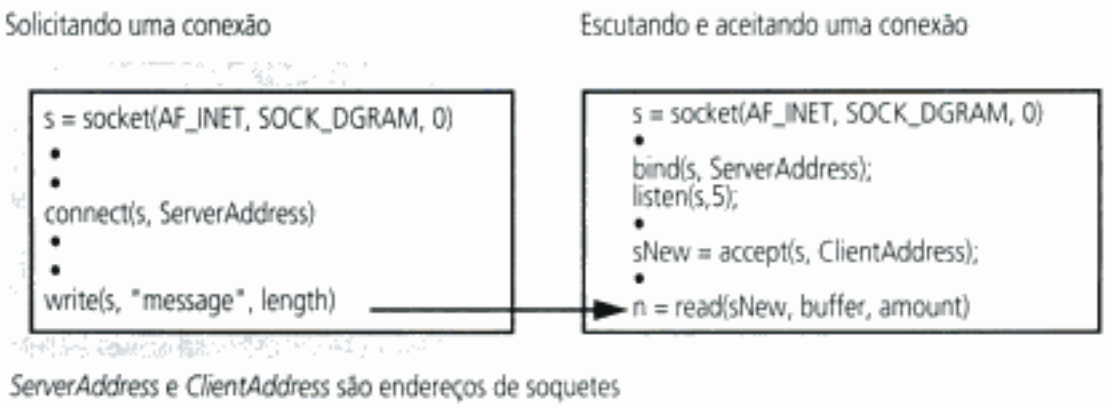


Figura 4.22 **Usando soquetes para comunicação por fluxos.**

A comunicação por fluxo está ilustrada na Figura 4.22, na qual os detalhes dos argumentos estão simplificados. A figura não mostra o servidor fechando o soquete que usa para "escutar" os pedidos de conexão. Normalmente, um servidor primeiro "escuta" e "aceita" uma conexão e depois cria um novo processo para se comunicar com o cliente. Nesse meio-tempo, ele continuaria "escutando" pedidos de conexão no processo original.

- O servidor ou processo de "escuta" primeiro usa a operação *socket* para criar um soquete de fluxo e a operação *bind* para vincular seu soquete ao endereço de soquete do servidor. O segundo argumento da chamada de sistema *socket* é dado como SOCK_STREAM para indicar que é exigida comunicação por fluxo. Se o terceiro argumento for deixado como zero, o protocolo TCP/IP será selecionado automaticamente. Ele usa a operação *listen* para "escutar" nesse soquete os pedidos de conexões do cliente. O segundo argumento da chamada de sistema *listen* especifica o número máximo de pedidos de conexões que pode ser enfileirado nesse soquete.
- O servidor usa a chamada de sistema *accept* para aceitar uma conexão solicitada por um cliente e para obter um novo soquete para comunicação com esse cliente. O soquete original é usado para aceitar conexões de outros clientes.
- O processo cliente usa a operação *socket* para criar um soquete de fluxo e depois usa a chamada de sistema *connect* para solicitar uma conexão por meio do endereço de soquete do processo de "escuta" (servidor). Como a chamada *connect* vincula automaticamente um nome de soquete ao soquete do chamador, uma vinculação antecipada é desnecessária.
- Após uma conexão ter sido estabelecida, os dois processos podem então usar as operações *write* e *read* em seus respectivos soquetes, para enviar e receber seqüências de bytes por meio da conexão. A operação *write* é semelhante à operação de escrita em arquivos; ela especifica a mensagem a ser enviada para um soquete. Essa mensagem é repassada ao protocolo TCP/IP e o número real de caracteres enviados é retornado. A operação *read* recebe alguns caracteres em seu buffer e retorna o número de caracteres recebidos.

As propriedades da comunicação por fluxo no UNIX são iguais àquelas descritas na Seção 4.2.4.

4.7 Resumo

A primeira seção deste capítulo mostrou que os protocolos Internet fornecem dois blocos básicos a partir dos quais os protocolos em nível de aplicativos podem ser construídos. Há um compromisso interessante entre os dois protocolos: o protocolo UDP fornece um recurso simples de passagem de mensagem, que sofre de falhas por omissão, mas não tem penalidades de desempenho incorporadas. Por outro lado, em boas condições, o protocolo TCP garante a entrega das mensagens, mas à custa de mensagens adicionais e com latência e custos de armazenamento mais altos.

A segunda seção mostrou três estilos alternativos de empacotamento. O CORBA e seus predecessores optam por empacotar os dados para uso pelos destinatários que têm conhecimento anterior dos tipos de seus componentes. Em contraste, quando a linguagem Java serializa dados, ela inclui informações completas sobre os tipos de seu conteúdo, permitindo ao destinatário reconstruí-los puramente a partir do conteúdo. A linguagem XML, assim como a Java, inclui informações completas sobre os tipos de dados. Outra grande diferença é que o CORBA exige uma especificação dos tipos dos dados a serem empacotados (no IDL) para gerar os métodos de empacotamento e desempacotamento, enquanto a linguagem Java usa reflexão para serializar e desserializar objetos. Uma variedade de meios é usada para gerar código XML, dependendo do contexto. Por exemplo, muitas linguagens de programação, incluindo a Java, fornecem processadores para fazer a transformação entre objetos em nível de XML e de linguagem.

A seção sobre o protocolo requisição-resposta mostrou que um protocolo de propósito especial eficiente para sistemas distribuídos pode ser baseado nos datagramas UDP. A mensagem de resposta constitui uma confirmação para a mensagem de requisição, evitando assim as sobrecargas das mensagens adicionais de confirmação. O protocolo pode se tornar mais confiável, se necessário. Conforme foi visto, não há garantia de que o envio de uma mensagem de requisição resulte na execução de um método – isso pode ser suficiente para algumas aplicações. Mas uma confiabilidade adicional pode ser conseguida com o uso de identificadores e retransmissões de mensagem, para garantir que um método finalmente seja executado. Para serviços com operações idempotentes, isso é suficiente. Entretanto, outras aplicações exigem que mensagens de resposta sejam transmitidas sem uma nova execução do método requisitado no pedido, o que pode ser conseguido com a ajuda de um histórico. Isso ilustra o ponto de que é uma boa idéia construir protocolos de acordo com as diferentes classes de aplicativos, em vez de construir um único protocolo ultra-confiável para uso geral, pois ele poderia ter um mau desempenho no caso normal, no qual raramente ocorrem falhas.

Mensagens *multicast* são usadas na comunicação entre os membros de um grupo de processos. O *multicast* IP é disponível tanto para redes locais como para a Internet. O *multicast* IP tem a mesma semântica de falhas que os datagramas UDP, porém, apesar de sofrer de falhas por omissão, é uma ferramenta útil para muitas aplicações *multicast*. Algumas aplicações têm requisitos mais restritos – em particular, o de que a distribuição *multicast* deva ser atômica; isto é, ela deve ter uma distribuição do tipo tudo ou nada. Outros requisitos do *multicast* estão relacionados ao ordenamento das mensagens, o mais exigente dos quais impõe que todos os membros de um grupo devem receber todas as mensagens na mesma ordem.

Exercícios

4.1 É concebivelmente útil que uma porta tenha vários receptores? *páginas 127–128*

4.2 Um servidor cria uma porta que utiliza para receber pedidos dos clientes. Discuta os problemas de projeto relativos ao relacionamento entre o nome dessa porta e os nomes usados pelos clientes. *páginas 127–128*

4.3 Os programas das Figuras 4.3 e 4.4 estão disponíveis no endereço [www.cdk4.net/ipc]. Utilize-os para fazer um kit de testes para determinar as condições nas quais os datagramas às vezes são descartados. Dica: o programa cliente deve ser capaz de variar o número de mensagens enviadas e seus tamanhos; o servidor deve detectar quando uma mensagem de um cliente em particular é perdida. *páginas 129–130*

4.4 Use o programa da Figura 4.3 para fazer um programa cliente que leia repetidamente uma linha de entrada do usuário, a envie para o servidor em uma mensagem datagrama UDP e depois receba uma mensagem do servidor. O cliente estabelece um tempo limite em seu soquete para que possa informar o usuário quando o servidor não responder. Teste esse programa cliente com o servidor da Figura 4.4. *páginas 129–130*

4.5 Os programas das Figuras 4.5 e 4.6 estão disponíveis no endereço [www.cdk4.net/ipc]. Modifique-os de modo que o cliente leia repetidamente uma linha de entrada do usuário e a escreva no fluxo. O servidor deve ler repetidamente o fluxo, imprimindo o resultado de cada leitura. Faça uma comparação entre o envio de dados em mensagens de datagrama UDP e por meio de um fluxo. *página 133*

4.6 Use os programas desenvolvidos no Exercício 4.5 para testar o efeito sobre o remetente quando o receptor falha e vice-versa. *página 133*

4.7 O XDR da Sun empacota dados convertendo-os para uma forma *big-endian* antes de transmiti-los. Discuta as vantagens e desvantagens desse método, quando comparado com o CDR do CORBA. *página 138*

4.8 O XDR da Sun alinha cada valor primitivo em um limite de quatro bytes, enquanto o CDR da CORBA alinha um valor primitivo de tamanho *n* em um limite de *n* bytes. Discuta os compromissos na escolha dos tamanhos ocupados pelos valores primitivos. *página 138*

4.9 Por que não há nenhuma tipagem de dados explícita no CDR do CORBA? *página 138*

4.10 Escreva um algoritmo, em pseudocódigo, para descrever o procedimento de serialização mostrado na Seção 4.3.2. O algoritmo deve mostrar quando identificadores são definidos ou substituídos por classes e instâncias. Descreva a forma serializada que seu algoritmo produziria para uma instância da seguinte classe *Couple*.

```
class Couple implements Serializable{
        private Person one;
        private Person two;
        public Couple(Person a, Person b) {
            one = a;
            two = b;
        }
}
```

página 138

4.11 Escreva um algoritmo, em pseudocódigo, para descrever a desserialização da forma serializada produzida pelo algoritmo definido no Exercício 4.10. Dica: use reflexão para criar uma classe a partir de seu nome, um construtor a partir de seus tipos de parâmetro e uma nova instância de um objeto a partir do construtor e dos valores de argumento. *página 139–140*

4.12 Por que dados binários não podem ser representados diretamente em XML, por exemplo, como valores em Unicode? Os elementos XML podem transportar strings representados como *base64*. Discuta as vantagens ou desvantagens de usar esse método para representar dados binários. *página 141–142*

4.13 Defina uma classe cujas instâncias representem referências de objeto remoto. Ela deve conter informações semelhantes àquelas mostradas na Figura 4.13 e deve fornecer métodos de acesso necessários para o protocolo requisição-resposta. Explique como cada um dos métodos de acesso será usado por esse protocolo. Dê uma justificativa para o tipo escolhido para a variável de instância que contém informações sobre a interface do objeto remoto. *página 145–146*

4.14 Defina uma classe cujas instâncias representem mensagens de requisição e resposta, conforme ilustrado na Figura 4.16. A classe deve fornecer dois construtores, um para as mensagens de requisição e outro para as mensagens de resposta, mostrando como o identificador de requisição é atribuído. Ela também deve fornecer um método para empacotá-la em um array de bytes e para desempacotar um array de bytes em uma instância. *página 145–146*

4.15 Programe cada uma das três operações do protocolo requisição-resposta da Figura 4.15, usando comunicação UDP, mas sem adicionar quaisquer medidas de tolerância a falhas. Você deve usar as classes que definiu nos Exercícios 4.13 e 4.14. *página 146–147*

4.16 Faça um esboço da implementação de um servidor, mostrando como as operações *getRequest* e *sendReply* são usadas por um servidor que cria uma nova *thread* para executar cada requisição de cliente. Indique como o servidor copiará a *requestId* da mensagem de requisição na mensagem de resposta e como obterá o endereço IP e a porta do cliente. *página 146–147*

4.17 Defina uma nova versão do método *doOperation* que estabeleça um *timeout* para a espera pela mensagem de resposta. Após o *timeout*, a mensagem de requisição é transmitida *n* vezes. Se ainda não houver resposta, ela informa ao chamador. *página 148–149*

4.18 Descreva um cenário no qual um cliente poderia receber uma resposta de uma chamada anterior. *página 146–147*

4.19 Descreva as maneiras pelas quais o protocolo requisição-resposta mascara a heterogeneidade dos sistemas operacionais e das redes de computador. *página 146–147*

4.20 Discuta se as seguintes operações são *idempotentes*:

- Pressionar o botão de subida de um elevador;
- Gravar dados em um arquivo;
- Anexar dados em um arquivo.

É uma condição necessária para a idempotência que a operação não deva estar associada a qualquer estado? *página 148–149*

4.21 Explique as escolhas de projeto relevantes para minimizar o volume de dados de resposta mantidos em um servidor. Compare os requisitos de armazenamento quando são usados os protocolos RR e RRA. *página 148–149*

4.22 Suponha que o protocolo RRA esteja sendo usado. Por quanto tempo os servidores devem manter os dados de resposta não reconhecidos? Os servidores devem enviar a resposta repetidamente, em uma tentativa de receber uma confirmação? *página 148–149*

4.23 Por que o número de mensagens trocadas em um protocolo poderia ser mais significativo para o desempenho do que o volume total de dados enviados? Projete uma variante do protocolo RRA na qual a confirmação seja transportada (isto é, transmitida em uma mesma mensagem) na próxima requisição, quando apropriado, e enviada como uma mensagem separada, caso contrário. (Dica: use um temporizador extra no cliente.) *página 148–149*

4.24 O *multicast* IP fornece um serviço que sofre de falhas por omissão. Faça um kit de testes, baseado no programa da Figura 4.20, para descobrir as condições sob as quais uma mensagem *multicast* às vezes é eliminada por um dos membros do grupo *multicast*. O kit de testes deve ser projetado de forma a permitir a existência de vários processos remetentes. *página 154–155*

4.25 Esboce o projeto de um esquema que utilize retransmissões de mensagem *multicast* IP para superar o problema das mensagens descartadas. Seu esquema deve levar em conta os seguintes pontos:

i) podem existir vários remetentes;

ii) geralmente apenas uma pequena parte das mensagens é descartada;

iii) ao contrário do protocolo de requisição-resposta, os destinatários podem não enviar uma mensagem necessariamente dentro de um limite de tempo em particular.

Suponha que as mensagens que não são descartadas chegam na ordem do remetente. *página 155–156*

4.26 Sua solução para o Exercício 4.25 deve ter superado o problema das mensagens descartadas no *multicast* IP. Em que sentido sua solução difere da definição de *multicast* confiável? *página 155–156*

4.27 Imagine um cenário no qual mensagens *multicast* enviadas por diferentes clientes são entregues em ordens diferentes em dois membros do grupo. Suponha que esteja em uso alguma forma de retransmissões de mensagem, mas que as mensagens que não são descartadas cheguem na ordem do remetente. Sugira o modo como os destinos poderiam remediar essa situação. *página 155–156*

4.28 Defina a semântica e projete um protocolo para uma forma de interação tipo requisição-resposta de grupo usando, por exemplo, *multicast* IP. *páginas 146–147, 154–155*

Objetos Distribuídos e Invocação Remota 5

Neste capítulo, apresentaremos a comunicação entre objetos distribuídos feita por meio da invocação a método remoto (RMI – *Remote Method Invocation*). Os objetos que podem receber invocações a métodos remotos são chamados de objetos remotos e implementam uma interface remota. Devido à possibilidade de falhas independentes dos objetos invocadores e invocados, as RMIs têm semânticas diferentes das invocações a métodos locais. Elas podem ser feitas de modo a ser bastante semelhantes às invocações locais, mas a transparência total não é necessariamente desejável. O código para empacotar e desempacotar argumentos e para enviar mensagens de requisição e resposta pode ser gerado automaticamente por um compilador de interface, a partir da definição da interface remota.

A chamada de procedimento remoto (RPC – *Remote Procedure Call*) está para a RMI assim como a chamada de procedimento está para a invocação de objetos. Ela será descrita sucintamente e ilustrada por meio de um estudo de caso da RPC Sun.

Os sistemas distribuídos baseados em eventos permitem que os objetos se inscrevam nos eventos que estão ocorrendo nos objetos remotos de interesse e, por sua vez, recebam notificações, quando tais eventos ocorrerem. Os eventos e notificações oferecem uma maneira para que objetos heterogêneos se comuniquem de forma assíncrona. A especificação de evento distribuído do Jini será apresentada como um estudo de caso.

O uso de RMI será ilustrado em um estudo de caso da RMI Java.

O Capítulo 20 contém um estudo de caso sobre o CORBA, que inclui a RMI do CORBA e o CORBA *Event Service*.

5.1 Introdução

Este capítulo fala a respeito dos modelos de programação para aplicativos distribuídos – isto é, aqueles aplicativos compostos de programas que estão em cooperação, executados em vários processos diferentes. Tais programas precisam executar (invocar) operações em outros processos, freqüentemente sendo executados em diferentes computadores. Para se conseguir isso, alguns modelos de programação foram estendidos para aplicação em programas distribuídos:

- O mais antigo, e talvez mais conhecido deles, foi a extensão do modelo de chamada de procedimentos convencional para o modelo de *chamada de procedimento remoto*, que permite aos programas clientes chamarem procedimentos de programas servidores que estejam sendo executados em processos separados e, geralmente, em computadores diferentes do cliente.
- Nos anos 90, o modelo da programação baseada em objetos foi ampliado para permitir que objetos de diferentes processos se comunicassem por intermédio da *invocação a método remoto* (RMI). A RMI é uma extensão da invocação a método local que permite a um objeto que está em um processo invocar os métodos de um objeto que está em outro processo.
- O modelo da programação baseada em eventos permite que objetos recebam notificação dos eventos que ocorrem em outros objetos, nos quais têm interesse. Esse modelo foi estendido para permitir a escrita de programas distribuídos baseados em eventos.

Note que usamos o termo "RMI" para nos referirmos à invocação a método remoto de uma maneira genérica – isso não deve ser confundido com exemplos particulares de invocação a método remoto, como a RMI Java. Uma grande parte do software de sistemas distribuídos atuais é escrita em linguagens orientadas a objetos e a RPC pode ser entendida em relação à RMI. Portanto, este capítulo se concentra nos paradigmas da RMI e dos eventos, cada um dos quais se aplica aos objetos distribuídos. A comunicação entre objetos distribuídos será apresentada na Seção 5.2, seguida por uma discussão sobre o projeto e a implementação da RMI. Um estudo de caso sobre a RMI Java será apresentado na Seção 5.5. A RPC será discutida no contexto de um estudo de caso sobre a RPC Sun, na Seção 5.3; a RPC também é usada por serviços web que serão discutidos no Capítulo 19. Os eventos e as notificações distribuídas serão discutidos na Seção 5.4. Um estudo de caso mais aprofundado sobre o CORBA será apresentado no Capítulo 20.

Middleware ◊ O software que fornece um modelo de programação acima dos blocos de construção básicos de processos e de passagem de mensagens é chamado de *middleware*. A camada de *middleware* usa protocolos baseados em mensagens entre processos para fornecer suas abstrações de mais alto nível, como as invocações e eventos remotos, conforme ilustrado na Figura 5.1. Por exemplo, a abstração da invocação a método remoto é baseada no protocolo requisição-resposta discutido na Seção 4.4.

Um aspecto importante do *middleware* é a transparência da localização e a independência dos detalhes dos protocolos de comunicação, sistemas operacionais e hardware de computador. Algumas formas de *middleware* permitem que componentes distintos sejam escritos em diferentes linguagens de programação.

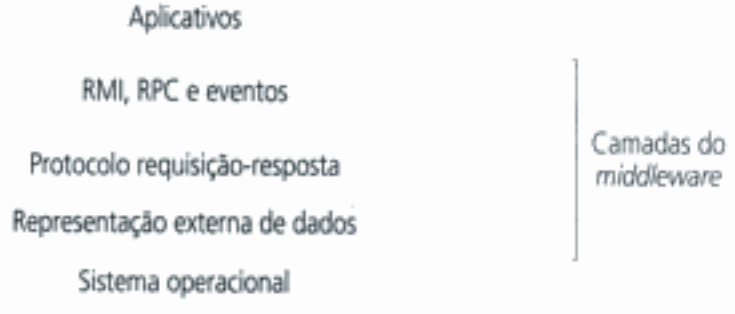


Figura 5.1 Camada do *middleware*.

Transparência de localização: na RPC, o cliente que chama um procedimento não pode identificar se ele é executado no mesmo processo ou em um outro processo, possivelmente em um computador diferente. O cliente também não precisa saber a localização do servidor. Analogamente, na RMI, o objeto que está fazendo a invocação não pode identificar se o objeto invocado é local ou não, e não precisa saber sua localização. Além disso, nos programas distribuídos baseados em eventos, os objetos que geram eventos e os objetos que recebem notificações desses eventos não precisam conhecer a localização uns dos outros.

Protocolos de comunicação: os protocolos que suportam as abstrações de *middleware* são independentes dos protocolos de transporte subjacentes. Por exemplo, o protocolo requisição-resposta pode ser implementado sobre UDP ou sobre TCP.

Hardware de computador: três padrões aceitos para representação externa de dados foram descritos na Seção 4.3. Os padrões são usados no empacotamento e desempacotamento de mensagens. Eles ocultam as diferenças devido às arquiteturas de hardware, como a ordem dos bytes.

Sistemas operacionais: as abstrações de mais alto nível fornecidas pela camada de *middleware* são independentes dos sistemas operacionais subjacentes.

Uso de várias linguagens de programação: alguns *middlewares* são projetados de forma a permitir que os aplicativos distribuídos utilizem mais do que uma linguagem de programação. Em particular, o CORBA (veja o Capítulo 20) permite que clientes escritos em uma linguagem invoquem métodos em objetos que estão em programas servidores escritos em outra linguagem. Isso é obtido pelo uso de uma *linguagem de definição de interface* (IDL – *Interface Description Language*) para definir interfaces. As IDLs serão discutidas na próxima seção.

5.1.1 Interfaces

A maioria das linguagens de programação modernas fornece uma maneira de organizar um programa como um conjunto de módulos que podem se comunicar. A comunicação entre os módulos pode ser feita por meio de chamadas de procedimentos entre eles ou pelo acesso direto às variáveis de outro módulo. Para controlar as interações possíveis entre os módulos, uma *interface* explícita é definida para cada módulo. A interface de um módulo especifica os procedimentos e as variáveis que podem ser acessadas a partir de outros módulos. Os módulos são implementados de modo a ocultar todas as informações sobre eles, exceto as que estiverem disponíveis por intermédio de sua interface. Desde que sua interface permaneça a mesma, a implementação pode ser alterada sem afetar os usuários do módulo.

Interfaces em sistemas distribuídos ◊ Em um programa distribuído, os módulos podem ser executados em processos separados. Não é possível um módulo ser executado em um processo e acessar as variáveis de um módulo em outro processo. Portanto, a interface de um módulo destinado a RPC, ou RMI, não pode especificar acesso direto às variáveis. Note que as interfaces IDL do CORBA podem especificar atributos, o que parece violar essa regra. Entretanto, os atributos não são acessados diretamente, mas por meio de alguns procedimentos de obtenção e configuração (*getter* e *setter*), adicionadas automaticamente na interface.

Os mecanismos de passagem de parâmetros (por exemplo, chamada por valor e chamada por referência) usados na chamada de procedimento local, não são convenientes quando o chamador e o procedimento estão em processos diferentes. A especificação de um procedimento, ou método na interface, de um módulo em um programa distribuído descreve os parâmetros como sendo *de entrada* ou *de saída* ou, às vezes, ambos. Os parâmetros *de entrada* são passados para o módulo remoto pelo envio dos valores dos argumentos na mensagem de requisição e, então, repassados como argumentos para a operação a ser executada no servidor. Os parâmetros *de saída* são retornados na mensagem de resposta e são usados como resultado da chamada ou para substituir os valores das variáveis correspondentes no ambiente de chamada. Quando um parâmetro é usado tanto para entrada como para saída, o valor deve ser transmitido nas mensagens de requisição e de resposta.

Outra diferença entre módulos locais e remotos é que as variáveis do tipo ponteiros de um processo não são válidas em outro processo remoto. Portanto, os ponteiros não podem ser passados como argumentos, nem retornados como resultado de chamadas para módulos remotos.

Os dois próximos parágrafos discutem as interfaces usadas no modelo cliente-servidor original da RPC e no modelo de objeto distribuído da RMI:

Interfaces de serviço: no modelo cliente-servidor, cada servidor fornece um conjunto de procedimentos que estão disponíveis para uso pelos clientes. Por exemplo, um servidor de arquivos forneceria procedimentos para ler e gravar arquivos. O termo *interface de serviço* é usado para se referir à especificação dos procedimentos oferecidos por um servidor, definindo os tipos dos argumentos de entrada e saída de cada um dos procedimentos.

Interfaces remotas: no modelo de objeto distribuído, uma interface remota especifica os métodos de um objeto que estão disponíveis para invocação por parte dos objetos de outros processos, definindo os tipos dos argumentos de entrada e saída de cada um deles. Entretanto, a grande diferença é que os métodos nas interfaces remotas podem passar objetos como argumentos e resultados de métodos. Além disso, também podem ser passadas referências para objetos remotos – elas não devem ser confundidas com os ponteiros, que se referem a posições específicas de memória. (A Seção 4.3.4 descreveu o conteúdo das referências de objeto remoto.)

Nem as interfaces de serviço, nem as interfaces remotas, podem especificar acesso direto às variáveis. Neste último caso, isso proíbe o acesso direto às variáveis de instância de um objeto.

Linguagens de definição de interface ◊ Um mecanismo de RMI pode ser integrado a uma linguagem de programação em particular, caso inclua uma notação adequada para definir interfaces, permitindo que os parâmetros de entrada e saída sejam mapeados no uso normal de parâmetros da linguagem. A RMI Java é um exemplo no qual um mecanismo de RMI foi adicionado a uma linguagem de programação orientada a objetos. Essa estratégia é útil quando todas as partes de um aplicativo distribuído podem ser escritas na mesma linguagem. Ela também é conveniente, pois permite que o programador utilize uma única linguagem para invocação local e remota.

Entretanto, muitos serviços úteis existentes são escritos em C++ e em outras linguagens. Seria vantajoso permitir que programas escritos em uma variedade de linguagens, incluindo Java, os acessassem de forma remota. As *linguagens de definição de interface* (ou IDLs) são projetadas para permitir que objetos implementados em diferentes linguagens invoquem uns aos outros. Uma IDL fornece uma notação para definir interfaces, na qual cada um dos parâmetros de um método pode ser descrito como sendo de *entrada* ou *saída*, além de ter seu tipo especificado.

A Figura 5.2 mostra um exemplo simples de IDL do CORBA. A estrutura *Person* é a mesma usada para ilustrar o empacotamento, na Seção 4.3.1. A interface chamada *PersonList* especifica os métodos disponíveis para RMI em um objeto remoto que implementa essa interface. Por exemplo, o método *addPerson* especifica seu argumento como *in*, significando que se trata de um argumento de

```
// Arquivo de entrada Person.idl
struct Person {
    string name;
    string place;
    long year;
};
interface PersonList {
    readonly attribute string listname;
    void addPerson(in Person p);
    void getPerson(in string name, out Person p);
    long number();
};
```

Figura 5.2 Exemplo de IDL do CORBA.

entrada (*input*); e o método *getPerson*, que recupera uma instância de *Person* pelo nome, especifica seu segundo argumento como *out*, significando que se trata de um argumento de *saída* (*output*).

Nossos estudos de caso incluem a IDL do CORBA como exemplo de IDL para RMI (no Capítulo 20) e a XDR da Sun como exemplo de IDL para RPC (na Seção 5.3). A linguagem de descrição de serviços web (WSDL – *Web Services Description Language*) é projetada para permitir RPC na Internet (veja a Seção 19.3).

Outros exemplos incluem a linguagem de definição de interface para o sistema RPC no DCE (*Distributed Computing Environment*) da OSF [OSF 1997], que utiliza a sintaxe da linguagem C e é chamada IDL; e a IDL do DCOM, que é baseada na IDL do DCE [Box 1998] e usada no *Distributed Component Object Model* da Microsoft.

5.2 Comunicação entre objetos distribuídos

O modelo baseado em objetos para um sistema distribuído estende o modelo suportado pelas linguagens de programação orientadas a objetos para aplicá-lo em objetos distribuídos. Esta seção trata da comunicação entre objetos distribuídos por meio de RMI. O material é apresentado sob os seguintes tópicos:

O modelo de objeto: uma breve revisão dos aspectos relevantes do modelo de objeto, apropriado para os leitores que tenham conhecimento básico de uma linguagem de programação orientada a objetos, por exemplo Java ou C++.

Objetos distribuídos: uma apresentação dos sistemas distribuídos baseados em objetos, afirmando que o modelo de objeto é muito apropriado para os sistemas distribuídos.

O modelo de objeto distribuído: uma discussão sobre as extensões feitas no modelo de objeto, necessárias para suportar objetos distribuídos.

Problemas de projeto: um conjunto de argumentos sobre as alternativas de projeto:

1. As invocações locais são executadas exatamente uma vez, mas qual semântica apropriada é possível para invocações remotas?
2. Como a semântica da RMI pode ser semelhante à da invocação a método local e quais diferenças não podem ser eliminadas?

Implementação: uma explicação de como uma camada de *middleware* acima do protocolo requisição-resposta pode ser projetada de modo a suportar RMI entre objetos distribuídos em nível de aplicativo.

Coleta de lixo (*garbage collection*) *distribuída*: uma apresentação de um algoritmo para coleta de lixo distribuída, conveniente para uso com a implementação de RMI.

5.2.1 O modelo de objeto

Um programa orientado a objetos, por exemplo em Java ou C++, consiste em um conjunto de objetos interagindo, cada um dos quais composto de um conjunto de dados e um conjunto de métodos. Um objeto se comunica com outros objetos invocando seus métodos, geralmente passando argumentos e recebendo resultados. Os objetos podem encapsular seus dados e o código de seus métodos. Algumas linguagens, por exemplo Java e C++, permitem que os programadores definam objetos cujas variáveis de instância podem ser acessadas diretamente. Mas, para uso em um sistema de objeto distribuído, os dados de um objeto devem ser acessíveis somente por intermédio de seus métodos.

Referências de objeto ◊ Os objetos podem ser acessados por meio de referências de objeto. Por exemplo, em Java, uma variável que parece conter um objeto, na verdade contém uma referência para esse objeto. Para invocar um método em um objeto, são fornecidos a referência do objeto e o nome do método, junto com os argumentos necessários. Às vezes, o objeto cujo método é invocado é chamado de *alvo*, *destino* e, às vezes, de *receptor*. As referências de objeto são valores de primeira classe, significando que eles podem, por exemplo, ser atribuídos a variáveis, passados como argumentos e retornados como resultados de métodos.

Interfaces ◊ Uma interface fornece a definição das assinaturas de um conjunto de métodos (isto é, os tipos de seus argumentos, valores de retorno e exceções), sem especificar sua implementação. Um objeto fornecerá uma interface em particular, caso sua classe contenha código que implemente os métodos dessa interface. Em Java, uma classe pode implementar várias interfaces e os métodos de uma interface podem ser implementados por qualquer classe. Uma interface também define os tipos que podem ser usados para declarar o tipo de variáveis, ou dos parâmetros, e valores de retorno dos métodos. Note que as interfaces não têm construtores.

Ações ◊ Em um programa orientado a objetos, a ação é iniciada por um objeto invocando um método em outro objeto. Uma invocação pode incluir informações adicionais (argumentos) necessárias para executar o método. O receptor executa o método apropriado e depois retorna o controle para o objeto que fez a invocação, às vezes fornecendo um resultado. A ativação de um método pode ter três efeitos:

1. o estado do receptor pode ser alterado;
2. um novo objeto pode ser instanciado; por exemplo, pelo uso de um construtor em Java ou C++; e
3. outras invocações podem ocorrer nos métodos de outros objetos.

Como uma invocação pode levar a mais invocações a métodos em outros objetos, uma ação é um encadeamento de invocações relacionadas de métodos, cada uma com seu respectivo retorno. Esta explicação não leva em conta as exceções.

Exceções ◊ Os programas podem encontrar muitos tipos de erros e condições inesperadas, de diversos graus de gravidade. Durante a execução de um método, muitos problemas diferentes podem ser descobertos: por exemplo, valores inconsistentes nas variáveis do objeto ou falhas nas tentativas de ler ou gravar arquivos ou soquetes de rede. Quando os programadores precisam inserir testes em seu código para tratar de todos os casos incomuns ou errôneos possíveis, isso diminui a clareza do caso normal. As exceções proporcionam uma maneira clara de tratar com condições de erro, sem complicar o código. Além disso, cada cabeçalho de método lista, explicitamente, como exceções as condições de erro que pode encontrar, permitindo que os usuários desse método as tratem adequadamente. Pode ser definido um bloco de código para *disparar* uma exceção, quando surgirem condições inesperadas ou erros em particular. Isso significa que o controle passa para outro bloco de código, que *captura* a exceção. O controle não retorna para o lugar onde a exceção foi disparada.

Coleta de lixo (*garbage collection*) ◊ É necessário fornecer uma maneira de liberar o espaço em memória ocupado pelos objetos, quando eles não são mais necessários. Uma linguagem (por exemplo, Java) que pode detectar automaticamente quando um objeto não está mais acessível, recupera o espaço em memória e o torna disponível para alocação por outros objetos. Esse processo é chamado de *coleta de lixo* ou, em inglês, *garbage collection*. Quando uma linguagem não suporta coleta de lixo (por exemplo, C++), o programador tem de se preocupar com a liberação do espaço alocado para os objetos. Isso pode ser uma importante fonte de erros.

5.2.2 Objetos distribuídos

O estado de um objeto consiste nos valores de suas variáveis de instância. No paradigma baseado em objetos, o estado de um programa é dividido em partes separadas, cada uma das quais associada a um objeto. Como os programas baseados em objetos são logicamente particionados, a distribuição física dos objetos em diferentes processos ou computadores de um sistema distribuído é uma extensão natural.

Os sistemas de objetos distribuídos podem adotar a arquitetura cliente-servidor. Nesse caso, os objetos são gerenciados pelos servidores e seus clientes invocam seus métodos usando invocação a método remoto, RMI. Na RMI, a requisição de um cliente para invocar um método de um objeto é enviada em uma mensagem para o servidor que gerencia o objeto. A invocação é realizada pela execução de um método do objeto no servidor e o resultado é retornado para o cliente em outra mensagem. Para permitir encadeamentos de invocações relacionadas, os objetos nos servidores podem se tornar clientes de objetos em outros servidores.

Os objetos distribuídos podem adotar outros modelos arquiteturais. Por exemplo, os objetos podem ser replicados para usufruir das vantagens normais da tolerância a falhas e do melhor desempenho, e os objetos podem ser migrados com o intuito de melhorar seu desempenho e sua disponibilidade.

O fato de ter objetos clientes e servidores em diferentes processos impõe o encapsulamento. Isso significa que o estado de um objeto pode ser acessado somente pelos métodos do objeto, implicando que não é possível a métodos não autorizados agirem livremente sobre o estado. Por exemplo, a possibilidade de RMIs concorrentes, a partir de objetos em diferentes computadores, significa que um objeto pode ser acessado de forma concorrente. Portanto, surge a possibilidade de conflito de acessos. Entretanto, o fato de os dados de um objeto serem acessados somente pelos seus próprios métodos permite que os objetos forneçam métodos para protegerem-se contra acessos incorretos. Nesse caso, por exemplo, eles podem usar primitivas de sincronização, como variáveis de condição, para proteger o acesso às suas variáveis de instância.

Outra vantagem de tratar o estado compartilhado de um programa distribuído como um conjunto de objetos é que um objeto pode ser acessado via RMI ou ser copiado em uma cache local e acessado diretamente, desde que a implementação da classe esteja disponível de forma local.

O fato dos objetos serem acessados somente por meio de seus métodos oferece outra vantagem para sistemas heterogêneos, pois diferentes formatos de dados podem ser usados em diferentes instalações – esses formatos não serão notados pelos clientes que utilizam RMI para acessar os métodos dos objetos.

5.2.3 O modelo de objeto distribuído

Esta seção discute as extensões feitas no modelo de objeto para torná-lo aplicável aos objetos distribuídos. Cada processo contém um conjunto de objetos, alguns dos quais podem receber invocações locais e remotas, enquanto os outros objetos podem receber somente invocações locais, como mostrado na Figura 5.3. As invocações a métodos entre objetos em diferentes processos, sejam no mesmo computador ou não, são conhecidas como *invocações a métodos remotos*. As invocações a métodos entre objetos no mesmo processo são invocações a métodos locais.

Nos referimos aos objetos que podem receber invocações remotas como *objetos remotos*. Na Figura 5.3, B e F são objetos remotos. Todos os objetos podem receber invocações locais de outros objetos que contenham referências a eles. Por exemplo, o objeto C deve ter uma referência ao objeto E para poder invocar um de seus métodos. Os dois conceitos fundamentais a seguir estão no centro do modelo de objeto distribuído:

Referência de objeto remoto: outros objetos podem invocar os métodos de um objeto remoto, se tiverem acesso à sua *referência de objeto remoto*. Por exemplo, na Figura 5.3, uma referência de objeto remoto de B deve estar disponível para A.

Interface remota: todo objeto remoto tem uma interface remota especificando qual de seus métodos pode ser invocado de forma remota. Por exemplo, os objetos B e F devem ter interfaces remotas.

Os parágrafos a seguir discutem as referências de objeto remoto, as interfaces remotas e outros aspectos do modelo de objeto distribuído.

Figura 5.3 Invocação a métodos locais e remotos.

Referências de objeto remoto ◊ A noção de referência de objeto é estendida para permitir que qualquer objeto que possa receber uma RMI tenha uma referência de objeto remoto. Uma referência de objeto remoto é um identificador que pode ser usado por todo um sistema distribuído para se referir a um objeto remoto único em particular. Sua representação, que geralmente é diferente da representação de referências de objeto local, foi discutida na Seção 4.3.4. As referências de objeto remoto são análogas às locais, pois:

1. o objeto remoto que vai receber uma invocação a método remoto é especificado pelo *invocador* como uma referência de objeto remoto; e
2. as referências de objeto remoto podem ser passadas como argumentos e resultados de invocações a métodos remotos.

Interfaces remotas ◊ A classe de um objeto remoto implementa os métodos de sua interface remota; por exemplo, como métodos de instância públicos em Java. Objetos em outros processos só podem invocar os métodos pertencentes à interface remota, como mostrado na Figura 5.4. Os objetos locais podem invocar os métodos da interface remota, assim como outros métodos implementados por um objeto remoto. Note que as interfaces remotas, assim como todas as interfaces, não têm construtores.

O sistema CORBA fornece uma linguagem de definição de interface (IDL) que é usada para definir interfaces remotas. Veja, na Figura 5.2, um exemplo de interface remota definida na IDL do CORBA. As classes de objetos remotos e os programas clientes podem ser implementados em qualquer linguagem, como C++, Java ou Python, para a qual esteja disponível um compilador de IDL. Os clientes CORBA não precisam usar a mesma linguagem que o objeto remoto para invocar seus métodos de forma remota.

Na RMI Java, as interfaces remotas são definidas da mesma forma que qualquer outra interface Java. Elas adquirem a capacidade de serem interfaces remotas estendendo uma interface como *Remote*. Tanto a IDL do CORBA (Seção 20.2.3) como a linguagem Java suportam herança múltipla de interfaces, isto é, uma interface pode estender uma ou mais interfaces.

Ações em um sistema de objeto distribuído ◊ Assim como no caso não-distribuído, uma ação é iniciada por uma invocação a método, a qual pode resultar em mais invocações a métodos em outros objetos. Mas, no caso distribuído, os objetos envolvidos em um encadeamento de invocações relacionadas podem estar localizados em diferentes processos ou em diferentes computadores. Quando uma invocação cruza o limite de um processo, ou computador, a RMI é usada e a referência remota do objeto deve estar disponível para o invocador. Na Figura 5.3, o objeto A precisa conter uma referência de objeto remoto para o objeto B. As referências de objeto remoto podem ser obtidas como resultado de invocações a métodos remotos. Por exemplo, na Figura 5.3, o objeto A poderia obter uma referência remota para o objeto F, a partir do objeto B.

Quando uma ação levar à instanciação de um novo objeto, esse objeto normalmente ficará dentro do processo onde a instanciação é feita; por exemplo, onde o construtor foi usado. Se o objeto recentemente instanciado tiver uma interface remota, ele será um objeto remoto com uma referência de objeto remoto.

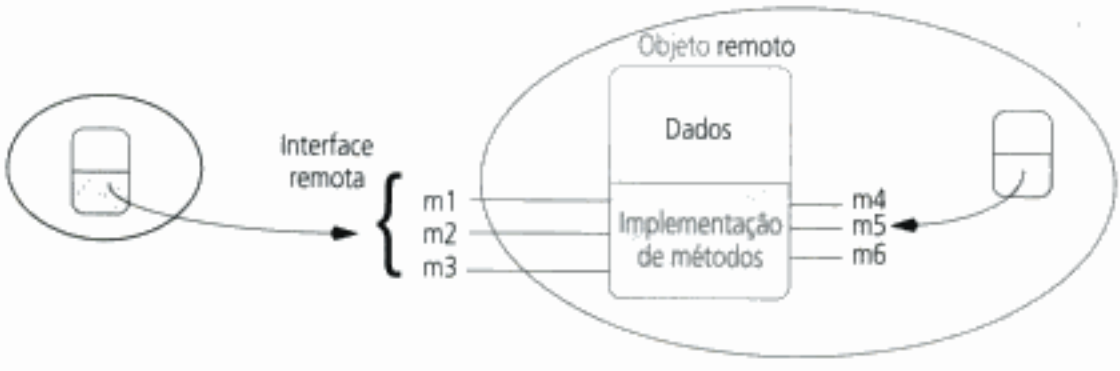


Figura 5.4 Um objeto remoto e sua interface remota.

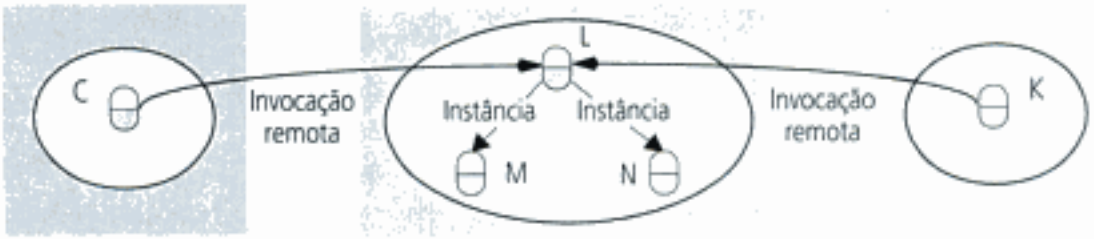


Figura 5.5 Instanciação de objetos remotos.

Os aplicativos distribuídos podem fornecer objetos remotos com métodos para instanciar objetos que podem ser acessados por uma RMI, fornecendo assim, efetivamente, o efeito da instanciação remota de objetos. Suponha, por exemplo, que o objeto L da Figura 5.5 contivesse um método para criar objetos remotos e que as invocações remotas de C e K pudessem levar à instanciação dos objetos M e N, respectivamente.

Coleta de lixo em um sistema de objeto distribuído ◊ Se uma linguagem, por exemplo Java, suporta coleta de lixo, então qualquer sistema RMI associado deve permitir a coleta de lixo de objetos remotos. A coleta de lixo distribuída geralmente é obtida pela cooperação entre o coletor de lixo local existente e um módulo adicionado que realiza uma forma de coleta de lixo distribuída, normalmente baseada em contagem de referência. A Seção 5.2.6 descreverá esse esquema em detalhes. Se a coleta de lixo não estiver disponível, então os objetos remotos que não são mais necessários deverão ser excluídos.

Exceções ◊ Uma invocação remota pode falhar por motivos relacionados ao fato do objeto invocado estar em um processo ou computador diferente do invocador. Por exemplo, o processo que contém o objeto remoto pode ter falhado ou estar ocupado demais para responder, ou a mensagem de invocação ou de resultado ter se perdido. Portanto, a invocação a método remoto deve ser capaz de lançar exceções, como *timeouts*, causados pelo envio de mensagens, assim como aquelas ocorridas durante a execução do método invocado. Exemplos destas últimas são uma tentativa de ler além do final de um arquivo ou de acessar um arquivo sem as permissões corretas.

A IDL do CORBA fornece uma notação para especificar exceções em nível de aplicativo e o sistema subjacente gera exceções padrão quando ocorrem erros devido à distribuição de mensagens. Os programas clientes CORBA precisam ser capazes de tratar as exceções. Por exemplo, um programa cliente em C++ usará os mecanismos de exceção em C++.

5.2.4 Problemas de projeto para RMI

A seção anterior sugeriu que a RMI é uma extensão natural da invocação a método local. Nesta seção, discutiremos dois problemas de projeto que surgem ao se fazer essa extensão:

- A escolha da semântica de invocação – embora as invocações locais sejam executadas exatamente uma vez, nem sempre esse pode ser o caso para invocações a métodos remotos.
- O nível de transparência desejável para a RMI.

No restante da Seção 5.2, nos referimos aos processos que contém objetos remotos como servidores e aos processos que contém seus invocadores, como clientes. Os servidores também podem ser clientes.

Semântica de invocação RMI ◊ Os protocolos requisição-resposta foram discutidos na Seção 4.4, onde mostramos que a operação *doOperation* pode ser implementada de várias maneiras para fornecer diferentes garantias de entrega. As principais escolhas são:

Retentativa de mensagem de requisição: para retransmitir a mensagem de requisição até que uma resposta seja recebida ou que se presuma que o servidor falhou.

Filtragem de duplicatas: quando são usadas retransmissões ou para eliminar requisições replicadas no servidor.

Retransmissão de resultados: para manter um histórico das mensagens de respostas a fim de permitir que os resultados perdidos sejam retransmitidos sem uma nova execução das operações no servidor.

Combinações dessas escolhas levam a uma variedade de semânticas possíveis para a confiabilidade das invoções remotas vistas pelo ativador. A Figura 5.6 mostra as escolhas de interesse, com os nomes correspondentes da semântica da invocação que elas produzem. Note que, para invocações a métodos locais, a semântica é *exatamente uma vez*, significando que todo método é executado exatamente uma vez. As escolhas de semântica de invoção RMI são definidas como segue:

Semântica de invocação *talvez*: com a semântica de invocação *talvez*, o método remoto pode ser executado uma vez ou não ser executado. A semântica *talvez* surge quando nenhuma das medidas de tolerância a falhas é aplicada. Ela pode sofrer os seguintes tipos de falhas:

- falhas por omissão, se a mensagem de invocação ou de resultado for perdida;
- falhas por colapso, quando o servidor que contém o objeto remoto falha.

Se a mensagem de resultado não tiver sido recebida após um dado tempo limite (*timeout*) e não houver novas tentativas, não haverá certeza se o método foi executado. Se a mensagem de invocação foi perdida, então o método não terá sido executado. Por outro lado, o método pode ter sido executado e a mensagem de resultado, perdida. Uma falha por colapso pode ocorrer antes ou depois do método ser executado. Além disso, em um sistema assíncrono, o resultado da execução do método pode chegar após o *timeout*. A semântica *talvez* é útil apenas para aplicações nas quais são aceitáveis invocações mal-sucedidas ocasionais.

Semântica de invocação *pelo menos uma vez*: com a semântica de invocação *pelo menos uma vez*, o invocador recebe um resultado quando o método foi executado pelo menos uma vez, ou recebe uma exceção, informando-o que nenhum resultado foi obtido. A semântica de invocação *pelo menos uma vez* pode ser obtida pela retransmissão das mensagens de requisição, o que mascara as falhas por omissão da mensagem de invocação ou resultado. A semântica de invocação *pelo menos uma vez* pode sofrer dos seguintes tipos de falhas:

- falhas por colapso quando o servidor que contém o objeto remoto falha;
- falhas arbitrárias (bizantinas), nos casos em que a mensagem de invocação é retransmitida, o objeto remoto pode recebê-la e executar o método mais de uma vez, possivelmente causando o armazenamento ou o retorno de valores errados.

O Capítulo 4 definiu uma *operação idempotente* como aquela que pode ser executada repetidamente, com o mesmo efeito de que se tivesse sido executada exatamente uma vez. As operações não idempotentes podem ter um efeito colateral de provocar erros, caso sejam executadas mais de uma vez. Por exemplo, uma operação para aumentar um saldo bancário em $10 deve ser executada apenas uma vez; se ela fosse repetida, o saldo só aumentaria e aumentaria! Se os objetos em um servidor podem ser

Medidas de tolerância à falha			*Semântica de invocação*
Reenvio da mensagem de requisição	*Filtragem de duplicatas*	*Reexecução de procedimento ou retransmissão da resposta*	
Não	Não aplicável	Não aplicável	*Talvez*
Sim	Não	Executa o procedimento novamente	*Pelo menos uma vez*
Sim	Sim	Retransmite a resposta	*No máximo uma vez*

Figura 5.6 Semânticas de invocação.

projetados de modo que todos os métodos em suas interfaces remotas sejam operações idempotentes, então a semântica *pelo menos uma vez* pode ser aceitável.

Semântica de invocação *no máximo uma vez*: com a semântica de invocação *no máximo uma vez*, ou o ativador recebe um resultado quando o método foi executado exatamente uma vez, ou em caso contrário, uma exceção. A semântica de ativação *no máximo uma vez* pode ser obtida pelo uso de medidas de tolerância a falhas. Como no caso anterior, o emprego de retentativas mascara as falhas por omissão pela perda das mensagens de requisição ou de resultado. É necessário ainda que as falhas arbitrárias sejam evitadas, garantindo que, para cada RMI, um método nunca seja executado mais de uma vez. Na RMI Java e no CORBA, a semântica de ativação é *no máximo uma vez*, mas o CORBA permite que a semântica *talvez* seja usada para métodos que não retornam resultados. A RPC Sun fornece semântica de chamada *pelo menos uma vez*.

Transparência ◊ Os criadores da RPC, Birrell e Nelson [1984], pretendiam tornar as chamadas de procedimentos remotos o mais parecido possível com as chamadas de procedimentos locais, sem nenhuma distinção na sintaxe entre uma chamada de procedimento local e um remoto. Todas as etapas necessárias para empacotamento e trocas de mensagens são ocultadas do programador que faz uma chamada de procedimento remoto. Embora as mensagens de requisição sejam retransmitidas após um *timeout*, isso é transparente para o chamador – para tornar a semântica das chamadas de procedimentos remotos igual à das chamadas de procedimentos locais. Essa noção de transparência foi ampliada para ser aplicada em objetos distribuídos, mas envolve a ocultação não apenas do empacotamento e da troca de mensagens, mas também da tarefa de localizar e contatar um objeto remoto. Como exemplo, a RMI Java torna as invocações a métodos remotos muito parecidas com as locais, permitindo que elas utilizem a mesma sintaxe.

Entretanto, as invocações remotas são mais vulneráveis à falhas do que as locais, pois envolvem a rede, outro computador e outro processo. Qualquer que seja a semântica de invocação escolhida, há sempre a chance de que nenhum resultado seja recebido e, no caso de falhas, é impossível distinguir entre falha da rede e do processo servidor remoto. Isso exige que os objetos que estão fazendo invocações remotas possam se recuperar de tais situações.

A latência de uma invocação remota é muito maior do que a de uma local. Isso sugere que os programas que utilizam invocações remotas precisam levar esse fator em consideração, talvez minimizando as interações remotas. Os projetistas do Argus [Liskov e Scheifler 1982] sugeriram que deveria ser possível para um chamador cancelar uma chamada de procedimento remoto que está demorando demais, de tal maneira que isso não tenha efeito sobre o servidor. Para possibilitar isso, o servidor precisaria restaurar as variáveis para o modo como estavam antes que o procedimento fosse chamado. Esses problemas serão discutidos no Capítulo 13.

Waldo *et al.* [1994] dizem que a diferença entre objetos locais e remotos deve ser expressa na interface remota, para permitir que os objetos reajam de maneira consistente às possíveis falhas parciais. Outros sistemas foram ainda mais longe do que isso, argumentando que a sintaxe de uma chamada remota deve ser diferente da de uma chamada local: no caso do Argus, a linguagem foi ampliada para tornar as operações remotas explícitas para o programador.

A escolha quanto ao fato das invocações remotas deverem ser transparentes também está disponível para os projetistas de IDLs. Por exemplo, no CORBA, uma invocação remota lança uma exceção quando o cliente não é capaz de se comunicar com um objeto remoto. Isso exige que o programa cliente trate de tais exceções, permitindo que ele lide com essas falhas. Uma IDL também pode fornecer um recurso para especificar a semântica de chamada de um método. Isso pode ajudar o projetista do objeto remoto – por exemplo, se a semântica *pelo menos uma vez* for escolhida para evitar as sobrecargas da semântica *no máximo uma vez*, as operações do objeto serão projetadas de modo a serem idempotentes.

O consenso atual parece ser o de que as invocações remotas devem se tornar transparentes, no sentido de que a sintaxe de uma invocação remota é a mesma de uma invocação local, mas que a diferença entre objetos locais e remotos deve ser expressa em suas interfaces. No caso da RMI Java, os objetos remotos podem ser distinguidos pelo fato de implementarem a interface *Remote* e dispararem exceções *RemoteException*. Os implementadores de um objeto remoto cuja interface é especificada em uma IDL também sabem da diferença. O conhecimento de que um objeto se destina a ser acessado por invocação remota tem outra implicação para seu projetista: ele deve ser capaz de manter seu estado consistente na presença de acessos concorrentes feitos por vários clientes.

5.2.5 Implementação de RMI

Vários objetos e módulos separados estão envolvidos na realização de uma invocação a método remoto. Eles aparecem na Figura 5.7, na qual um objeto A invoca um método em um objeto remoto B, para o qual mantém uma referência de objeto remoto. Esta seção discute as funções de cada um dos componentes mostrados nessa figura, tratando primeiro da comunicação e dos módulos de referência remota e depois do software RMI neles executado.

O restante desta seção trata dos seguintes tópicos relacionados: a geração de *proxies*, a associação de nomes às suas referências de objeto remoto, a invocação e a passividade de objetos e a localização de objetos a partir de suas referências de objeto remoto.

Módulo de comunicação ◊ Dois módulos de comunicação cooperam para executar o protocolo requisição-resposta, o qual transmite mensagens de *requisição* e *resposta* entre o cliente e o servidor. O conteúdo das mensagens de *requisição* e *resposta* aparece na Figura 4.16. O módulo de comunicação usa apenas os três primeiros elementos, os quais especificam o tipo de mensagem, sua *requestId* e a referência remota do objeto a ser invocado. O identificador de método (*methodId*), e todo o empacotamento e desempacotamento, é de responsabilidade do software RMI como discutido a seguir. Juntos, os módulos de comunicação são responsáveis por fornecer uma semântica de invocação específica, por exemplo, *no máximo uma vez*.

O módulo de comunicação no servidor seleciona o despachante para a classe do objeto a ser invocado, passando sua referência local, obtida a partir do módulo de referência remota através do identificador do objeto remoto dado pela mensagem de requisição. A função do despachante será discutida no software RMI, a seguir.

Módulo de referência remota ◊ O módulo de referência remota é responsável pela transformação entre referências de objeto local e remoto e pela criação de referências de objeto remoto. Para executar sua funcionalidade, o módulo de referência remota de cada processo contém uma *tabela de objetos remotos* para registrar a correspondência entre as referências de objeto local desse processo e as referências de objeto remoto (que abrangem todo o sistema). A tabela inclui:

- Uma entrada para todos os objetos remotos mantidos pelo processo. Por exemplo, na Figura 5.7, o objeto remoto B será registrado na tabela do servidor.
- Uma entrada para cada *proxy* local. Por exemplo, na Figura 5.7, o *proxy* de B será registrado na tabela do cliente.

A função de um *proxy* será discutida mais adiante. As ações do módulo de referência remota são as seguintes:

- Quando um objeto remoto precisa ser passado como argumento ou resultado pela primeira vez, o módulo de referência remota cria uma referência de objeto remoto, a qual adiciona em sua tabela.

Figura 5.7 A função do *proxy* e do esqueleto na invocação a método remoto.

- Quando uma referência de objeto remoto chega em uma mensagem de requisição ou resposta, é solicitada ao módulo de referência remota a correspondente referência de objeto local, a qual pode se referir a um *proxy* ou a um objeto remoto. No caso da referência de objeto remoto não estar na tabela, o software RMI cria um novo *proxy* e pede ao módulo de referência remota para adicioná-lo na tabela.

Esse módulo é chamado pelos componentes do software RMI quando estão empacotando e desempacotando referências de objeto remoto. Por exemplo, quando chega uma mensagem de requisição, a tabela é usada para descobrir qual objeto local deve ser invocado.

Serventes ◊ Um servente é uma instância de uma classe que fornece o corpo de um objeto remoto. É o servente que trata as requisições remotas repassadas pelo esqueleto correspondente. Os serventes são partes integrantes do processo servidor. Eles são criados quando os objetos remotos são instanciados e permanecem em uso até não serem mais necessários, sendo finalmente excluídos.

O software RMI ◊ É uma camada de software – *middleware* – entre os objetos do aplicativo e os módulos de comunicação e de referência remota. As funções dos seus componentes, mostrados na Figura 5.7, são as seguintes:

Proxy: a função de um *proxy* é tornar a invocação a método remoto transparente para os clientes, comportando-se como um objeto local para o invocador; mas, em vez de executar uma invocação local, ele a encaminha em uma mensagem para um objeto remoto. Ele oculta os detalhes da referência de objeto remoto, do empacotamento de argumentos, do desempacotamento dos resultados, e do envio e recepção de mensagens do cliente. Existe um *proxy* para cada objeto remoto que um processo faz uma referência de objeto remoto. A classe de um *proxy* implementa os métodos da interface remota do objeto remoto que ele representa. Isso garante que as invocações a métodos remotos sejam apropriadas para o objeto remoto em questão. Entretanto, o *proxy* implementa os métodos de uma forma diferente. Cada método do *proxy* empacota uma referência para o objeto alvo, o *methodId* e seus argumentos em uma mensagem de *requisição* e a envia para o objeto alvo. A seguir espera pela mensagem de *resposta*, quando a recebe, desempacota e retorna os resultados para o invocador.

Despachante: um servidor tem um despachante e um esqueleto para cada classe que representa um objeto remoto. Em nosso exemplo, o servidor tem um despachante e um esqueleto para a classe do objeto remoto B. O despachante recebe a mensagem de *requisição* do módulo de comunicação e utiliza o *methodId* para selecionar o método apropriado no esqueleto, repassando a mensagem de *requisição*. O despachante e o *proxy* usam o mesmo *methodId* para os métodos da interface remota.

Esqueleto: a classe de um objeto remoto tem um *esqueleto*, o qual implementa os métodos da interface remota, mas de uma forma diferente dos métodos implementados no servente que personifica o objeto remoto. Um método de esqueleto desempacota os argumentos na mensagem de *requisição* e invoca o método correspondente no servente. Ele espera que a invocação termine e, em seguida, empacota o resultado, junto com as exceções, em uma mensagem de *resposta* que é enviada para o método do *proxy* que fez a requisição.

As referências de objeto remoto são empacotadas na forma mostrada na Figura 4.13, que inclui informações sobre a interface remota do objeto remoto; por exemplo, o nome da interface remota ou da classe do objeto remoto. Essas informações permitem que a classe do *proxy* seja determinada para, quando necessário, que um novo *proxy* possa ser criado. Por exemplo, o nome da classe do *proxy* pode ser gerado anexando "*proxy*" ao nome da interface remota.

Geração das classes para *proxies*, despachantes e esqueletos ◊ As classes para o *proxy*, despachante e esqueleto usados na RMI são geradas automaticamente por um compilador de interface. Por exemplo, na implementação Orbix do CORBA, as interfaces dos objetos remotos são definidas na IDL do CORBA e o compilador de interface pode ser usado para gerar as classes para *proxies*, despachantes e esqueletos em C++ ou em Java [www.iona.com]. Para a RMI Java, o conjunto de métodos oferecidos por um objeto remoto é definido como uma interface Java implementada dentro da classe do objeto remoto. O compilador da RMI Java gera as classes de *proxy*, despachante e esqueleto a partir da classe do objeto remoto.

Ativação dinâmica: uma alternativa aos *proxies* ◊ O *proxy* que acabamos de descrever é estático, no sentido de que sua classe é gerada a partir de uma definição de interface e depois compilada no código do cliente. Às vezes, isso não é prático: suponha que um programa cliente receba uma referência remota para um objeto cuja interface remota não estava disponível no momento da compilação. Nesse caso, ele precisa de outra maneira para invocar o objeto remoto: isso é chamado de *invocação dinâmica*. Ele dá ao cliente acesso a uma representação genérica de uma invocação remota, como o método *DoOperation* usado no Exercício 5.8, que está disponível como parte da infra-estrutura de RMI (veja a Seção 4.4). O cliente fornecerá a referência de objeto remoto, o nome do método e os argumentos para *DoOperation* e depois esperará para receber os resultados.

Note que, embora a referência de objeto remoto inclua informações sobre a interface do objeto remoto, como seu nome, isso não é suficiente – os nomes dos métodos e os tipos dos argumentos são necessários para se fazer uma invocação dinâmica. O CORBA fornece suas informações por meio de um componente chamado *Interface Repository* (repositório de interfaces), que será descrito na Seção 20.2.2.

A interface de invocação dinâmica não é tão conveniente para usar como *proxy*, mas é útil em aplicações onde nem todas as interfaces dos objetos remotos são previstas no momento do projeto. Um exemplo de tal aplicação é o quadro branco compartilhado usado para ilustrar a RMI Java (Seção 5.5), o CORBA (Seção 20.2) e os serviços web (Seção 19.2.3). Resumindo: o aplicativo de quadro branco compartilhado exibe muitos tipos diferentes de figuras, como círculos, retângulos e linhas, mas precisa mostrar novas figuras que não foram previstas quando o cliente foi compilado. Um cliente que utilize invocação dinâmica é capaz de resolver esse problema. Na Seção 5.5, veremos que o *download* dinâmico de classes em clientes é uma alternativa à invocação dinâmica. Isso está disponível na RMI Java.

Esqueletos dinâmicos: a partir do exemplo anterior, fica claro que um servidor também precisará conter objetos remotos cujas interfaces não eram conhecidas no momento da compilação. Por exemplo, um cliente pode fornecer um novo tipo de figura para o servidor de quadro branco compartilhado armazenar. Um servidor com esqueletos dinâmicos resolveria essa situação. Vamos deixar para descrever os esqueletos dinâmicos no capítulo sobre CORBA, na Seção 20.2.2. Entretanto, conforme veremos na Seção 5.2, a RMI Java resolve esse problema usando um despachante genérico e o *download* dinâmico de classes no servidor.

Programas clientes e servidores ◊ O programa servidor contém as classes para os despachantes e esqueletos, junto com as implementações das classes de todos os serventes que suporta. Além disso, o programa servidor contém uma seção de *inicialização* (por exemplo, em um método *main* em Java ou C++). A seção de *inicialização* é responsável por criar e inicializar pelo menos um dos serventes que fazem parte do servidor. Mais serventes podem ser criados em resposta às requisições dos clientes. A seção de *inicialização* também pode registrar alguns de seus serventes com um vinculador (*binder*) (veja o próximo parágrafo), porém, geralmente, é registrado apenas um servente, o qual pode ser usado para acessar os restantes.

O programa cliente conterá as classes dos *proxies* de todos os objetos remotos que ativará. Ele pode usar um vinculador para pesquisar referências de objeto remoto.

Métodos de fábrica: mencionamos anteriormente que as interfaces de objeto remoto não incluem construtores. Isso significa que os serventes não podem ser criados por invocação remota em construtores. Os serventes são criados pela *inicialização* ou por métodos de uma interface remota destinados a esse propósito. O termo *método de fábrica* às vezes é usado para se referir a um método que cria serventes, e um *objeto de fábrica* é um objeto com métodos de fábrica. Todo objeto remoto que precise criar novos objetos remotos a pedido dos clientes deve fornecer métodos em sua interface remota para esse propósito. Tais métodos são chamados de métodos de fábrica, embora na verdade sejam apenas métodos normais.

O vinculador (*binder*) ◊ Os programas clientes geralmente exigem uma maneira de obter uma referência de objeto remoto para pelo menos um dos objetos remotos mantidos por um servidor. Por exemplo, na Figura 5.3, o objeto A exige uma referência de objeto remoto para o objeto B. Em um sistema distribuído, um *vinculador* é um serviço separado que mantém uma tabela contendo mapeamentos dos nomes textuais para referências de objeto remoto. Ele é usado pelos servidores para regis-

trar seus objetos remotos pelo nome e pelos clientes, para pesquisá-los. O Capítulo 20 contém uma discussão sobre o serviço de atribuição de nomes (*Naming Service*) do CORBA. O vinculador Java, RMIregistry, será discutido brevemente no estudo de caso sobre a RMI Java, na Seção 5.5.

***Threads* no servidor** ◊ Quando um objeto executa uma invocação remota pode acarretar execução em mais invocações a métodos em outros objetos remotos, os quais podem levar algum tempo para retornar. Para evitar que a execução de uma invocação remota atrase a execução de outra, geralmente os servidores criam uma *thread* para cada invocação remota. Quando isso acontece, o projetista da implementação de um objeto remoto deve levar em conta os efeitos das execuções concorrentes sobre seu estado.

Invocação de objetos remotos ◊ Algumas aplicações exigem que as informações sejam válidas por longos períodos de tempo. Entretanto, não é prático que os objetos que representam tais informações sejam mantidos por períodos ilimitados em processos que estejam em execução, particularmente porque eles não estão necessariamente em uso o tempo todo. Para evitar o potencial desperdício de recursos, devido à execução de todos os servidores que gerenciam objetos remotos, os mesmos podem ser iniciados somente quando forem necessários para os clientes. Isso é análogo ao que acontece no conjunto padrão de serviços TCP, como o FTP, que são iniciados de acordo com a demanda, por um serviço chamado *Inetd*. Os processos que iniciam processos servidores para conter objetos remotos são chamados de *ativadores* pelos motivos a seguir.

Um objeto remoto é dito *ativo* quando está disponível para invocação dentro de um processo em execução, e é chamado de *passivo* se não estiver ativo no momento da invocação, mas puder se tornar. Um objeto passivo consiste em duas partes:

1. na implementação de seus métodos; e
2. em seu estado na forma empacotada.

A *ativação* consiste na criação de um objeto ativo a partir do objeto passivo correspondente, pela criação de uma nova instância de sua classe e pela inicialização de suas variáveis de instância a partir do estado armazenado. Os objetos passivos podem ser ativados por demanda; por exemplo, quando eles precisam ser invocados por outros objetos.

Um *ativador* é responsável por:

- Registrar os objetos passivos que estão disponíveis para ativação, o que envolve registrar os nomes dos servidores com os URLs ou nomes de arquivo dos objetos passivos correspondentes.
- Iniciar processos servidores, identificá-los e ativar objetos remotos neles.
- Controlar a localização dos servidores de objetos remotos que já tenha ativado.

A RMI Java fornece a capacidade de tornar objetos remotos *passíveis de ativação* [java.sun.com IX]. Quando um objeto passível de ativação é invocado, se já não estiver correntemente ativo, ele se tornará ativo a partir de seu estado empacotado e depois será invocado. A RMI Java emprega um ativador para cada computador servidor.

O estudo de caso do CORBA descreve o repositório de implementação – uma forma simplificada de ativador para disparar serviços contendo objetos em um estado inicial.

Repositório de objetos persistentes ◊ Um objeto que mantém seu estado entre invocações é chamado de *objeto persistente*. Geralmente, os objetos persistentes são gerenciados por repositórios de objetos persistentes, que guardam seus estados em uma forma empacotada no disco. Exemplos incluem o serviço de estado persistente do CORBA (veja a Seção 20.3), *Java Data Objects* [java.sun.com VIII] e *Persistent Java* [Jordan 1996, java.sun.com IV].

Em geral, um repositório de objetos persistentes gerencia uma grande quantidade de objetos, os quais são armazenados em disco, ou em um banco de dados, até serem necessários. Eles serão ativados quando seus métodos forem invocados por outros objetos. A ativação geralmente é projetada para ser transparente – isto é, o invocador não deve saber se um objeto já está na memória principal ou se precisa ser ativado antes de seu método ser invocado. Os objetos persistentes que não são mais necessários na memória principal podem ser postos em estado passivo. Na maioria dos casos, os objetos são salvos no repositório de objetos persistentes sempre que atingirem um estado consistente, isso para

fornecer um grau de tolerância a falhas. O repositório de objetos persistentes precisa de uma estratégia para decidir quando deve tornar passivo os objetos. Por exemplo, ele pode fazer isso em resposta a um pedido feito pelo programa que ativou os objetos; por exemplo, no final de uma transação ou quando o programa termina. Geralmente, os repositórios de objetos persistentes tentam otimizar esse procedimento, salvando apenas os objetos que foram modificados desde a última vez que foram salvos.

De modo geral, os repositórios de objetos persistentes permitem que conjuntos de objetos persistentes relacionados tenham nomes legíveis por seres humanos, como nomes de caminho ou URLs. Na prática, cada um desses nomes é associado à raiz de um conjunto de objetos persistentes.

Existem duas abordagens para decidir se um objeto é persistente ou não:

- O repositório de objetos persistentes mantém raízes persistentes e todo objeto atingido a partir de uma raiz persistente é definido como persistente. Esta estratégia é usada por *Persistent Java*, *Java Data Objects* e por *PerDiS* [Ferreira *et al.* 2000]. Elas utilizam um coletor de lixo para descartar os objetos que não podem mais ser atingidos a partir das raízes persistentes.
- O repositório de objetos persistentes fornece algumas classes bases para a obtenção de persistência – os objetos persistentes pertencem às suas subclasses. Por exemplo, em Arjuna [Parrington *et al.* 1995], os objetos persistentes são baseados nas classes C++ que fornecem transações e recuperação. Os objetos que não são mais necessários devem ser explicitamente excluídos.

Alguns repositórios de objetos persistentes, por exemplo PerDiS e Khazana [Carter *et al.* 1998] permitem que os objetos sejam ativados em caches locais aos usuários, em vez de serem ativados nos servidores. Nesse caso, é exigido um protocolo de consistência de cache; o Capítulo 18 discutirá uma variedade de modelos de consistência.

Localização de objetos ◊ A Seção 4.3.4 descreveu uma forma de referência de objeto remoto que continha o endereço IP e o número de porta do processo que criou o objeto remoto como uma maneira de garantir a exclusividade. Essa forma de referência de objeto remoto também pode ser usada como endereço de um objeto remoto, desde que ele permaneça no mesmo processo enquanto existir. Mas, eventualmente, alguns objetos remotos poderão fazer parte de processos diferentes, possivelmente em distintos computadores. Nesse caso, uma referência de objeto remoto não pode atuar como endereço. Os clientes que fazem invocações precisam tanto de uma referência de objeto remoto como de endereço para enviar as invocações.

Um *serviço de localização* ajuda os clientes a encontrarem objetos remotos a partir de suas referências de objeto remoto. Ele utiliza um banco de dados que faz o mapeamento das referências de objeto remoto para suas prováveis localizações correntes – as localizações são prováveis porque um objeto pode ter migrado novamente, desde a última vez que foi encontrado. Por exemplo, o sistema Clouds [Dasgupta *et al.* 1991] e o sistema Emerald [Jul *et al.* 1988] usavam um esquema de cache/*broadcast* no qual um membro de um serviço de localização em cada computador continha uma pequena cache de mapeamentos de referência de objeto remoto para localização. Se uma referência de objeto remoto estivesse na cache, a invocação era feita nesse endereço e falhava se o objeto tivesse mudado de lugar. Para localizar um objeto que tinha mudado de lugar, ou cuja localização não fosse a cache, o sistema fazia uma requisição em *broadcast*. Esse esquema pode ser aprimorado pelo uso de ponteiros de previsão de localização, os quais contêm sugestões sobre a nova localização de um objeto. Outro exemplo é o serviço de resolução de nomes, mencionado na Seção 9.1, usado para transformar o URN de um recurso em seu URL corrente.

5.2.6 Coleta de lixo distribuída

O objetivo de um coletor de lixo distribuído é garantir que, se uma referência local ou remota para um objeto ainda for mantida em algum lugar, em um conjunto de objetos distribuídos, então o próprio objeto continuará a existir; mas assim que não houver mais nenhuma referência para ele, esse objeto será coletado e a memória por ele utilizada será recuperada.

Descrevemos o algoritmo de coleta de lixo distribuída Java, que é semelhante ao descrito por Birrell *et al.* [1995]. Ele é baseado na contagem de referência. Quando uma referência de objeto remoto

for feita em um processo, um *proxy* será criado e existirá enquanto for necessário. O processo onde o objeto reside (seu servidor) deve ser informado desse *proxy* cliente. Então, posteriormente, quando o *proxy* deixar de existir no cliente, o servidor deverá ser novamente informado. O coletor de lixo distribuído trabalha em cooperação com os coletores de lixo locais, como segue:

- Cada processo servidor mantém um conjunto de nomes dos processos que contêm referências de objeto remoto para cada um de seus objetos remotos; por exemplo, *B.holders* é o conjunto de processos clientes (máquinas virtuais) que têm *proxies* para o objeto B. (Na Figura 5.7, esse conjunto incluirá o processo cliente ilustrado.) Esse conjunto pode ser mantido em uma coluna adicional na tabela de objetos remotos.
- Quando um cliente *C* recebe pela primeira vez uma referência remota para um objeto remoto particular *B*, ele faz uma invocação à *addRef(B)* no servidor desse objeto remoto e depois cria um *proxy*; o servidor adiciona *C* a *B.holders*.
- Quando o coletor de lixo de um cliente *C* percebe que um *proxy* para o objeto remoto *B* não é mais necessário, ele faz uma invocação à *removeRef(B)* no servidor correspondente e depois exclui o *proxy*; o servidor remove *C* de *B.holders*.
- Quando *B.holders* estiver vazio, o coletor de lixo local do servidor recuperará o espaço ocupado por *B*, a não ser que existam clientes (*holders*) locais.

Esse algoritmo é feito para ser executado por meio de comunicação do tipo requisição-resposta, com uma semântica de invocação *no máximo uma vez* entre os módulos de referência remota presentes nos processos cliente e servidor – ele não exige nenhum sincronismo global. Note também que invocações normais à coleta de lixo não afetam cada RMI; elas ocorrem quando os *proxie*s são criados e excluídos.

Existe a possibilidade de que um cliente possa fazer uma invocação *removeRef(B)* praticamente ao mesmo tempo em que outro cliente faz uma invocação *addRef(B)*. Se a invocação *removeRef* chegar primeiro e *B.holders* estiver vazio, o objeto remoto *B* poderá ser excluído antes da chegada da invocação *addRef*. Para evitar essa situação, se o conjunto *B.holders* estiver vazio no momento em que uma referência de objeto remoto for transmitida, uma entrada temporária é adicionada até a chegada de *addRef*.

O algoritmo de coleta de lixo distribuída Java tolera falhas de comunicação usando a seguinte estratégia. As operações *addRef* e *removeRef* são idempotentes. No caso de uma chamada *addRef(B)* retornar uma exceção (significando que o método foi executado uma vez ou não foi executado), o cliente não criará o *proxy*, mas fará uma chamada *removeRef(B)*. O efeito de *removeRef* estará correto, tenha *addRef* obtido êxito ou não. O caso em que *removeRef* falha é tratado por *leasing*, conforme descrito a seguir.

O algoritmo de coleta de lixo distribuída Java pode tolerar a falha de processos clientes. Para conseguir isso, os servidores *cedem* seus objetos para os clientes por um tempo limitado. O período de arrendamento (*leasing*) começa quando o cliente faz uma invocação a *addRef* no servidor, e termina quando o tempo tiver expirado ou quando o cliente fizer uma invocação a *removeRef* no servidor. As informações armazenadas pelo servidor, relativas a cada *leasing*, contêm o identificador da máquina virtual do cliente e o período de *leasing*. Os clientes são responsáveis por pedir ao servidor para que renove seus *leasings* antes que expirem.

Arrendamento (*leasing*) no Jini ◊ O sistema distribuído Jini inclui uma especificação para *arrendamento* [Arnold *et al.* 1999] que pode ser usada nas situações em que um objeto oferece um recurso para outro objeto como, por exemplo, quando objetos remotos oferecem referências para outros objetos. Os objetos que oferecem tais recursos correm o risco de terem que mantê-los mesmo quando os usuários não estiverem mais interessados ou quando seus programas tiverem terminado. Para evitar a necessidade de protocolos complicados para descobrir se os usuários ainda estão interessados no recurso, os mesmos são oferecidos por tempo limitado. A concessão do uso de um recurso por um período de tempo é chamada de *arrendamento* (*leasing*). O objeto que está oferecendo o recurso o manterá até o momento em que o arrendamento expirar. Os usuários do recurso são responsáveis, quando necessário, por solicitar a renovação do período de arrendamento.

O período de um arrendamento pode ser negociado entre o cedente e o receptor, embora isso não aconteça com os arrendamentos usados na RMI Java. Um objeto que representa um arrendamento implementa a interface *Lease*. Ela contém informações sobre o período de arrendamento e os métodos que permitem sua renovação ou seu cancelamento. O cedente retorna uma instância de *Lease*, quando fornece um recurso para outro objeto.

5.3 Chamada de procedimento remoto

Uma chamada de procedimento remoto é muito parecida com uma invocação a método remoto, pois um processo cliente chama um procedimento que está sendo executado em um processo servidor. Os servidores podem ser clientes de outros servidores para permitir encadeamentos de RPCs. Conforme mencionado na introdução deste capítulo, um processo servidor define em sua *interface de serviço* os procedimentos que estão disponíveis para serem chamados de forma remota. Na verdade, esse tipo de serviço é muito parecido com um objeto remoto simples, pois ele tem estado e métodos. Entretanto, ele não tem a capacidade de criar novas instâncias de objetos e, portanto, não suporta referências de objeto remoto.

A RPC, assim como a RMI, pode ser implementada de forma a ter uma das opções de semântica de invocação discutidas na Seção 5.2.4 – geralmente são escolhidas as semânticas *pelo menos uma vez* ou *no máximo uma vez*. De modo geral, a RPC é implementada sobre um protocolo requisição-resposta como aquele discutido na Seção 4.4, que é simplificado pela ausência das referências de objeto nas mensagens de requisição. O conteúdo das mensagens de requisição e resposta é igual àquele ilustrado para a RMI, na Figura 4.16, exceto que o campo *ObjectReference* foi omitido.

O software que suporta RPC aparece na Figura 5.8 da forma como seria para uma linguagem procedural, como C, que não suporta classes nem objetos. Ele é relevante para nosso estudo de caso sobre a RPC Sun, que utiliza a linguagem C. Ele também é interessante por motivos históricos, pois a maior parte dos primeiros sistemas RPC era baseada em C.

Esse software é semelhante ao que aparece na Figura 5.7, exceto que nenhum módulo de referência remota é exigido, pois a chamada de procedimento não se preocupa com objetos e referências de objeto. O cliente que acessa um serviço inclui um *procedimento stub* para cada procedimento da interface de serviço. A função de um procedimento *stub* é semelhante à de um método *proxy*. Ele se comporta como um procedimento local para o cliente, mas em vez de executar a chamada, ela empacota o identificador de procedimento e os argumentos em uma mensagem de requisição e a envia para o servidor por meio de seu módulo de comunicação. Quando a mensagem de resposta chega, ela desempacota os resultados. O processo servidor contém um despachante junto com um procedimento *stub* de servidor e um procedimento de serviço, para cada procedimento na interface de serviço. O despachante seleciona um dos procedimentos *stub* de servidor, de acordo com o identificador de

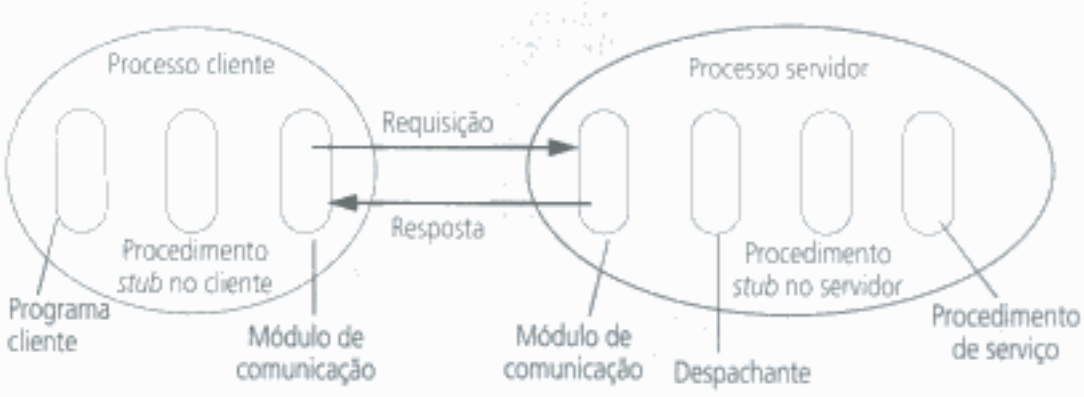


Figura 5.8 Função dos procedimentos *stub* no cliente e no servidor na RPC, no contexto de uma linguagem procedural.

procedimento presente na mensagem de requisição. Um *procedimento stub de servidor* é como um método esqueleto, pois ele desempacota os argumentos presentes na mensagem de requisição, chama o procedimento de serviço correspondente e empacota os valores de retorno para a mensagem de resposta. Os procedimentos de serviço implementam os procedimentos da interface de serviço.

Os procedimentos *stub* do cliente e do servidor e o despachante podem ser gerados por um compilador de interface, a partir da definição de interface do serviço. Se o cliente ou o servidor usar uma linguagem baseada em objetos, como C++ ou Java, o conjunto de procedimentos *stub* do cliente de um serviço poderá ser implementado como um *proxy*, e o conjunto de procedimento *stub* do servidor, como um esqueleto.

5.3.1 Estudo de caso: RPC da Sun

O RFC 1831 [Srinivasan 1995a] descreve a RPC Sun, que foi projetada para comunicação cliente-servidor no sistema de arquivos de rede NFS (*Network File System*). A RPC Sun às vezes é chamada de RPC ONC (*Open Network Computing*). O suporte a RPC é fornecido como parte integrante dos vários sistemas operacionais da Sun e outros sistemas operacionais derivados do UNIX. Os implementadores têm a opção de usar chamadas de procedimentos remotos sobre UDP ou TCP. Quando a RPC Sun é usada com UDP, os comprimentos das mensagens de requisição e de resposta são limitados, teoricamente, em 64 Kbytes, mas, na prática, mais freqüentemente, em 8 ou 9 Kbytes. A semântica usada é a *pelo menos uma vez*. Existe a opção de usar *broadcast* de RPC.

O sistema RPC Sun fornece uma linguagem de interface chamada XDR e um compilador de interface chamado *rpcgen*, destinado a ser usado com a linguagem de programação C.

Linguagem de definição de interface ◊ A linguagem XDR da Sun, originalmente projetada para especificar representações externas de dados, foi estendida para se tornar uma linguagem de definição de interface. Ela pode ser usada para definir uma interface de serviço para RPC, por meio da especificação de um conjunto de definições de procedimentos, junto com definições de tipo suportados. A notação é bastante primitiva, em comparação com aquela usada pela IDL do CORBA ou Java. Em particular:

- A maioria das linguagens permite a especificação de nomes de interface, mas a RPC Sun não permite – em vez disso, são fornecidos um número de programa e um número de versão. Os números de programa podem ser obtidos a partir de uma autoridade central, para permitir que cada programa tenha seu próprio número exclusivo. O número de versão muda quando uma assinatura de procedimento é alterada. Tanto o número de programa como o número de versão são passados na mensagem de requisição para que o cliente e o servidor possam verificar se estão usando a mesma versão.
- A definição de um procedimento especifica uma assinatura e um número que é usado como identificador de procedimento nas mensagens de requisição. (Seria possível o compilador de interface gerar identificadores de procedimentos).
- É permitido apenas um parâmetro de entrada. Portanto, os procedimentos que necessitam vários parâmetros devem incluí-los como membros de uma única estrutura.
- Os parâmetros de saída de um procedimento são retornados por meio de um único resultado.
- A assinatura de um procedimento consiste no tipo do resultado, no nome do procedimento e no tipo do parâmetro de entrada. O tipo do resultado e do parâmetro de entrada pode especificar um único valor ou uma estrutura contendo vários valores.

Por exemplo, veja a definição de XDR na Figura 5.9, de uma interface com dois procedimentos para gravar e ler arquivos. O número do programa é 9999 e o número da versão é 2. O procedimento *READ* (linha 2) recebe como parâmetro de entrada uma estrutura com três componentes, especificando um identificador de arquivo, uma posição no arquivo e o número de bytes a serem lidos. Seu resultado é uma estrutura contendo o número de bytes retornados e os dados do arquivo. O procedimento *WRITE* (linha 1) não tem nenhum resultado. Os procedimentos *WRITE* e *READ* recebem os números 1 e 2. O número zero é reservado para um procedimento nulo, que é gerado automaticamente e se destina a testar se um servidor está disponível.

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};
struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
    version VERSION {
        void WRITE(writeargs)=1;                    1
        Data READ(readargs)=2;                      2
    }=2;
} = 9999
```

Figura 5.9 Exemplo de interfaces de procedimentos na XDR da Sun.

A linguagem de definição de interface fornece uma notação para definir constantes, *typedefs*, estruturas, tipos enumerados, uniões e programas propriamente ditos. Para a definição de *typedefs*, estruturas e tipos enumerados é empregado a sintaxe da linguagem C. A partir de uma definição de interface o compilador de interface *rpcgen* gera o seguinte:

- procedimentos *stub* no cliente;
- procedimento *main*, despachante e procedimentos *stub* no servidor;
- procedimentos de empacotamento e desempacotamento de XDR a serem usados pelo despachante e pelos procedimentos *stub* do cliente e do servidor.

Vinculação (*binding*) ◊ A RPC Sun executa em cada computador um serviço de vinculação local chamado *mapeador de porta* (*port mapper*) em um número de porta bem conhecido. Cada instância de um mapeador de porta grava o número do programa, o número da versão e o número da porta para cada um dos serviços em execução. Quando um servidor é iniciado, ele registra seu número de programa, número de versão e número de porta no seu mapeador de porta. Quando um cliente é disparado, ele descobre a porta do serviço fazendo um pedido remoto para o mapeador de porta no computador onde o servidor executa, especificando o número de programa e o número de versão.

Quando um serviço tem várias instâncias sendo executadas em diferentes computadores, as instâncias podem usar diferentes números de porta para receber as requisições dos clientes. Se um cliente precisa enviar uma requisição para todas as instâncias de um serviço que usam diferentes números de porta, ele pode fazê-lo enviando uma mensagem *multicast* IP para todos os mapeadores de porta, especificando o número do programa e da versão. Cada mapeador de porta encaminha tais chamadas para o programa de serviço local apropriado, se houver um.

Autenticação ◊ As mensagens de requisição e resposta da RPC Sun fornecem campos adicionais que permitem autenticar as informações a serem passadas entre cliente e servidor. A mensagem de

requisição contém as credenciais do usuário que está executando o programa cliente. Por exemplo, na autenticação UNIX, as credenciais incluem o *uid* e o *gid* do usuário. Mecanismos de controle de acesso podem ser construídos sobre informações de autenticação repassadas aos procedimentos do servidor por intermédio de um segundo argumento. O programa servidor é responsável por impor o controle de acesso, decidindo se vai ou não executar cada chamada de procedimento de acordo com as informações de autenticação. Por exemplo, ao se tratar de um servidor de arquivos NFS, ele pode verificar se o usuário tem direitos suficientes para executar a operação solicitada em um determinado arquivo.

Vários protocolos de autenticação diferentes podem ser suportados. Isso inclui:

- nenhum;
- estilo UNIX, como descrito anteriormente;
- baseado em uma chave compartilhada para assinar as mensagens RPC; ou
- autenticação Kerberos (veja o Capítulo 7).

Um campo no cabeçalho RPC indica qual forma está sendo usada.

Uma estratégia de segurança mais genérica está descrita no RFC 2203 [Eisler *et al.* 1997]. Ela proporciona sigilo e integridade das mensagens RPC, assim como autenticação. A estratégia permite que cliente e servidor negociem um contexto de segurança onde escolhem não usar segurança alguma, garantir integridade das mensagens, garantir privacidade das mensagens, ou ambas.

Programas clientes e servidores ◊ Mais material sobre a RPC Sun está disponível no endereço [www.cdk4.net/rmi]. Ele inclui exemplos de programas clientes e servidores correspondentes à interface definida na Figura 5.9.

5.4 Eventos e notificações

A idéia por trás do uso de eventos é que um objeto pode reagir a uma alteração ocorrida em outro objeto. As notificações de eventos são basicamente assíncronas e determinadas pelos seus receptores. Em particular, nos aplicativos interativos, as ações executadas pelos usuários sobre objetos, por exemplo, clicando em um botão com o mouse ou digitando caracteres em uma caixa de texto por meio do teclado, são consideradas *eventos* que causam alterações nos objetos que mantêm o estado do aplicativo. Os objetos responsáveis por exibir a visão do estado corrente são *notificados* quando o estado muda.

Os sistemas distribuídos baseados em eventos ampliam o modelo de evento local, permitindo que vários objetos em diferentes localizações sejam notificados de eventos ocorrendo em um objeto. Para isso, eles usam o paradigma *publicar-assinar* (*publish-subscriber*), no qual um objeto que gera eventos *publica* (*publish*) os tipos de eventos que tornará disponíveis para observação por outros objetos. Os objetos interessados em um determinado evento fazem uma *assinatura* (*subscribe*) para receber notificações a respeito desse evento. Diferentes *tipos de evento* podem, por exemplo, invocar distintos métodos no objeto de interesse. Os objetos que representam eventos são chamados de *notificações*. As notificações podem ser armazenadas, enviadas em mensagens, consultadas e aplicadas em uma variedade de diferentes requisições. Quando um objeto gerar um evento, os assinantes que expressaram interesse nesse tipo de evento receberão notificações. Inscrever-se em um tipo particular de evento também é chamado de *registrar interesse* no evento.

Os eventos e as notificações podem ser usados em uma ampla variedade de aplicações; por exemplo, para comunicar a adição de uma figura em um desenho, uma modificação em um documento, o fato de uma pessoa ter entrado ou saído de uma sala, ou ainda, que um equipamentos ou livro etiquetados eletronicamente estão em um novo local. Estes dois últimos exemplos se tornam possíveis com o uso de crachás e etiquetas ativas (veja a Seção 16.1).

Os sistemas distribuídos baseados em eventos têm duas características principais:

Heterogêneos: quando notificações de evento são usadas como meio de comunicação entre objetos distribuídos, os componentes de um sistema distribuído que não foram projetados para interagir podem trabalhar em conjunto. Basta que os objetos que geram os eventos publiquem os tipos

de eventos que oferecem e que outros objetos se inscrevam nos eventos e forneçam uma interface para receber notificações. Por exemplo, Bates *et al.* [1996] descrevem como os sistemas baseados em eventos podem ser usados para conectar componentes heterogêneos na Internet. Eles descrevem um sistema no qual os aplicativos podem saber as localizações e as atividades dos usuários, como o momento em que estão usando computadores, impressoras ou livros etiquetados eletronicamente. Eles contemplam seu uso futuro no contexto de uma rede doméstica com comandos como: "se as crianças chegarem em casa, ligar o aquecimento central".

Assíncronos: as notificações são enviadas de forma assíncrona pelos objetos geradores de eventos, para todos os objetos que fizeram uma assinatura deles. O Mushroom [Kindberg *et al.* 1996] é um sistema distribuído baseado em eventos, projetado para suportar trabalho cooperativo, no qual a interface com o usuário exibe objetos que representam usuários e objetos de informação, como documentos e blocos de notas, dentro de espaços de trabalho compartilhados, chamados *locais de rede*. O estado de cada local de rede é replicado nos computadores dos usuários que fazem parte do local. Os eventos são usados para descrever alterações nos objetos e em focos de interesse de um usuário. Por exemplo, um evento poderia especificar que um usuário em particular entrou ou saiu de um local ou que executou uma ação em particular em um objeto. Cada réplica é assinante dos tipos de eventos que são relevantes para ela e, assim, recebem as notificações quando esses eventos ocorrem. Os assinantes são desvinculados dos objetos que geram eventos, pois os vários usuários estão ativos em diferentes ocasiões.

Uma situação na qual os eventos podem ser úteis está ilustrada no exemplo da operadora de ações, a seguir.

Uma corretora de ações simples ◊ Considere um sistema simples para uma corretora de ações, cuja tarefa é permitir que os operadores usem computadores para ver as informações mais recentes sobre os preços de mercado das ações que negociam. O preço de mercado de uma única ação é representado por um objeto com diversas variáveis de instância. As informações chegam à corretora a partir de várias fontes externas diferentes, na forma de atualizações de algumas, ou de todas, variáveis de instância dos objetos que representam as ações, e são coletadas por processos chamados de provedores de informação. Normalmente, os operadores estão interessados apenas nas ações de sua carteira. Um sistema para a corretora poderia ser modelado por processos com duas tarefas diferentes:

- Um processo provedor de informação recebe continuamente novas informações de compra e venda de uma única fonte externa e a aplica nos objetos de ação apropriados. Cada uma das atualizações em um objeto de ação é considerada um evento. O objeto de ação que está experimentando tais eventos notifica a todos os operadores que se inscreveram na ação correspondente. Haverá um processo provedor de informação separado para cada fonte externa.
- Um processo operador cria um objeto para representar cada ação cuja exibição foi solicitada pelo usuário. Esse objeto local se inscreve no objeto que representa essa ação, no provedor de informação relevante. Então, ele recebe todas as informações enviadas em notificações e as exibe para o usuário.

A comunicação das notificações aparece na Figura 5.10.

Tipos de evento ◊ Uma fonte de evento pode gerar eventos de um ou mais *tipos* diferentes. Cada evento tem *atributos* que especificam informações sobre esse evento, como o nome ou identificador do objeto que o gerou, a operação, seus parâmetros e a hora (ou um número de seqüência). Tipos e atributos são usados na inscrição dos eventos e em notificações. Ao se inscrever em um evento, o tipo do evento é especificado, às vezes com um critério aplicado sobre valores dos atributos. Quando ocorrer um evento desse tipo que corresponda aos atributos especificados, as partes interessadas serão notificadas. No exemplo da corretora de ações, existe um único tipo de evento (a chegada de uma atualização de uma ação) e os atributos podem especificar o nome de uma ação, seu preço corrente e a alta ou queda mais recente. Os operadores podem, por exemplo, especificar que estão interessados em todos os eventos relacionados a uma ação com um nome em particular.

Na Seção 5.4.1, apresentaremos uma arquitetura que especifica as funções dos participantes de uma notificação de evento distribuída. Essa arquitetura trata das características heterogênea e assín-

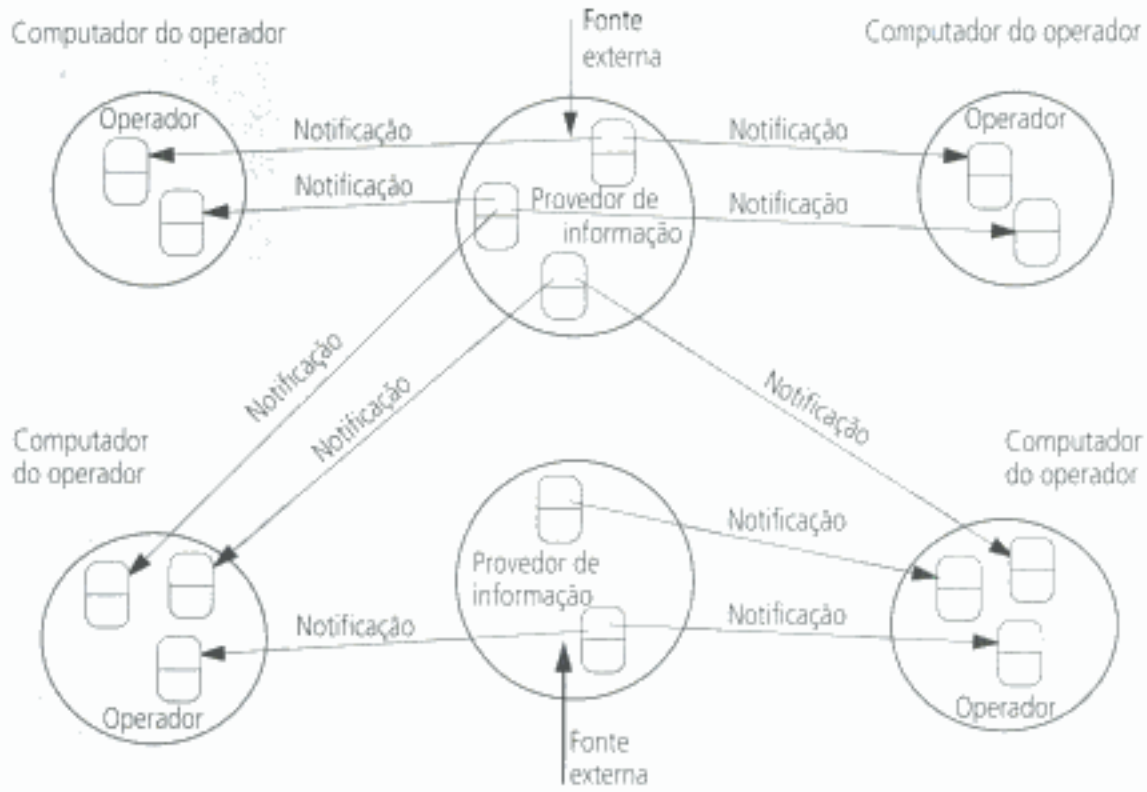


Figura 5.10 Sistema da corretora de ações.

crona da notificação de eventos. Isso é seguido por um estudo de caso do serviço de eventos Jini, na Seção 5.4.2. A Seção 16.3.1 do capítulo sobre computação móvel e ubíqua discutirá o uso de serviços de eventos para a comunicação de dados entre componentes em sistemas voláteis. A Seção 20.3.2, no estudo de caso do CORBA, apresentará o CORBA Event Service.

5.4.1 Os participantes da notificação de eventos distribuída

A Figura 5.11 mostra uma arquitetura que especifica as funções desempenhadas pelos objetos que participam dos sistemas distribuídos baseados em eventos. Nossa descrição é derivada do artigo sobre eventos e notificações na Internet, de Rosenblum e Wolf [1997]. A arquitetura é projetada de modo a desvincular os geradores de publicação dos assinantes, permitindo os geradores serem desenvolvidos independentemente de seus assinantes e, na medida do possível, limitando o trabalho imposto pelos assinantes sobre os geradores. O principal componente é um serviço de eventos que mantém um banco de dados de eventos publicados e dos interesses dos assinantes. Os eventos em um objeto de interesse são publicados no serviço de eventos. Os assinantes informam ao serviço de eventos sobre os tipos de eventos em que estão interessados. Quando ocorre um evento em um objeto, uma notificação é enviada para os assinantes interessados.

As funções dos objetos participantes são as seguintes:

O objeto de interesse: trata-se de um objeto que sofre mudanças de estado, como resultado da invocação de seus métodos. Essas mudanças de estado podem ter interesse para outros objetos. Esta descrição admite eventos como uma pessoa usando um crachá ativo entrando em uma sala, nesse exemplo, a sala é o objeto de interesse e a operação consiste na adição de informações sobre a pessoa, no registro de quem está na sala. O objeto de interesse é considerado parte do serviço de eventos, caso ele transmita notificações.

Evento: um evento ocorre em um objeto de interesse como resultado da conclusão da execução de um método.

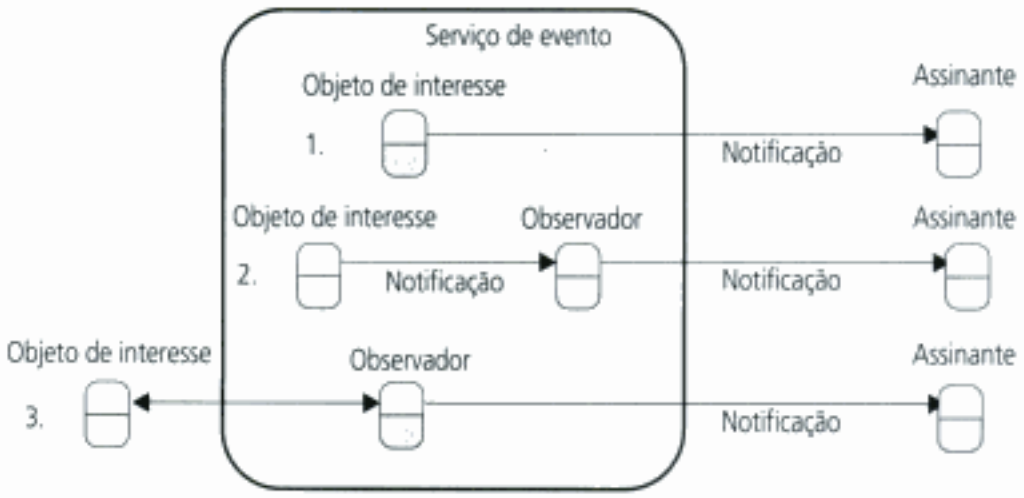


Figura 5.11 **Arquitetura de notificação de evento distribuída.**

Notificação: uma notificação é um objeto que contém informações sobre um evento. Normalmente, ela contém o tipo do evento e seus atributos, os quais geralmente incluem a identidade do objeto de interesse, o método invocado e a hora da ocorrência ou um número de seqüência.

Assinante: um assinante é um objeto que se inscreveu em algum tipo de evento em outro objeto. Ele recebe notificações sobre esses eventos.

Objetos observadores: o principal objetivo de um observador é desvincular um objeto de interesse de seus assinantes. Um objeto de interesse pode ter muitos assinantes diferentes, com interesses distintos. Por exemplo, os assinantes podem diferir quanto ao tipo de eventos em que estão interessados, ou aqueles que compartilham um mesmo tipo de evento podem diferir nos valores de atributo em que têm interesse. O objeto de interesse poderia ficar demasiadamente complicado, se tivesse que executar toda a lógica para distinguir entre as necessidades de seus assinantes. Um ou mais observadores podem ser inseridos entre um objeto de interesse e os assinantes. As funções dos observadores serão discutidas com mais detalhes a seguir.

Gerador de eventos (publisher): trata-se de um objeto que declara que vai gerar notificações de tipos de evento em particular. Um gerador pode ser um objeto de interesse ou um observador.

A Figura 5.11 mostra três casos:

1. Um objeto de interesse dentro do serviço de eventos sem um observador. Ele envia as notificações diretamente para os assinantes.
2. Um objeto de interesse dentro do serviço de eventos com um observador. O objeto de interesse envia as notificações para os assinantes por meio do observador.
3. Um objeto de interesse fora do serviço de eventos. Neste caso, um observador consulta o objeto de interesse para descobrir quando os eventos ocorrem. O observador envia as notificações para os assinantes.

Semânticas de entrega de notificações ◊ Uma variedade de garantias de entrega pode ser fornecida para as notificações – a escolhida deve depender dos requisitos das aplicações. Por exemplo, se for usado *multicast* IP para enviar notificações para um grupo de assinantes, o modelo de falha estará relacionado àquele descrito para o *multicast* IP, na Seção 4.5.1, e não garantirá que um assinante em particular receba uma dada mensagem de notificação. Isso é adequado para algumas aplicações; por exemplo, para distribuir o estado recente de um jogador em um jogo via Internet, nesse caso, a perda de uma atualização será compensada por uma próxima e provavelmente bastará para manter o estado do jogo coerente.

Entretanto, outras aplicações têm requisitos mais fortes. Considere a aplicação da corretora de ações: para sermos justos com os operadores interessados em uma ação em particular, precisamos que todos os operadores da mesma ação recebam as mesmas informações. Isso implica que deve ser usado um protocolo *multicast* confiável.

No sistema Mushroom, mencionado anteriormente, as notificações sobre a alteração do estado do objeto são distribuídas de modo confiável para um servidor, cuja responsabilidade é manter cópias atualizadas dos objetos. Entretanto, as notificações também podem ser enviadas para réplicas dos objetos nos computadores dos usuários por meio de *multicast* não confiável; no caso destas últimas perderem notificações, elas podem recuperar o estado a partir de um objeto do servidor. Quando a aplicação exigir, as notificações poderão ser ordenadas e enviadas de modo confiável para as réplicas dos objetos.

Algumas aplicações têm requisitos de tempo real. Isso inclui eventos em uma usina nuclear ou em um monitor de pacientes de um hospital. É possível projetar protocolos *multicast* que forneçam garantias em tempo real, assim como confiabilidade e ordenação, em um sistema que satisfaça as propriedades de um sistema distribuído síncrono.

Funções dos observadores ◊ Embora as notificações possam ser enviadas diretamente do objeto de interesse para o assinante, a tarefa de processar notificações pode ser dividida entre os processos observadores desempenhando uma variedade de funções diferentes. Descreveremos alguns exemplos:

Encaminhamento: um observador de encaminhamento pode realizar todo o trabalho de enviar notificações para os assinantes em nome de um ou mais objetos de interesse. Tudo que um objeto de interesse precisa é enviar uma notificação para o observador de encaminhamento, deixando-o continuar com sua tarefa normal. Para usar um observador de encaminhamento, um objeto de interesse passa as informações sobre os interesses de seus assinantes para esse observador.

Filtragem de notificações: filtros podem ser aplicados por um observador para reduzir o número de notificações recebidas, de acordo com algum predicado no conteúdo de cada notificação. Por exemplo, um evento poderia estar relacionado aos saques em uma conta bancária, mas o assinante está interessado apenas nos que forem maiores do que $100.

Padrões de eventos: quando um objeto se inscreve nos eventos de um objeto de interesse, ele pode especificar os padrões de eventos em que está interessado. Um padrão especifica um relacionamento entre vários eventos. Por exemplo, um assinante pode estar interessado em quando existem três saques de uma conta bancária sem um depósito interveniente. Um requisito semelhante é correlacionar eventos em uma variedade de objetos de interesse; por exemplo, notificar o assinante somente quando um certo número deles tiver gerado eventos.

Caixas de correio de notificação: em alguns casos, as notificações precisam ser postergadas até que um assinante em potencial esteja pronto para recebê-las. Por exemplo, se o assinante tiver conexões defeituosas ou quando um objeto tiver sido posto em um estado passivo e posteriormente reativado. Um observador pode assumir a função de caixa de correio de notificação, que serve para receber notificações em nome de um assinante, passando-as (em um único lote) apenas quando o assinante estiver pronto para recebê-las. O assinante deve ser capaz de ativar e desativar a entrega, conforme for exigido. O assinante configura uma caixa de correio de notificação ao registrar um objeto de interesse, especificando a caixa de correio como o local de envio de notificações.

5.4.2 Estudo de caso: especificação de eventos distribuídos Jini

A especificação de eventos distribuídos Jini, descrita por Arnold *et al.* [1999], permite que um assinante em potencial em uma máquina virtual Java (JVM) se inscreva e receba notificações sobre eventos em um objeto de interesse de outra JVM, normalmente em outro computador. Uma cadeia de observadores pode ser inserida entre o objeto de interesse e o assinante. Os principais objetos envolvidos na especificação de eventos distribuídos Jini são:

Geradores de evento: um gerador de eventos é um objeto que permite a outros objetos se inscreverem em seus eventos e que gera notificações.

Escutador de evento remoto*: um *escutador* de evento remoto é um objeto que pode receber notificações, ou seja, um assinante.

* N. de R.T.: O termo *escutador* é um neologismo empregado em vários livros sobre java como tradução para *listener*.

Eventos remotos: um evento remoto é um objeto que é passado por valor para os *escutadores* de evento remoto. Um evento remoto é o equivalente do que chamamos de notificação.

Agentes intermediários: agentes intermediários podem ser postos entre um objeto de interesse e um assinante. Eles são o equivalente aos observadores anteriormente descritos.

Um objeto se inscreve nos eventos informando o gerador de eventos sobre o tipo de evento e especificando um *escutador* de evento remoto como destino das notificações.

A RMI Java é usada para enviar notificações do gerador de eventos para o assinante, possivelmente por meio de um ou mais agentes interpostos. Os projetistas dizem que os *escutadores* de evento devem responder às chamadas de notificação, assim que possível, para evitar o atraso nos geradores de evento. Eles podem processar uma notificação após o retorno. A RMI Java também é usada para a assinatura de eventos. Os eventos Jini são fornecidos por meio das seguintes interfaces e classes:

RemoteEventListener: esta interface fornece um método chamado *notify*. Os assinantes e agentes intermediários implementam a interface *RemoteEventListener* para que possam receber notificações, quando o método *notify* é invocado. Uma instância da classe *RemoteEvent* representa uma notificação e é passada como argumento para o método *notify*.

RemoteEvent: esta classe tem variáveis de instância que contêm:

- uma referência para o gerador de eventos no qual o evento ocorreu;
- um identificador de evento, que especifica o tipo de evento nesse gerador de eventos;
- um número de seqüência, que se aplica aos eventos de um determinado tipo. O número de seqüência deve aumentar à medida que os eventos ocorrem com o passar do tempo. Ele pode ser usado para permitir aos assinantes ordenar eventos de determinada fonte ou para evitar a aplicação do mesmo evento duas vezes;
- um objeto empacotado. Isso é fornecido quando o receptor assina esse tipo de evento e pode ser usado por um assinante para qualquer propósito. Geralmente, ele contém as informações necessárias ao assinante para identificar o evento e reagir à sua ocorrência. Por exemplo, um assinante poderia incluir um método de encerramento a ser executado quando fosse notificado.

EventGenerator: esta interface fornece um método chamado *register*. Os geradores de evento implementam a interface *EventGenerator*, cujo método *register* é usado para assinar em eventos. Os argumentos de *register* especificam:

- um identificador de evento, que especifica o tipo de evento;
- um objeto empacotado a ser enviado de volta com cada notificação;
- uma referência remota para um objeto *escutador* de evento – o lugar para enviar notificações;
- um período de arrendamento (*leasing*) que especifica a duração do arrendamento exigida pelo assinante, mas a duração realmente garantida é retornada com os resultados de *register*. Limitar a duração da assinatura evita o problema dos geradores de evento contendo inscrições em eventos obsoletas. As assinaturas podem ser renovadas quando o limite de tempo do arrendamento expirar.

A especificação Jini diz que a interface *EventGenerator* é apenas um exemplo do tipo de interface que poderia ser usada pelos assinantes para registrar o interesse em eventos de um objeto de interesse. Algumas aplicações podem exigir uma interface diferente.

Agentes intermediários ◊ Os agentes intermediários que são interpostos entre um gerador de eventos e um assinante podem executar diversas funções úteis, incluindo todas aquelas descritas anteriormente.

No caso mais simples, um assinante registra o interesse em um tipo de evento em particular em um gerador de eventos e especifica-se como *escutador* de evento remoto. Isso corresponde ao caso 1, ilustrado na Figura 5.11.

Os agentes intermediários podem ser configurados por um gerador de eventos ou por um assinante.

Um gerador de eventos pode intercalar um ou mais agentes intermediários entre si e um assinante. Por exemplo, os geradores de evento de cada computador poderiam utilizar um agente intermediário compartilhado, responsável pela entrega confiável das notificações.

Um assinante pode construir uma cadeia de agentes intermediários para garantir uma política de entrega específica. Nesse caso, ele registra o interesse em um gerador de eventos, especificando o primeiro agente intermediário da seqüência de encadeamento como sendo o lugar para o envio de notificações. Por exemplo, um assinante pode fazer com que suas notificações sejam armazenadas por um agente intermediário, até estar pronto para recebê-las. O agente intermediário pode assumir a responsabilidade pela renovação dos arrendamentos.

5.5 Estudo de caso: RMI Java

A RMI Java estende o modelo de objeto Java para dar suporte para objetos distribuídos na linguagem Java. Em particular, ela permite que os objetos invoquem métodos em objetos remotos usando a mesma sintaxe das invocações locais. Além disso, a verificação de tipo se aplica igualmente às invocações remotas e às locais. Entretanto, um objeto que faz uma invocação remota sabe que seu destino é remoto, pois precisa tratar de exceções *RemoteException*; e o implementador de um objeto remoto sabe que ele é remoto porque precisa implementar a interface *Remote*. Embora o modelo de objeto distribuído seja integrado na linguagem Java de maneira natural, a semântica da passagem de parâmetros difere, pois o invocador e o alvo (destino) são remotos entre si.

A programação de aplicativos distribuídos na RMI Java é relativamente simples, pois se trata de um sistema desenvolvido com base em apenas uma linguagem – as interfaces remotas são definidas na linguagem Java. Se for usado um sistema que emprega várias linguagens em seu desenvolvimento, como o CORBA, o programador precisará aprender uma IDL e entender como faz o mapeamento em cada linguagem de implementação. Entretanto, mesmo quando é usada apenas uma linguagem de programação, o programador de um objeto remoto deve considerar seu comportamento em um ambiente concorrente.

No restante desta introdução, daremos um exemplo de interface remota e depois, com base nele, discutiremos a semântica da passagem de parâmetros. Finalmente, discutiremos o *download* de classes e o vinculador. A segunda seção deste estudo de caso discutirá como se faz a construção de programas cliente e servidor com base no exemplo de interface remota. A terceira seção se preocupará com o projeto e a implementação da RMI Java. Para detalhes completos sobre a RMI Java, consulte o exercício dirigido sobre invocação remota [java.sun.com 1].

Neste estudo de caso e no estudo de caso do CORBA, no Capítulo 20, assim como na discussão sobre serviços web, no Capítulo 19, usamos um quadro branco compartilhado como exemplo. Trata-se de um programa distribuído que permite um grupo de usuários compartilhar uma vista comum de uma superfície de desenho contendo objetos gráficos, como retângulos, linhas e círculos, cada um dos quais desenhado por um dos usuários. O servidor mantém o estado corrente de um desenho, fornecendo uma operação para os clientes informarem-no sobre a figura mais recente que seus usuários desenharam e mantendo um registro de todas as figuras que tiver recebido. O servidor também fornece operações que permitem aos clientes recuperarem as figuras mais recentes desenhadas por outros usuários, fazendo uma consulta seqüencial no servidor. O servidor tem um número de versão (um valor inteiro), que é incrementado sempre que chegar uma nova figura. O servidor fornece operações que permitem aos clientes perguntarem seu número de versão e o número de versão de cada figura, para que eles possam evitar a busca de figuras que já possuem.

Interfaces remotas na RMI Java ◊ As interfaces remotas são definidas pela ampliação de uma interface chamada *Remote*, fornecida no pacote *java.rmi*. Os métodos devem disparar a exceção *RemoteException*, mas exceções específicas do aplicativo também podem ser disparadas. A Figura 5.12 mostra um exemplo de duas interfaces remotas chamadas *Shape* e *ShapeList*. Nesse exemplo, *GraphicalObject* é uma classe que contém o estado de um objeto gráfico, por exemplo, seu tipo, sua posição, retângulo envoltório, cor da linha e cor de preenchimento, e fornece operações para acessar e atualizar seu estado. A classe *GraphicalObject* deve implementar a interface *Serializable*. Consi-

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException;                    1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;                2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Figura 5.12 Interfaces remotas *Shape* e *ShapeList*.

dere primeiro a interface *Shape*: o método *getVersion* retorna um valor inteiro, enquanto o método *getAllState* retorna uma instância da classe *GraphicalObject*. Agora, considere a interface *ShapeList*: seu método *newShape* passa como argumento uma instância de *GraphicalObject*, mas retorna como resultado um objeto com uma interface remota (isto é, um objeto remoto). Um ponto importante a notar é que tanto objetos locais como remotos podem aparecer como argumentos e resultados em uma interface remota. Estes últimos são sempre denotados pelo nome de suas interfaces remotas. No próximo parágrafo, discutiremos como os objetos locais e os objetos remotos são passados como argumentos e resultados.

Passagem de parâmetros e resultados ◊ Na RMI Java, supõe-se que os parâmetros de um método são parâmetros de *entrada* e o resultado de um método é um único parâmetro de *saída*. A Seção 4.3.2 descreveu a serialização Java, que é usada para empacotar argumentos e resultados na RMI Java. Qualquer objeto que seja serializável – isto é, que implemente a interface *Serializable* – pode ser passado como argumento ou ser resultado na RMI Java. Todos os tipos primitivos e objetos remotos são serializáveis. As classes de argumentos e valores de resultado são carregadas por *download* no destino pelo sistema RMI, quando necessário.

Passagem de objetos remotos: quando o tipo de um parâmetro ou valor de resultado é definido como uma interface remota, o argumento ou resultado correspondente é sempre passado como uma referência de objeto remoto. Por exemplo, na Figura 5.12, linha 2, o valor de retorno do método *newShape* é definido como *Shape* – uma interface remota. Quando uma referência de objeto remoto é recebida, ela pode ser usada para fazer chamadas RMI no objeto remoto a que se refere.

Passagem de objetos não-remotos: todos os objetos não-remotos serializáveis são copiados e passados por valor. Por exemplo, na Figura 5.12 (linhas 2 e 1), o argumento de *newShape* e o valor de retorno de *getAllState* são de tipo *GraphicalObject*, que é serializável e passado por valor. Quando um objeto é passado por valor, um novo objeto é criado no processo destino. Os métodos desse novo objeto podem ser invocados de forma local, possivelmente fazendo com que seu estado seja diferente do estado do objeto original no processo do remetente.

Assim, em nosso exemplo, o cliente usa o método *newShape* para passar uma instância de *GraphicalObject* para o servidor; o servidor cria um objeto remoto de tipo *Shape* contendo o estado de *GraphicalObject* e retorna uma referência de objeto remoto para ele. Os argumentos e valores de retorno em uma invocação remota são serializados em um fluxo de bytes, usando o método descrito na Seção 4.3.2, com as seguintes modificações:

1. Quando um objeto que implementa a interface *Remote* é serializado, ele é substituído por sua referência de objeto remoto, a qual contém o nome de sua classe (do objeto remoto).
2. Quando um objeto é serializado, suas informações de classe são anotadas com a localização da classe (como um URL), permitindo que a classe seja carregada por *download* pelo destino.

***Download* de classes** ◊ A linguagem Java é projetada para permitir que as classes sejam carregadas por *download* de uma máquina virtual para outra. Isso é particularmente relevante para objetos distribuídos que se comunicam por meio de invocação remota. Vimos que os objetos não-remotos são passados por valor e os objetos remotos são passados por referência, como argumentos e resultados das RMIs. Se o destino ainda não possuir a classe de um objeto passado por valor, seu código será carregado por *download* automaticamente. Analogamente, se o destino de uma referência de objeto remoto ainda não possuir a classe de um *proxy*, seu código será carregado por *download* automaticamente. Isso tem duas vantagens:

1. Não há necessidade de cada usuário manter o mesmo conjunto de classes em seu ambiente de trabalho.
2. Os programas clientes e servidores podem fazer uso transparente de instâncias de novas classes, quando elas forem adicionadas.

Como exemplo, considere o programa de quadro branco e suponha que sua implementação inicial de *GraphicalObject* não admita texto. Então, um cliente com um objeto textual pode implementar uma subclasse de *GraphicalObject*, que trata com texto, e passa uma instância para o servidor como argumento do método *newShape*. Depois disso, outros clientes poderão recuperar a instância usando o método *getAllState*. O código da nova classe será carregado por *download* automaticamente, do primeiro cliente para o servidor e depois, conforme for necessário, para outros clientes.

RMIregistry ◊ O RMIregistry é o vinculador da RMI Java. Uma instância de RMIregistry deve ser executada em cada computador servidor que contenha objetos remotos. Ele mantém uma tabela mapeando nomes textuais no estilo dos URLs, em referências para objetos remotos contidos nesse computador. Ele é acessado por métodos da classe *Naming*, cujos métodos recebem como argumento um string formatado como um URL, da forma:

//nomeComputador:porta/nomeObjeto

onde *nomeComputador* e *porta* se referem à localização do RMIregistry. Se eles forem omitidos, serão presumidos o computador local e a porta padrão. Sua interface oferece os métodos mostrados na Figura 5.13, na qual as exceções não estão listadas – todos os métodos podem disparar a exceção *RemoteException*. Não se trata de um serviço de vinculação no âmbito global do sistema. Os clientes devem direcionar suas requisições de *consulta* para computadores específicos.

void rebind (String name, Remote obj)
Este método é usado por um servidor para registrar o identificador de um objeto remoto pelo nome, como mostrado na Figura 5.14, linha 3.

void bind (String name, Remote obj)
Este método pode ser usado alternativamente por um servidor para registrar um objeto remoto pelo nome, mas se o nome já estiver vinculado a uma referência de objeto remoto, uma exceção será disparada.

void unbind (String name, Remote obj)
Este método remove um vínculo.

Remote lookup(String name)
Este método é usado pelos clientes para procurar um objeto remoto pelo nome, como mostrado na Figura 5.16, linha 1. É retornada uma referência de objeto remoto.

String [] list()
Este método retorna um array de *Strings* contendo os nomes vinculados no registro.

Figura 5.13 A classe *Naming* do RMIregistry Java.

5.5.1 Construindo programas cliente e servidor

Esta seção esboça as etapas necessárias para produzir programas cliente e servidor que utilizam as interfaces *Remote Shape* e *ShapeList*, mostradas na Figura 5.12. O programa servidor é uma versão simplificada do servidor de quadro branco que implementa as duas interfaces *Shape* e *ShapeList*. Descreveremos um programa cliente de consulta seqüencial simples e depois apresentaremos a técnica de *callback*, que pode ser usada para evitar a necessidade de fazer a consulta seqüencial no servidor. Versões completas das classes ilustradas nesta seção estão disponíveis no endereço [www.cdk4.net/rmi].

Programa servidor ◊ O programa é um servidor de quadro branco: ele representa cada figura como um objeto remoto instanciado por um servente que implementa a interface *Shape* e contém o estado de um objeto gráfico, assim como seu número de versão; ele representa sua coleção de figuras por meio de outro servente, que implementa a interface *ShapeList* e contém uma coleção de figuras em um *Vector*.

O servidor consiste em um método *main* e uma classe servente para implementar cada uma de suas interfaces remotas. O método *main* do servidor cria uma instância de *ShapeListServant* e a vincula a um nome no RMIregistry, como mostrado na Figura 5.14 (linhas 1 e 2). Note que o valor vinculado ao nome é uma referência de objeto remoto e seu tipo é o tipo de sua interface remota – *ShapeList*. As duas classes serventes são *ShapeListServant*, que implementa a interface *ShapeList*, e *ShapeServant*, para a interface *Shape*. A Figura 5.15 fornece um esboço da classe *ShapeListServant*. Note que *ShapeListServant* (linha 1), assim como muitas classes serventes, estende uma classe chamada *UnicastRemoteObject*, que fornece objetos remotos que existem apenas enquanto estão no processo em que são criados. (Um objeto passível de ativação estenderia uma classe chamada *Activatable*.)

As implementações dos métodos da interface remota em uma classe servente é extremamente simples, pois elas podem ser feitas sem nenhuma preocupação com os detalhes da comunicação. Considere o método de *newShape* na Figura 5.15 (linha 2), que poderia ser chamado de método de fábrica, pois ele permite que o cliente solicite a criação de um servente. Ele usa o construtor de *ShapeServant*, o qual cria um novo servente contendo o objeto *GraphicalObject* e o número de versão passados como argumentos. O tipo do valor de retorno de *newShape* é *Shape* – a interface implementada pelo novo servente. Antes de retornar, o método *newShape* adiciona a nova figura em seu vetor que contém a lista de figuras (linha 3).

O método *main* de um servidor precisa criar um gerenciador de segurança para permitir que a linguagem Java aplique a proteção apropriada para o servidor RMI. É fornecido um gerenciador de segurança padrão, chamado *RMISecurityManager*. Ele protege os recursos locais para garantir que as classes carregadas a partir de sites remotos não possam ter qualquer efeito sobre recursos como, por exemplo, arquivos, porém permite que o programa forneça seu próprio carregador de classe e use reflexão. Se um servidor RMI não configurar nenhum gerenciador de segurança, os *proxies* e as classes só poderão ser carregados a partir do caminho de classe local. O objetivo é proteger o programa do código que é carregado por *download* como resultado das invocações a métodos remotos.

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();                    1
            Naming.rebind("Shape List", aShapeList );                         2
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Figura 5.14 Classe *ShapeListServer* Java com o método *main*.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
    private Vector theList;// contém a lista de elementos Shape                    1
    private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException {             2
        version++;
        Shape s = new ShapeServant( g, version);                                  3
        theList.addElement(s);
        return s;
    }
    public Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
}
```

Figura 5.15 A classe *ShapeListServant* Java que implementa a interface *ShapeList*.

Programa cliente ◊ Um cliente simplificado para o servidor *ShapeList* está ilustrado na Figura 5.16. Todo programa cliente precisa ser iniciado usando um vinculador para pesquisar uma referência de objeto remoto. Nosso cliente configura um gerenciador de segurança e depois pesquisa uma referência de objeto remoto usando a operação *lookup* do RMIregistry (linha 1). Tendo obtido uma referência de objeto remoto inicial, o cliente continua, enviando RMIs para esse objeto remoto ou para outros objetos descobertos durante sua execução, de acordo com as necessidades de seu aplicativo. Em nosso exemplo, o cliente invoca o método *allShapes* no objeto remoto (linha 2) e recebe um vetor de referências de objeto remoto para todas as figuras correntemente armazenadas no servidor. Se o cliente implementasse uma tela para o quadro branco, ele usaria o método *getAllState* do servidor na interface *Shape* para recuperar cada um dos objetos gráficos do vetor e os exibir na janela. Sempre que o usuário terminar o desenho de um objeto gráfico, ele invocará o método *newShape* no servidor, passando o novo objeto gráfico como argumento. O cliente manterá um registro do número de versão mais recente no servidor e, de tempos em tempos, invocará *getVersion* no servidor para descobrir se figuras novas foram adicionadas por outros usuários. Se assim for, ele as recuperará e exibirá.

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList");          1
            Vector sList = aShapeList.allShapes();                                2
        } catch(RemoteException e) {System.out.println(e.getMessage());
        }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```

Figura 5.16 Cliente Java para *ShapeList*.

Callbacks ◊ A idéia geral por trás dos *callbacks* é que, em vez dos clientes fazerem consultas no servidor para descobrir se ocorreu algum evento, o servidor informa os clientes quando o evento ocorrer. O termo *callback* é usado para referenciar a ação de um servidor ao notificar os clientes sobre um evento. Os *callbacks* podem ser implementados na RMI como segue:

- O cliente cria um objeto remoto que implementa uma interface que contém um método para o servidor chamar. Nos referimos a isso como objeto de *callback*.
- O servidor fornece uma operação que permite aos clientes lhe informar as referências de objeto remoto de seus objetos de *callback*. O servidor as registra em uma lista.
- Quando ocorre um evento, o servidor chama os clientes interessados. Por exemplo, o servidor de quadro branco chamaria seus clientes quando um objeto gráfico fosse adicionado.

O uso de *callbacks* evita a necessidade de um cliente fazer consultas sistemáticas ao servidor para verificar seus objetos de interesse e suas conseqüentes desvantagens:

- O desempenho do servidor pode ser degradado pelas constantes consultas.
- Os clientes podem não notificar os usuários sobre as atualizações.

Entretanto, os *callbacks* têm seus próprios problemas: primeiro, o servidor precisa ter listas atualizadas dos objetos de *callback* dos clientes. Mas os clientes podem não informar o servidor antes de terminarem, deixando o servidor com listas incorretas. A técnica de arrendamento, discutida na Seção 5.2.6, pode ser usada para superar esse problema. O segundo problema associado aos *callbacks* é que o servidor precisa fazer uma série de RMIs síncronas nos objetos de *callback* da lista. Consulte a Seção 5.4.1 e o Exercício 5.18 para ver algumas idéias sobre como resolver o segundo problema.

Ilustramos o uso de *callbacks* no contexto do aplicativo de quadro branco. A interface *WhiteboardCallback* poderia ser definida como:

```
public interface WhiteboardCallback implements Remote {
    void callback(int version) throws RemoteException;
};
```

Essa interface é implementada como um objeto remoto pelo cliente, permitindo que o servidor envie ao cliente um número de versão quando um novo objeto é adicionado. Mas para o servidor fazer isso é necessário que o cliente informe o seu objeto de *callback*. Para tornar isso possível, a interface *ShapeList* exige métodos adicionais, como *register* e *deregister*, definidos como segue:

```
int register(WhiteboardCallback callback) throws RemoteException;
void deregister(int callbackId) throws RemoteException;
```

Após o cliente ter obtido uma referência para o objeto remoto com a interface *ShapeList* (por exemplo, na Figura 5.16, linha 1) e criado uma instância de objeto de *callback*, ele utiliza o método *register* de *ShapeList* para registrar no servidor o seu interesse em receber *callbacks*. O método *register* retorna um valor inteiro (um *callbackId*) que serve como identificador desse registro. Quando o cliente tiver terminado, ele deve chamar *deregister* para informar o servidor que não quer mais os *callbacks*. O servidor é responsável por manter uma lista dos clientes interessados e por notificar todos eles sempre que seu número de versão aumentar.

5.5.2 Projeto e implementação da RMI Java

O sistema RMI Java original usava todos os componentes mostrados na Figura 5.7. Mas no Java 1.2, os recursos de reflexão foram usados para fazer um despachante genérico e para evitar a necessidade de esqueletos. Os *proxies* clientes são gerados por um compilador chamado *rmic*, a partir das classes de servidor compiladas – e não das definições das interfaces remotas.

Uso de reflexão ◊ A reflexão é usada para passar informações, nas mensagens de requisição, sobre o método a ser invocado. Isso é obtido através da classe *Method* no pacote de reflexão. Cada instância de *Method* representa as características de um método em particular, incluindo sua classe, os tipos de seus

argumentos, o valor de retorno e exceções. A característica mais interessante dessa classe é que uma instância de *Method* pode ser invocada em um objeto de uma classe, por intermédio de seu método *invoke*. O método *invoke* exige dois argumentos: o primeiro especifica o objeto que vai receber a invocação e o segundo é um array de *Object* contendo os argumentos. O resultado é retornado como tipo *Object*.

Voltando ao uso da classe *Method* na RMI: o *proxy* precisa empacotar informações sobre um método e seus argumentos na mensagem de requisição. Para o método, ele empacota um objeto da classe *Method*. Ele coloca os argumentos em um array de elementos *Object* e depois empacota esse array. O despachante desempacota o objeto *Method* e seus argumentos no array de elementos *Object* da mensagem de *pedido*. Como sempre, a referência de objeto remoto do destino terá sido desempacotada e a referência de objeto local correspondente, obtida do módulo de referência remota. Então, o despachante chama o método *invoke* do objeto *Method*, fornecendo o destino e o array de valores de argumento. Quando o método tiver sido executado, o despachante empacotará o resultado ou as exceções na mensagem de *resposta*.

Assim, o despachante é genérico – isto é, o mesmo despachante pode ser usado por todas as classes de objeto remoto e nenhum esqueleto é exigido.

Classes Java que suportam RMI ◊ A Figura 5.17 mostra a estrutura de herança das classes que suportam servidores Java RMI. A única classe que o programador necessita conhecer é *UnicastRemoteObject*, que toda classe servente precisa estender. A classe *UnicastRemoteObject* estende uma classe abstrata chamada *RemoteServer*, a qual fornece versões abstratas dos métodos exigidos pelos servidores remotos. *UnicastRemoteObject* foi o primeiro exemplo de *RemoteServer* a ser fornecido. Outro método interessante, chamado *Activatable*, está disponível para fornecer objetos passíveis de ativação. Há também classes para fornecer suporte a replicação de objetos. A classe *RemoteServer* é uma subclasse de *RemoteObject* que tem uma variável de instância contendo a referência de objeto remoto e fornece os seguintes métodos:

equals: este método compara referências de objeto remoto;

toString: este método fornece o conteúdo da referência de objeto remoto como um *String*;

readObject, *writeObject*: estes métodos desserializam/serializam objetos remotos.

Além disso, o operador *instanceOf* pode ser usado para testar objetos remotos.

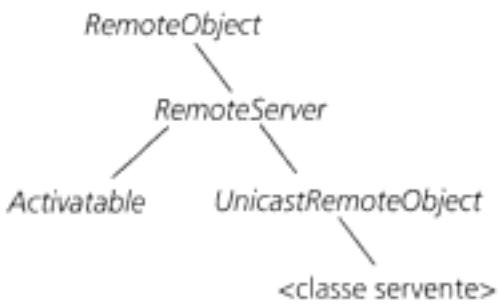


Figura 5.17 Classes que suportam a RMI Java.

5.6 Resumo

Este capítulo discutiu dois paradigmas da programação distribuída – a invocação a método remoto e os sistemas baseados em eventos. Esses dois paradigmas consideram os objetos distribuídos como entidades independentes que podem se comunicar. No primeiro caso, um método na interface remota de um objeto em particular é invocado de forma síncrona – com o invocador esperando por uma resposta. No segundo caso, as notificações são enviadas de forma assíncrona para vários assinantes, quando ocorre um evento em um objeto de interesse.

O modelo de objeto distribuído é uma ampliação do modelo de objeto local usado nas linguagens de programação baseadas em objetos. Os objetos encapsulados formam componentes úteis em um sistema distribuído, pois o encapsulamento os tornam inteiramente responsáveis por gerenciar seus próprios estados e as invocações a métodos locais podem ser estendidas para invocações remotas. Cada objeto em um sistema distribuído tem uma referência de objeto remoto (um identificador globalmente exclusivo) e uma interface remota que especifica quais de suas operações podem ser invocadas de forma remota.

As invocações a métodos locais fornecem a semântica *exatamente uma vez*, enquanto as invocações a métodos remotos não podem garantir o mesmo, pois os dois objetos participantes estão em computadores diferentes, os quais podem falhar independentemente e estão ligados por uma rede, que também pode falhar. O melhor que pode ser conseguido é a semântica *no máximo uma vez*. Devido às suas diferentes características de falhas e desempenho e à possibilidade de acesso concorrente aos objetos remotos, não é necessariamente uma boa idéia fazer a invocação remota parecer ser exatamente igual à uma invocação local.

As implementações de *middleware* da RMI fornecem componentes (incluindo *proxies*, esqueletos e despachantes) que ocultam aos programadores do cliente e do servidor os detalhes do empacotamento, passagem de mensagem e da localização de objetos remotos. Esses componentes podem ser gerados por um compilador de interface. A RMI Java estende a invocação local em remota usando a mesma sintaxe, mas as interfaces remotas devem ser especificadas estendendo uma interface chamada *Remote* e fazendo com que cada método dispare uma exceção *RemoteException*. Isso garante que os programadores saibam quando fazem invocações remotas ou implementam objetos remotos, permitindo a eles tratar de erros ou projetar objetos convenientes para acesso concorrente.

Os sistemas distribuídos baseados em eventos podem ser usados para permitir que conjuntos distribuídos de objetos heterogêneos se comuniquem. Ao contrário da RMI, os objetos não precisam ter interfaces remotas para receber mensagens – basta implementarem uma interface para receberem notificações e para assinarem seu interesse em eventos. Os objetos que geram eventos precisam enviar notificações assíncronas. A simplicidade das interfaces deve facilitar a adição de eventos em objetos existentes. O trabalho adicional de processar eventos (por exemplo, filtrar e procurar padrões) pode ser realizado por observadores – objetos intermediários, adicionados ao sistema para esse propósito.

Exercícios

5.1 A interface *Election* fornece dois métodos remotos:

vote: possui dois parâmetros por meio dos quais o cliente fornece o nome de um candidato (um string) e o "número do votante" (um valor inteiro usado para garantir que cada usuário vote apenas uma vez). Os números dos votantes são alocados esparsamente a partir do intervalo de inteiros para torná-los difíceis de adivinhar.

result: possui dois parâmetros com os quais o servidor fornece para o cliente o nome de um candidato e o número de votos desse candidato.

Quais dos parâmetros desses dois métodos são de *entrada* e quais são parâmetros de *saída*?

página 164–165

5.2 Discuta a semântica de invocação que pode ser obtida quando o protocolo requisição-resposta é implementado sobre uma conexão TCP/IP, a qual garante que os dados são distribuídos na ordem enviada, sem perda nem duplicação. Leve em conta todas as condições que causam a perda da conexão. *Seção 4.2.4 e página 171–172*

5.3 Defina a interface do serviço *Election* na IDL CORBA e na RMI Java. Note que a IDL CORBA fornece o tipo *long* para inteiros de 32 bits. Compare os métodos nas duas linguagens, para especificar argumentos de *entrada* e *saída*. *Figura 5.2, Figura 5.12*

5.4 O serviço *Election* deve garantir que um voto seja registrado quando o usuário achar que depositou o voto.

Discuta o efeito da semântica *talvez* no serviço *Election*.

A semântica *pelo menos uma vez* seria aceitável para o serviço *Election* ou você recomendaria a semântica *no máximo uma vez*? *página 172–173*

5.5 Um protocolo de requisição-resposta é implementado em um serviço de comunicação com falhas por omissão para fornecer semântica de invocação RMI *pelo menos uma vez*. No primeiro caso, o desenvolvedor presume um sistema assíncrono distribuído. No segundo caso, o desenvolvedor presume que o tempo máximo para a comunicação e a execução de um método remoto é *T*. De que maneira esta última suposição simplifica a implementação? *página 171–172*

5.6 Esboce uma implementação para o serviço *Election* que garanta que seus registros permaneçam consistentes quando ele é acessado simultaneamente por vários clientes. *página 173*

5.7 O serviço *Election* deve garantir que todos os votos sejam armazenados com segurança, mesmo quando o processo servidor falha. Explique como isso pode ser conseguido no esboço de implementação de sua resposta para o Exercício 5.6. *páginas 160–161*

5.8 Mostre como se usa reflexão Java para construir a classe *proxy* cliente para a interface *Election*. Forneça os detalhes da implementação de um dos métodos dessa classe, o qual deve chamar o método *doOperation* com a seguinte assinatura:

byte [] doOperation (RemoteObjectRef o, Method m, byte[] arguments);

Dica: uma variável de instância da classe *proxy* deve conter uma referência de objeto remoto (veja o Exercício 4.12). *Figura 4.15, página 194–195*

5.9 Mostre como se gera uma classe *proxy* cliente usando uma linguagem como C++, que não suporta reflexão, por exemplo, a partir da definição de interface CORBA dada em sua resposta para o Exercício 5.3. Forneça os detalhes da implementação de um dos métodos dessa classe, o qual deve chamar o método *doOperation* definido na Figura 4.15. *página 175–176*

5.10 Explique como se faz para usar reflexão Java para construir um despachante genérico. Forneça o código Java de um despachante cuja assinatura seja:

public void dispatch(Object target, Method aMethod, byte[] args)

Os argumentos fornecem o objeto de destino, o método a ser invocado e os argumentos desse método, em um array de bytes. *página 194–195*

5.11 O Exercício 5.8 exigia que o cliente convertesse argumentos *Object* em um array de bytes, antes de ativar *doOperation*, e o Exercício 5.10 exigia que o despachante convertesse um array de bytes em um array de elementos *Object*, antes de invocar o método. Discuta a implementação de uma nova versão de *doOperation* com a seguinte assinatura:

Object [] doOperation (RemoteObjectRef o, Method m, Object[] arguments);

que usa as classes *ObjectOutputStream* e *ObjectInputStream* para comunicar as mensagens de requisição-resposta entre cliente e servidor por meio de uma conexão TCP. Como essas alterações afetariam o projeto do despachante? *Seção 4.3.2 e página 194–195*

5.12 Um cliente faz chamadas de procedimento remoto a um servidor. O cliente demora 5 milissegundos para computar os argumentos de cada requisição e o servidor demora 10 milissegundos para processar cada requisição. O tempo de processamento do sistema operacional local para cada operação de envio ou recepção é de 0,5 milissegundos e o tempo que a rede leva para transmitir cada mensagem de requisição ou resposta é de 3 milissegundos. O empacotamento ou desempacotamento demora 0,5 milissegundos por mensagem.

Calcule o tempo que leva para o cliente gerar e retornar duas requisições:

(i) se ele tiver uma só *thread* e

(ii) se ele tiver duas *threads* que podem fazer requisições concorrentes em um único processador.

Você pode ignorar os tempos de troca de contexto. Há necessidade de RPC assíncrona se os processos cliente e servidor forem programados com múltiplas *threads*? *página 176–177*

5.13 Projete uma tabela de objetos remotos que possa suportar coleta de lixo distribuída, assim como fazer a transformação entre referências de objeto local e remota. Dê um exemplo envolvendo vários objetos remotos e *proxies* em diversos sites para ilustrar o uso da tabela. Mostre as alterações na tabela quando uma invocação faz um novo *proxy* ser criado. Em seguida, mostre as alterações na tabela quando um dos *proxies* se torna inatingível. *página 178*

5.14 Uma versão mais simples do algoritmo de coleta de lixo distribuída, descrito na Seção 5.2.6, apenas invoca *addRef* no site onde está um objeto remoto, quando um *proxy* é criado, e *removeRef*, quando um *proxy* é excluído. Esboce todos os efeitos possíveis das falhas de comunicação e de processos no algoritmo. Sugira como fazer para superar todos esses efeitos, mas sem usar arrendamentos. *página 178*

5.15 Discuta como se faz para usar eventos e notificações, conforme descrito na especificação de evento distribuído Jini, no contexto do aplicativo de quadro branco compartilhado. A classe *RemoteEvent* é definida como segue em Arnold *et al.* [1999].

```
public class RemoteEvent extends java.util.EventObject {
    public RemoteEvent(Object source, long eventID,
        long seqNum, MarshalledObject handback)
    public Object getSource () {...}
    public long getID() {...}
    public long getSequenceNumber() {...}
    public MarshalledObject getRegistrationObject() {...}
}
```

O primeiro argumento do construtor é um objeto remoto. As notificações informam aos *escutadores* que um evento ocorreu, mas estes são responsáveis por obter mais detalhes. *páginas 187, 188*

5.16 Sugira um projeto para um serviço de caixa de correio de notificação que seja destinado a armazenar notificações em nome de vários assinantes, permitindo a eles especificarem quando querem que as notificações sejam distribuídas. Explique como os assinantes que não estão sempre ativos podem utilizar o serviço descrito por você. Como o serviço tratará dos assinantes que falham, enquanto estão distribuindo notificações? *páginas 187–188*

5.17 Explique como um observador de encaminhamento pode ser usado para melhorar a confiabilidade e o desempenho de objetos de interesse em um serviço de eventos. *páginas 186–187*

5.18 Sugira maneiras pelas quais observadores podem ser usados para melhorar a confiabilidade ou o desempenho de sua solução para o Exercício 5.15. *páginas 186–187*

6 Sistema Operacional

Este capítulo descreve como o *middleware* é suportado pelos recursos do sistema operacional nos nós de um sistema distribuído. O sistema operacional facilita o encapsulamento e a proteção dos recursos dentro dos servidores e disponibiliza mecanismos de invocação necessários para acessar esses recursos, incluindo a comunicação e o escalonamento.

Um tema importante do capítulo é a função do núcleo do sistema operacional. O capítulo pretende fazer com que o leitor entenda as vantagens e desvantagens da divisão da funcionalidade entre os domínios de proteção – em particular, da divisão da funcionalidade entre o núcleo e o código em nível de usuário. Serão discutidos os compromissos entre os recursos em nível de núcleo e em nível de usuário, incluindo o compromisso entre eficiência e robusteza.

O capítulo examina o projeto e a implementação de *multithreading* e de recursos de comunicação. Ele explora as principais arquiteturas de núcleo de sistemas operacionais que já foram inventadas.

6.1 Introdução

O Capítulo 2 apresentou as camadas de software mais importantes de um sistema distribuído. Aprendemos que um aspecto importante dos sistemas distribuídos é o compartilhamento de recursos. Os aplicativos clientes invocam operações em recursos que freqüentemente estão em outro nó, ou pelo menos em outro processo. Aplicativos (na forma de clientes) e serviços (na forma de gerenciadores de recursos) usam a camada de *middleware* para suas interações. O *middleware* fornece invocações remotas entre objetos ou processos nos nós de um sistema distribuído. O Capítulo 5 explicou os principais tipos de invocação remota encontrados na camada de *middleware*, como a RMI Java e o CORBA. Neste capítulo, continuaremos a focalizar as invocações remotas, sem garantias de tempo real. (O Capítulo 17 examinará o suporte para comunicação multimídia, que se dá em tempo real e é orientada a fluxo.)

Abaixo da camada de *middleware* está a camada do sistema operacional (SO), que é o assunto deste capítulo. Vamos examinar o relacionamento entre as duas e, em particular, até que ponto os requisitos do *middleware* podem ser satisfeitos pelo sistema operacional. Esses requisitos incluem o acesso eficiente e robusto aos recursos físicos e a flexibilidade para implementar uma variedade de políticas de gerenciamento de recursos.

A tarefa de qualquer sistema operacional é fornecer abstrações dos recursos físicos subjacentes – processadores, memória, comunicação e mídias de armazenamento. Um sistema operacional como o UNIX (e suas variantes, como o Linux) ou o Windows (e suas variantes, como o XP) fornecem ao programador, por exemplo, a abstração de arquivos, em vez de blocos de disco, e soquetes, em vez de acesso direto à rede. O sistema operacional gerencia os recursos físicos de um computador apresentado-os através de suas abstrações via uma interface denominada de chamada de sistema.

Antes de iniciarmos nossa abordagem detalhada do suporte que um sistema operacional oferece para *middlewares*, é interessante ter certa perspectiva histórica, examinando dois conceitos do sistema operacional que estão relacionados ao desenvolvimento de sistemas distribuídos: sistemas operacionais de rede e sistemas operacionais distribuídos. As definições variam, mas os conceitos por trás delas são os que aparecem a seguir.

O UNIX e o Windows são exemplos de *sistemas operacionais de rede*. Eles têm um recurso de interligação em rede incorporado e, portanto, podem ser usados para acessar recursos remotos. O acesso é transparente com relação à rede para alguns tipos de recurso, mas não para todos. Por exemplo, por meio de um sistema de arquivos distribuído, como o NFS, os usuários têm acesso transparente aos arquivos no que diz respeito à rede. Isto é, muitos arquivos que os usuários acessam são armazenados de forma remota em um servidor e isso é totalmente transparente para seus aplicativos.

Mas a característica marcante é que os nós que estão sendo executados em um sistema operacional de rede mantêm a autonomia no gerenciamento de seus próprios recursos de processamento. Em outras palavras, existem várias imagens do sistema, uma por nó. Com um sistema operacional de rede, um usuário pode se conectar em outro computador de forma remota, usando *rlogin* ou *telnet*, e executar processos nele. Entretanto, ao contrário do controle exercido pelo sistema operacional sobre os processos que estão sendo executados em seu próprio nó, ele não escalona os processos nos vários nós.

Em contraste, alguém poderia imaginar um sistema operacional no qual o usuário nunca se preocupasse com o local onde seus programas são executados, ou com a localização de quaisquer recursos. Haveria uma *única imagem do sistema*. O sistema operacional teria controle sobre todos os nós do sistema e dispararia novos processos de forma transparente, no nó mais conveniente, de acordo com sua política de escalonamento. Por exemplo, o sistema operacional poderia criar um novo processo no nó menos carregado do sistema para evitar que nós individuais se tornassem desnecessariamente sobrecarregados.

Um sistema operacional que produz uma única imagem do sistema como esse, para todos os recursos de um sistema distribuído, é chamado de *sistema operacional distribuído* [Tanenbaum e van Renesse 1985].

***Middleware* e sistemas operacionais de rede** ◊ Na verdade, não existem sistemas operacionais distribuídos para uso geral, existem apenas sistemas operacionais de rede como UNIX, Mac OS e Windows. É provável que continue assim, por dois motivos principais. O primeiro é que os usuários

têm muito investimento feito em software aplicativo, o qual freqüentemente atende suas necessidades atuais; eles não adotarão um novo sistema operacional que não execute seus aplicativos, independente das vantagens relacionadas à eficiência que ele ofereça. Foram feitas tentativas de simular o UNIX, e outros núcleos de sistema operacional, sobre novos núcleos, mas o desempenho das simulações não foi satisfatório. De qualquer modo, manter atualizadas simulações de todos os principais sistemas operacionais, à medida que eles evoluem, seria uma tarefa enorme.

O segundo motivo contra a adoção de sistemas operacionais distribuídos é que os usuários preferem ter certo grau de autonomia em suas máquinas, mesmo em uma empresa muito fechada. Isso acontece particularmente devido ao desempenho [Douglis e Ousterhout 1991]. Por exemplo, Jones precisa de boa capacidade de resposta interativa enquanto escreve seus documentos e se ressentiria se os programas de Smith diminuíssem a velocidade de processamento de sua máquina.

A combinação de *middleware* e sistemas operacionais de rede proporciona um equilíbrio aceitável entre os requisitos de autonomia, por um lado, e o acesso aos recursos, transparente com relação à rede, por outro. O sistema operacional de rede permite que os usuários executem, de forma independente dos demais, seu processador de textos predileto e outros aplicativos. A camada de *middleware* permite que eles tirem proveito de serviços distribuídos que se tornem disponíveis para seu sistema operacional.

A próxima seção explicará a função da camada do sistema operacional. A Seção 6.3 examinará os mecanismos de baixo nível para proteção de recursos, os quais precisamos entender para avaliar o relacionamento entre processos e *threads* e a função do núcleo em si. A Seção 6.4 examinará as abstrações de processo, espaço de endereçamento e de *thread*. Aqui, os principais tópicos são a concorrência, o gerenciamento e a proteção de recurso local, e o escalonamento. Em seguida, a Seção 6.5 abordará a comunicação como parte dos mecanismos de invocação. A Seção 6.6 discutirá os diferentes tipos de arquitetura de sistemas operacionais, incluindo os assim chamados projetos monolíticos e de micronúcleo. O leitor poderá encontrar estudos de caso do núcleo Mach e dos sistemas operacionais Amoeba, Chorus e Clouds, no endereço [www.cdk4.net/oss].

6.2 A camada do sistema operacional

Os usuários só ficarão satisfeitos se sua combinação de *middleware*–SO tiver bom desempenho. Em um sistema distribuído, o *middleware* pode ser executado sobre uma variedade de sistemas operacionais, sobre diferentes hardwares, em cada nó. O par SO-hardware é genericamente denominado de plataforma. O SO que está sendo executado em um nó – composto por um núcleo e por serviços em nível de usuário, por exemplo, bibliotecas – fornece de acordo com seus recursos de hardware locais seu próprio conjunto de abstrações para processamento, armazenamento e comunicação. O *middleware* utiliza essas abstrações para implementar seus mecanismos de invocações remotas entre objetos ou processos nos nós.

A Figura 6.1 mostra como a camada de sistema operacional, em cada um de dois nós, oferece suporte para uma camada de *middleware* comum para fornecer uma infra-estrutura distribuída para aplicativos e serviços.

Nosso objetivo neste capítulo é examinar o impacto dos mecanismos do SO, em particular, sobre a capacidade do *middleware* de apresentar compartilhamento de recursos distribuído para os usuários. Os núcleos e os processos clientes e servidores neles executados são os principais componentes que nos interessam. Os núcleos e os processos servidores são os componentes que gerenciam os recursos e que os apresentam aos clientes através de uma interface. Desse modo, exigimos deles pelo menos o seguinte:

Encapsulamento: eles devem fornecer uma interface de serviço útil para seus recursos – isto é, um conjunto de operações que satisfaça as necessidades de seus clientes. Os detalhes, como gerenciamento de memória e os dispositivos usados para implementar os recursos, devem ser ocultados dos clientes.

Proteção: os recursos exigem proteção contra acessos ilegítimos – por exemplo, os arquivos são protegidos contra leitura de usuários sem permissões de leitura e os registradores dos dispositivos de E/S são protegidos contra acessos de processos aplicativos.

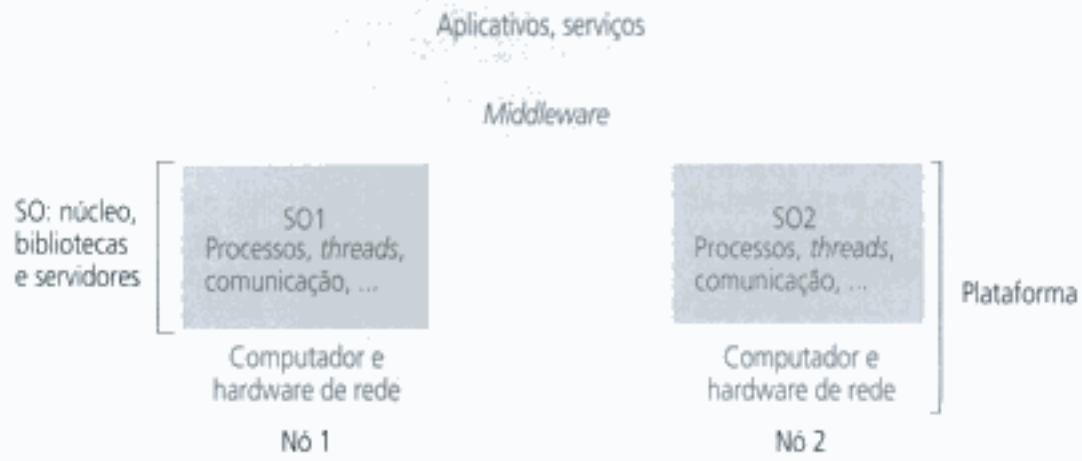


Figura 6.1 **Camadas de sistema.**

Processamento concorrente: os clientes podem compartilhar recursos e acessá-los concorrentemente. Os gerenciadores de recurso são responsáveis pela transparência da concorrência.

Os clientes acessam recursos fazendo, por exemplo, invocações a métodos remotos em um objeto servidor ou chamadas de sistema em um núcleo. Denominamos a maneira de acessar um recurso encapsulado de *mecanismo de invocação*, não importa como isso seja implementado. Uma combinação de bibliotecas, núcleos e servidores pode ser empregada para realizar as seguintes tarefas relacionadas à invocação:

Comunicação: parâmetros e resultados de operação precisam ser passados por gerenciadores de recursos, por meio de uma rede ou dentro de um computador.

Escalonamento: quando uma operação é invocada, seu processamento deve ser agendado dentro do núcleo ou do servidor.

A Figura 6.2 mostra as funcionalidades básicas de um SO com que vamos nos preocupar: gerenciamento de processos e *threads*, gerenciamento de memória e comunicação entre processos no mesmo computador (as divisões horizontais na figura denotam dependências). O núcleo fornece grande parte dessa funcionalidade – toda ela, no caso de alguns sistemas operacionais.

Figura 6.2 **Funcionalidades básicas de um SO.**

Multiprocessadores de memória compartilhada ◊ Os computadores multiprocessadores de memória compartilhada vêm equipados com vários processadores que compartilham um ou mais módulos de memória (RAM). Os processadores também podem ter sua própria memória privativa. Os computadores multiprocessadores podem ser construídos de diversas formas [Stone 1993]. Os multiprocessadores mais simples e mais baratos são construídos por meio da incorporação de uma placa contendo alguns processadores (2–8) em um computador pessoal.

Na *arquitetura de processamento simétrico* comum, cada processador executa o mesmo núcleo e os núcleos desempenham papéis equivalentes no gerenciamento dos recursos de hardware. Os núcleos compartilham suas principais estruturas de dados, como a fila de *threads* prontas para execução, mas alguns de seus dados de trabalho são privativos. Cada processador pode, simultaneamente, executar uma *thread*, acessando dados na memória compartilhada, que pode ser privativa (protegida pelo hardware) ou compartilhada com as demais *threads*.

Os multiprocessadores podem ser usados para muitas tarefas de computação de alto desempenho. Nos sistemas distribuídos, eles são particularmente úteis para a implementação de servidores, pois o servidor pode executar um único programa com várias *threads*, tratando simultaneamente requisições de diversos clientes – por exemplo, fornecendo acesso a um banco de dados compartilhado (veja a Seção 6.4).

O software do SO é projetado para ser portável entre arquiteturas de computador. Isso significa que a maior parte dele é codificada em uma linguagem de alto nível, como C, C++ ou Modula-3, e que seus recursos são dispostos em camadas para que os componentes dependentes de máquina sejam reduzidos a uma camada inferior mínima. Alguns núcleos podem ser executados em multiprocessadores de memória compartilhada, os quais estão descritos no quadro acima.

Os componentes básicos do SO são os seguintes:

Gerenciador de processos: trata da criação de processos e das operações neles executadas. Um processo é uma unidade de gerenciamento de recursos, incluindo um espaço de endereçamento e uma ou mais *threads*.

Gerenciador de threads: serve para a criação de *threads*, sincronização e escalonamento. As *threads* são fluxos de execução associados aos processos e serão descritas na Seção 6.4.

Gerenciador de comunicação: trata da comunicação entre *threads* associadas a diferentes processos no mesmo computador. Alguns núcleos também suportam comunicação entre *threads* de processos remotos. Outros necessitam de serviços adicionais para prover comunicação externa com outras máquinas. A Seção 6.5 discutirá o projeto de sistemas de comunicação.

Gerenciador de memória: trata do gerenciamento da memória física e virtual. As Seções 6.4 e 6.5 descrevem a utilização das técnicas de gerenciamento de memória para cópia e compartilhamento eficiente dos dados.

Supervisor: trata do envio de interrupções, das capturas das chamadas de sistema e de exceções; do controle da unidade de gerenciamento de memória e de suas caches; do processador e da unidade em ponto flutuante. No Windows, ele é conhecido como camada de abstração de hardware, ou HAL (*Hardware Abstraction Layer*). O leitor pode consultar Bacon [2002] e Tanenbaum [2001] para uma descrição mais completa sobre os aspectos do núcleo dependentes do computador.

6.3 Proteção

Dissemos anteriormente que os recursos exigem proteção contra acessos ilegítimos. Note que a ameaça à integridade de um sistema não é proveniente apenas de código construído de forma maliciosa. Um código benigno, contendo um erro ou um comportamento imprevisto, pode fazer com que parte do sistema aja incorretamente.

Para entender o que queremos dizer com "acesso ilegítimo" a um recurso, considere um arquivo. Vamos supor, para esta explicação, que os arquivos abertos tenham apenas duas operações, *leitura* e *escrita*. A proteção do arquivo consiste em dois subproblemas. O primeiro é garantir que cada uma das duas operações do arquivo possa ser executada somente pelos clientes que tenham o direito de executá-la. Por exemplo, Smith, a quem pertence o arquivo, tem direitos de *leitura* e *escrita* nele. Jones só pode executar a operação de *leitura*. Um acesso ilegítimo seria se Jones conseguisse de algum modo executar uma operação de *escrita* no arquivo. Uma solução completa para esse subproblema de proteção de recurso em um sistema distribuído exige técnicas de criptografia e vamos deixá-la para o Capítulo 7.

O outro tipo de acesso ilegítimo, que vamos tratar aqui, é quando um cliente mal comportado consegue realizar operações diferentes daquelas disponibilizadas por um recurso. Em nosso exemplo, isso aconteceria se Smith, ou Jones, conseguissem de algum modo executar uma operação que não fosse nem de *leitura* nem de *escrita*. Suponha, por exemplo, que Smith conseguisse acessar diretamente a variável de ponteiro do arquivo. Ele poderia, então, construir uma operação *setFilePointerRandomly*, que configuraria o ponteiro do arquivo com um número aleatório. Como essa função tem uma utilidade praticamente nula, ela nunca faria parte do conjunto de funções disponibilizadas por um sistema de arquivos.

Podemos proteger os recursos contra invocações ilegítimas, como a operação *setFilePointerRandomly*. Uma maneira é usar uma linguagem fortemente tipada, como Java ou Modula-3. Uma linguagem fortemente tipada é desenvolvida de tal forma que nenhum módulo pode acessar um módulo de destino, a menos que tenha uma referência válida para ele – ou seja, não é permitido criar um ponteiro e acessá-lo, como seria viável em C ou C++. Além disso, nesses casos, a referência ao módulo de destino só pode ser usada para realizar as invocações, a métodos ou procedimentos, que o programador do destino tornou disponíveis. Em outras palavras, não é possível alterar as variáveis do destino arbitrariamente. Em contraste, em C++, o programador pode definir um ponteiro, convertê-lo para um tipo qualquer (*casting*) e, assim, realizar quaisquer invocações e acessos.

Também é possível empregar suporte de hardware para proteger os módulos uns dos outros, independentemente da linguagem em que eles foram escritos. Para explorar essa possibilidade em um computador de propósito geral, é necessário contar com o auxílio do núcleo do sistema operacional.

Núcleo e proteção ◊ O núcleo é um programa diferenciado pelo fato de que está sempre sendo ativado e seu código é executado com privilégios de acesso completos aos recursos físicos presentes em seu computador *host*. Em particular, ele pode controlar a unidade de gerenciamento de memória e configurar os registradores do processador de modo que nenhum outro código possa acessar os recursos físicos da máquina, exceto de formas consideradas como aceitáveis.

A maioria dos processadores tem, em hardware, um registrador de modo cuja configuração determina se instruções privilegiadas podem ser executadas, como aquelas usadas para determinar quais tabelas de proteção são correntemente empregadas pela unidade de gerenciamento de memória. O processo núcleo é executado com o processador no modo *supervisor* (privilegiado) e o núcleo providencia para que os outros processos sejam executados no modo *usuário* (não privilegiado).

O núcleo também configura *espaços de endereçamento* para proteger a si mesmo, e a outros processos, dos acessos de um processo anômalo e para fornecer aos processos uma área de memória virtual. Um espaço de endereçamento é um conjunto de intervalos de posições de memória virtual em cada um dos quais se aplica uma combinação específica de direitos de acesso à memória como, por exemplo, somente leitura ou gravação. Um processo não pode acessar posições de memória fora do seu espaço de endereçamento. Os termos *processo de usuário* ou *processo em nível de usuário* são normalmente usados para descrever aquele processo que é executado no modo usuário e tem um espaço de endereçamento em nível de usuário, isto é, possui direitos de acesso à memória restritos, comparado ao espaço de endereçamento do núcleo.

Quando um processo executa pode alternar entre espaço de endereçamento de usuário e espaço de endereçamento do núcleo, dependendo se o código em execução está vinculado a aplicação ou ao núcleo. O processo chaveia do espaço de endereçamento de usuário para o espaço de endereçamento do núcleo por meio de exceções, como uma interrupção ou uma *chamada do sistema* – o mecanismo

de invocação de recursos do núcleo. Uma chamada do sistema é implementada por uma instrução de máquina do tipo *TRAP*, que coloca o processador no modo supervisor e passa para o espaço de endereçamento do núcleo. Quando a instrução *TRAP* é executada, assim como acontece com qualquer tipo de exceção, o hardware obriga o processador a executar uma função de tratamento de exceção, fornecida pelo núcleo, para que nenhum processo possa ganhar o controle ilícito do hardware.

Os programas pagam um preço pela proteção. O chaveamento entre os espaços de endereçamento pode ocupar muitos ciclos do processador e uma chamada do sistema é uma operação mais dispendiosa do que uma simples chamada de procedimento ou de método. Veremos agora como essas penalidades entram nos custos da invocação.

6.4 Processos e *threads*

Nos anos 80, foi descoberto que a noção tradicional do sistema operacional, de um processo que executa um único fluxo de execução, era diferente dos requisitos dos sistemas distribuídos – e também dos aplicativos mais sofisticados que utilizam um único processador, mas que exigiam concorrência de atividades interna. O problema, conforme mostraremos, é que o processo tradicional torna complicado e dispendioso o compartilhamento de recursos entre atividades relacionadas.

A solução encontrada foi aprimorar a noção de processo, para que ele pudesse ser associado a múltiplas atividades. Atualmente, um processo consiste em um ambiente de execução, junto com uma ou mais *threads*. Uma *thread* é a abstração do sistema operacional de uma atividade (o termo é derivado da frase "fio (*thread*) de execução"). O *ambiente de execução* é a unidade de gerenciamento de recursos: um conjunto de recursos locais gerenciados pelo núcleo, aos quais suas *threads* têm acesso. Um ambiente de execução consiste principalmente em:

- um espaço de endereçamento;
- recursos de sincronização e comunicação entre *threads*, como semáforos e interfaces de comunicação (por exemplo, soquetes);
- recursos de nível mais alto, como arquivos e janelas abertas.

Normalmente, a criação e o gerenciamento de ambientes de execução são operações dispendiosas, mas várias *threads* podem compartilhá-los – isto é, elas podem compartilhar todos os recursos disponíveis dentro deles. Em outras palavras, um ambiente de execução representa um domínio de proteção no qual suas *threads* são executadas.

As *threads* podem ser criadas e destruídas dinamicamente, conforme necessário. O objetivo principal em se ter múltiplas *threads* de execução é maximizar o grau de execução concorrente entre operações, permitindo assim a sobreposição da computação com operações de entrada e saída, e possibilitando a execução simultânea de atividades em máquinas do tipo multiprocessadores. Isso pode ser particularmente útil para servidores, onde o processamento simultâneo de requisições de clientes pode reduzir a tendência de criação de gargalos de processamento. Por exemplo, uma *thread* pode processar a requisição de um cliente, enquanto uma segunda *thread*, que está atendendo outra requisição, espera que um acesso ao disco termine.

Um ambiente de execução fornece proteção contra as *threads* que estão fora dele, de modo que os dados e outros recursos nele contidos são, por padrão, inacessíveis pelas *threads* residentes em outros ambientes de execução. Entretanto, certos núcleos permitem o compartilhamento controlado de recursos, como a memória física, entre ambientes de execução residentes no mesmo computador.

Como muitos sistemas operacionais mais antigos só permitem uma *thread* por processo, às vezes vamos usar o termo *processo multi-threaded* para dar ênfase aos que possuem mais de uma *thread*. Infelizmente, para confundir tudo, em alguns modelos de programação e sistemas operacionais, o termo "processo" significa executar uma *thread*. O leitor pode encontrar na literatura os termos *processo pesado*, para se referenciar aos ambientes de execução (processo), e *processo leve* para as *threads*. Veja o quadro a seguir para uma analogia descrevendo *threads* e ambientes de execução.

Uma analogia para *threads* e processos ◊ A maneira memorável, talvez ligeiramente repugnante, de pensar sobre os conceitos de *threads* e ambientes de execução apareceu no grupo da USENET *comp.os.mach* e é atribuído a Chris Lloyd. Um ambiente de execução consiste em um jarro tampado com ar e o alimento dentro dele. Inicialmente, há uma mosca – uma *thread* – no jarro. Essa mosca, assim como suas próprias descendentes, pode produzir outras moscas e matá-las. Qualquer mosca pode consumir qualquer recurso (ar ou alimento) no jarro. As moscas podem ser programadas para entrarem em uma fila, de maneira ordenada, para consumir recursos. Se não tiverem essa disciplina, elas podem se chocar umas com as outras dentro do jarro – isto é, colidir e produzir resultados imprevisíveis ao tentarem consumir os mesmos recursos de maneira não controlada. As moscas podem se comunicar com (enviar mensagens para) moscas de outros jarros, mas nenhuma pode escapar do jarro e nenhuma mosca de fora pode entrar nele. Nessa visão, um processo padrão UNIX é um único jarro com uma única mosca estéril dentro dele.

6.4.1 Espaços de endereçamento

Um espaço de endereçamento é a unidade de gerenciamento da memória virtual de um processo. Normalmente, o espaço de endereçamento possui uma grande capacidade (2^{32} bytes e, às vezes, até 2^{64} bytes) e consiste em uma ou mais *regiões*, separadas por áreas não acessíveis de memória virtual. Uma região (Figura 6.3) é uma área contígua de memória virtual que é acessível para as *threads* do processo que a possui. As regiões não se sobrepõem. Note que fazemos distinção entre as regiões e seus conteúdos. Cada região é especificada pelas seguintes propriedades:

- sua extensão (menor endereço virtual e tamanho);
- permissões de leitura/gravação/execução para as *threads* do processo;
- pode crescer em ambas as direções de endereçamento (para baixo e para cima)

Consideraremos para o restante de capítulo que o modelo de gerência de memória é baseado em paginação e não em segmentação. As regiões podem se sobrepor, caso aumentem de tamanho. Para possibilitar seu crescimento, são deixadas lacunas entre as regiões. Essa representação do espaço de endereços como um conjunto esparso de regiões desmembradas é uma generalização do espaço de

Figura 6.3 Espaço de endereçamento.

endereçamento do UNIX, que possui três regiões: uma região de texto fixa, não modificável, contendo código de programa; uma de região *heap*, parte da qual é inicializada por valores armazenados no arquivo binário do programa e que pode ser estendida para endereços virtuais mais altos; e uma região pilha, que pode ser estendida para endereços virtuais mais baixos.

O uso de um número indefinido de regiões é motivado por diversos fatores. Um deles é a necessidade de suportar uma pilha separada para cada *thread*. A alocação de uma região de pilha separada para cada *thread* torna possível detectar tentativas de ultrapassar seus limites e controlar seu crescimento. A memória virtual não alocada fica além de cada região de pilha e tentativas de acessá-la causarão uma exceção (um erro de acesso inválido a página). A alternativa é alocar pilhas para *threads* no *heap*, mas então ficará difícil detectar quando uma *thread* tiver ultrapassado seu limite de pilha.

Outra motivação é permitir que arquivos em geral – e não apenas as seções de texto e dados dos arquivos binários – sejam mapeados no espaço de endereçamento. Um *arquivo mapeado* é aquele que pode ser acessado como um array de bytes na memória. O sistema de memória virtual garante que os acessos feitos na memória se reflitam no sistema de arquivos. A Seção CDK3-18.6 (no endereço [www.cdk4.net/oss/mach]) descreve como o núcleo Mach estende a abstração de memória virtual para que as regiões possam corresponder a "objetos de memória" arbitrários – e não apenas a arquivos.

A necessidade de compartilhar memória entre processos, ou entre processos e o núcleo, é outro fator que leva a regiões extras no espaço de endereçamento. Uma *região de memória compartilhada* (ou, resumidamente, *região compartilhada*) é aquela em que uma porção de memória física é mapeada para uma ou mais regiões pertencentes a vários espaços de endereçamento. Portanto, os processos acessam as mesmas posições de memória nas regiões compartilhadas, enquanto suas regiões não compartilhadas permanecem protegidas. Os usos de regiões compartilhadas incluem os seguintes:

Bibliotecas: o código de uma biblioteca pode ser muito grande e desperdiçaria uma memória considerável se fosse carregada separadamente para cada processo que a usasse. Em vez disso, uma única cópia da biblioteca pode ser compartilhada, sendo mapeada como uma região nos espaços de endereçamentos dos processos que a utilizam.

Núcleo: freqüentemente, o código e os dados do núcleo são mapeados em uma mesma região (compartilhada) em cada espaço de endereçamento. Quando um processo faz uma chamada de sistema, ou quando ocorre uma exceção, não há necessidade de trocar para um novo conjunto de mapeamentos de endereço.

Compartilhamento de dados e comunicação: dois processos, ou um processo e o núcleo, talvez precisem compartilhar dados para colaborarem em alguma tarefa. Pode ser consideravelmente mais eficiente os dados serem compartilhados sendo mapeados como regiões nos dois espaços de endereçamentos do que passados em mensagens entre eles. O uso de compartilhamento de região para comunicação será descrito na Seção 6.5.

6.4.2 Criação de um novo processo

Tradicionalmente, a criação de um novo processo é uma operação indivisível fornecida pelo sistema operacional. Por exemplo, a chamada de sistema *fork* do UNIX cria um processo com um ambiente de execução copiado do processo que efetuou a chamada (exceto pelo valor de retorno de *fork*). A chamada de sistema *exec* faz com que o processo que a realiza carregue um novo código e o execute.

Para um sistema distribuído, o projeto do mecanismo de criação de processos precisa levar em conta a utilização de vários computadores; conseqüentemente, a infra-estrutura de suporte a processos é dividida em distintos serviços de sistema.

A criação de um novo processo pode ser separada em dois aspectos independentes:

- A escolha de um *host* de destino. Por exemplo, o *host* pode ser escolhido dentre os nós de um agrupamento (*cluster*) de computadores atuando como um servidor de computação (veja o quadro a seguir).
- A criação de um ambiente de execução (e de uma *thread* inicial dentro dele).

Escolha do *host* ◊ A escolha do nó em que o novo processo residirá – a decisão de alocação do processo – é uma questão de emprego de políticas. Em geral, as políticas de alocação de processos variam

Agrupamentos (*clusters*) ◊ Um agrupamento, ou aglomerado, é um conjunto de computadores convencionais (às vezes milhares deles) interligados por uma rede de comunicação de alta velocidade, como uma Gigabit Ethernet, comutada. Os computadores individuais podem ser PCs, estações de trabalho padrão, placas de processador montadas sobre um *rack*; eles ainda podem ser monoprocessadores ou multiprocessadores. Uma aplicação de *clusters* é o fornecimento de serviços de alta disponibilidade e flexíveis – como os mecanismos de busca fornecidos para os usuários em toda a Internet – pela replicação ou divisão do processamento e do estado do servidor em todos processadores do *cluster* [Fox *et al.* 1997]. Os *clusters* também são usados para executar programas paralelos [Anderson *et al.* 1995, now.cs.berkeley.edu, TFCC].

desde sempre executar os novos processos na estação de trabalho de seus criadores até o balanceamento da carga de processamento entre um conjunto de computadores. Eager *et al.* [1986] distinguem as seguintes categorias de políticas para o balanceamento de carga.

A *política de transferência* determina se um novo processo será criado localmente ou remotamente. Isso pode depender, por exemplo, do nó local estar pouco ou muito carregado.

A *política de localização* determina qual nó deve receber um novo processo selecionado para transferência. Essa decisão pode depender das cargas relativas dos nós, de suas arquiteturas de máquina e dos recursos especializados que eles possam ter. O sistema V [Cheriton 1984] e o Sprite [Douglis e Ousterhout 1991] fornecem comandos para os usuários executarem um programa em uma estação de trabalho ociosa (freqüentemente, existem muitas delas em dado momento), escolhida pelo sistema operacional. No sistema Amoeba [Tanenbaum *et al.* 1990], o *servidor de execução* escolhe um *host* para cada processo a partir de um conjunto de processadores compartilhados. Em todos os casos, a escolha do *host* de destino é transparente para o programador e para o usuário. Entretanto, programas desenvolvidos para explorar paralelismo explícito, ou para considerar aspectos de tolerância a falhas, podem exigir uma maneira de especificar a localização do processo.

As políticas de localização de processo podem ser *estáticas* ou *adaptativas*. As primeiras operam sem considerar o estado corrente do sistema, embora sejam projetadas de acordo com as características de longo prazo esperadas pelo sistema. Elas são baseadas em uma análise matemática destinada a otimizar um parâmetro, como a quantidade de processos executados por unidade de tempo. Elas podem ser determinísticas ("o nó A sempre deve transferir processos para o nó B") ou probabilísticas ("o nó A deve transferir processos para qualquer um dos nós B–E, aleatoriamente"). Por outro lado, as políticas adaptativas aplicam heurísticas para tomar suas decisões de alocação com base em fatores de tempo de execução imprevisíveis, como a medida da carga em cada nó.

Os sistemas de balanceamento de carga podem ser centralizados, hierárquicos ou descentralizados. No primeiro caso existe um único componente gerenciador de carga e, no segundo, existem vários, organizados em uma estrutura em árvore. Os gerenciadores de carga reúnem informações sobre os nós e as utilizam para alocar novos processos nos nós. Nos sistemas hierárquicos, os gerenciadores tomam as decisões de alocação de processo no nível da árvore mais baixo possível, mas, sob certas condições de carga, eles podem transferir processos uns para os outros por meio de um nó antecessor comum. Em um sistema de balanceamento de carga descentralizado, os nós trocam informações diretamente uns com os outros, para tomar decisões de alocação. O sistema Spawn [Waldspurger *et al.* 1992], por exemplo, considera que os nós são "compradores" e "vendedores" de recursos computacionais e os organiza em uma "economia de mercado" (descentralizada).

Nos algoritmos de balanceamento de carga *iniciado pela origem*, o nó que faz a criação de um novo processo é responsável por iniciar a decisão de transferência. Normalmente, ele inicia uma transferência quando sua própria carga ultrapassa um limite. Em contraste, nos algoritmos *iniciados pelo destino*, um nó cuja carga está abaixo de determinado limite anuncia sua existência para outros nós, para que os nós relativamente carregados transfiram trabalho para ele.

Os sistemas de balanceamento de carga que oferecem suporte a *migração* podem transferir carga a qualquer momento e não apenas quando um novo processo é criado. Eles usam um mecanismo chamado *migração de processo*: a transferência de um processo em execução de um nó para outro. Milojicic *et al.* [1999] fornecem um conjunto de artigos sobre migração de processo e outros tipos de

mobilidade. Embora vários mecanismos de migração de processo tenham sido construídos, eles não são amplamente usados. Isso principalmente devido ao seu custo computacional e à grande dificuldade de extrair o estado de um processo de dentro do núcleo para movê-lo para outro nó.

Eager *et al.* [1986] estudaram três estratégias de balanceamento de carga e concluíram que a simplicidade é uma propriedade importante para qualquer esquema. Pois o custo de coletar informações de carga e tomar decisões de balanceamento pode comprometer a vantagem de usá-lo.

Criação de um novo ambiente de execução ◊ Uma vez que o computador *host* tenha sido selecionado, a criação de um novo processo exige um ambiente de execução composto por um espaço de endereçamento com conteúdos inicializados (e talvez outros recursos, como arquivos padrão abertos).

Existem duas estratégias para definir e inicializar o espaço de endereçamento de um processo recentemente criado. A primeira estratégia é usada onde o espaço de endereçamento tem um formato definido estaticamente. Por exemplo, ele pode conter apenas uma região de texto, uma região de *heap* e uma região de pilha. Nesse caso, as regiões do espaço de endereçamento são criadas a partir de uma lista que especifica seus respectivos tamanhos. As regiões do espaço de endereçamento são inicializadas a partir de um arquivo executável, ou preenchidas com valores zero, conforme apropriado.

Como alternativa, o espaço de endereçamento pode ser definido com relação a um ambiente de execução existente. No caso da semântica *fork* do UNIX, por exemplo, o processo filho, recentemente criado, compartilha fisicamente a região de texto do processo pai (seu criador) e tem cópias das regiões de *heap* e pilha. Esse esquema foi generalizado de modo que cada região do processo pai possa ser herdada (ou omitida) pelo processo filho. Uma região herdada pode ser compartilhada, ou logicamente copiada, da região do pai. Quando pai e filho compartilham uma região, os quadros (*frames*) – unidades de memória física correspondentes às páginas de memória virtual – pertencentes à região do pai são mapeados simultaneamente na região correspondente do filho.

Os núcleos Mach [Accetta *et al.* 1986] e Chorus [Rozier *et al.* 1988, 1990], por exemplo, aplicam uma otimização chamada *cópia na escrita* (*copy-on-write*), quando uma região herdada é copiada do pai. A região é copiada, mas nenhuma cópia física ocorre por padrão. Os quadros que compõem a região herdada são compartilhados entre os dois espaços de endereçamento até que um ou outro processo tente modificá-la. Nesse ponto, o quadro é fisicamente copiado de uma região para outra.

A cópia na escrita é uma técnica genérica – por exemplo, ela também é usada para mensagens grandes; portanto, passaremos algum tempo explicando seu funcionamento. Vamos acompanhar um exemplo das regiões *RA* e *RB*, cuja memória é compartilhada com base em *cópia na escrita* entre dois processos, *A* e *B* (Figura 6.4). Por questões de clareza, vamos supor que o processo *A* configura a região *RA* para ser uma copiada por seu filho, o processo *B*, e que a região *RB* foi, portanto, criada no processo *B*.

Por simplicidade, supomos que as páginas pertencentes à região A residem na memória. Inicialmente, todos os quadros associados às regiões são compartilhados entre as tabelas de páginas dos dois processos. As páginas são inicialmente protegidas contra escrita, mesmo que possam pertencer a regiões que sejam logicamente graváveis. Se uma *thread* de um dos dois processos tentar modificar os dados, será gerada uma exceção de hardware chamada *erro de acesso inválido a página*. Digamos que o processo *B* tentou a escrita. A rotina de *erro de acesso inválido a página* aloca um novo quadro para o processo *B* e copia nele os dados do quadro original, byte por byte. O número de quadro antigo é substituído pelo novo número de quadro na tabela de páginas de um processo – não importa qual deles – e o número de quadro antigo é deixado na outra tabela de páginas. Cada uma das duas páginas correspondentes nos processos *A* e *B*, mapeados em quadros da memória, passam a ter autorização para escrita (modificação). Depois que tudo isso tiver acontecido, a instrução de escrita do processo *B* poderá prosseguir.

6.4.3 *Threads*

O próximo aspecto importante de um processo a considerar com mais detalhes são suas *threads*. Esta subseção examina as vantagens de permitir que os processos clientes e servidores possuam mais de uma *thread*. Então, ela discute a programação com *threads*, usando *threads* Java como um estudo de caso, e termina com projetos alternativos para implementação de *threads*.

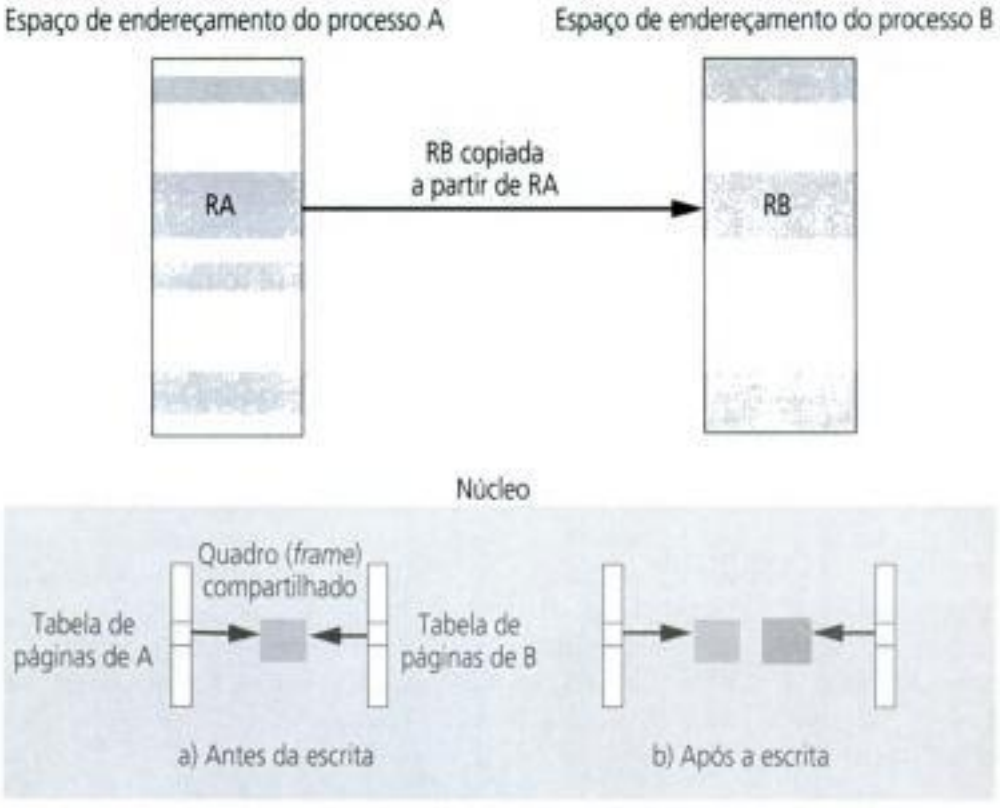


Figura 6.4 Cópia na escrita.

Considere o servidor mostrado na Figura 6.5 (veremos o cliente em breve). O servidor tem um pool de uma ou mais *threads*, cada uma das quais retira um pedido de uma fila de requisições recebidas e o processa. Não vamos nos preocupar, por enquanto, com o modo como os pedidos são recebidos e enfileirados para as *threads*. Além disso, por simplicidade, supomos que cada *thread* executa a mesma rotina para processar os pedidos. Vamos supor que cada pedido tenha, em média, um atraso de 2 milissegundos de processamento e, 8 milissegundos de E/S (entrada/saída), quando o servidor lê um disco (não há utilização de cache). Vamos supor ainda, por enquanto, que o servidor é executado em um computador com um único processador.

Considere a *taxa de rendimento* (*throughput*) máxima do servidor, medida em pedidos de clientes manipulados por segundo, para diferentes números de *threads*. Se uma única *thread* tiver que realizar todo o processamento, então o tempo gasto para manipular qualquer pedido será, em média, 2 + 8 =

Figura 6.5 Cliente e servidor com *threads*.

10 milissegundos; portanto, esse servidor pode manipular 100 pedidos de clientes por segundo. Todas as novas mensagens de pedido que chegam enquanto o servidor está manipulando um pedido são enfileiradas.

Agora, considere o que acontece se o pool do servidor contiver duas *threads*. Supomos que as *threads* são escalonadas independentemente – isto é, uma *thread* pode ser executada quando a outra for bloqueada para E/S. Então, a *thread* número dois pode processar um segundo pedido, enquanto a *thread* número um está bloqueada e vice-versa. Isso aumenta a taxa de rendimento. Infelizmente, em nosso exemplo, por acessarem uma única unidade de disco, ambas *threads* podem ficar bloqueadas a espera da conclusão de E/S. Se todos os pedidos de disco forem dispostos em série e demorarem 8 milissegundos cada, então o desempenho de saída máximo será de 1000/8 = 125 pedidos por segundo.

Suponha, agora, que seja introduzida uma cache de blocos de disco. O servidor mantém os dados que lê em buffers no seu espaço de endereçamento; uma *thread*, ao ler dados, primeiro examina a cache e, se os encontrar lá, evita o acesso ao disco. Se possuir uma taxa de acerto de 75%, o tempo de E/S médio por pedido será reduzido a (0,75 × 0 + 0,25 × 8) = 2 milissegundos e a taxa de rendimento máxima aumentará para 500 pedidos por segundo. Mas se o tempo de processamento médio para um pedido aumentou para 2,5 milissegundos, como resultado do uso de cache (leva tempo para pesquisar dados colocados na cache em cada operação), então esse valor não pode ser atingido. O servidor, limitado pelo tempo de processamento, pode agora manipular no máximo 1000/2,5 = 400 pedidos por segundo.

A taxa de rendimento pode ser aumentada, usando-se um multiprocessador de memória compartilhada para diminuir o gargalo de processamento. Um processo *multi-threaded* é mapeado naturalmente em um multiprocessador de memória compartilhada. O ambiente de execução pode ser implementado na memória compartilhada e as múltiplas *threads* podem ser programadas para executar nos múltiplos processadores. Considere agora o caso em que nosso exemplo de servidor é executado em um multiprocessador com dois processadores. Dado que as *threads* podem ser escalonadas independentemente nos diferentes processadores, então, até duas *threads* podem processar pedidos em paralelo. Como exercício, o leitor deve verificar que duas *threads* podem processar 444 pedidos por segundo, e três ou mais *threads*, limitadas pelo tempo de E/S, podem processar 500 pedidos por segundo.

Arquiteturas de servidores *multi-threadeds* ◊ Descrevemos como o uso de várias *threads* permite aos servidores maximizarem sua taxa de rendimento, medido como o número de pedidos processados por segundo. Para descrevermos as diversas formas de mapear pedidos em *threads* dentro de um servidor, resumimos a narrativa de Schmidt [1998], que descreve as arquiteturas baseadas em *threads* de várias implementações do ORB (*Object Request Broker*) do CORBA. Os ORBs processam os pedidos que chegam através de mensagens em soquetes TCP. Essas arquiteturas baseadas em *threads* são relevantes para muitos tipos de servidor, independentemente do CORBA ser usado.

A Figura 6.5 mostra uma das possíveis arquiteturas baseadas em *threads*, a *arquitetura do pool de trabalhadores*. Em sua forma mais simples, ao ser inicializado, o servidor cria um pool fixo de *threads* "trabalhadores" para processar os pedidos. O módulo identificado como "recepção e enfileiramento", na Figura 6.5, é normalmente implementado por uma *thread* de "E/S", que recebe pedidos de um conjunto de soquetes, ou portas, e os coloca em uma fila de pedidos para serem recuperados pelos trabalhadores.

Às vezes, há necessidade de tratar dos pedidos com prioridades distintas. Por exemplo, um servidor web corporativo poderia dar prioridade ao processamento dos pedidos de acordo com a classe de cliente da qual o pedido deriva [Bhatti e Friedrich 1999]. Podemos tratar as prioridades de pedidos introduzindo múltiplas filas na arquitetura de pool de trabalhadores, de modo que as *threads* trabalhadores percorram as filas na ordem decrescente de prioridade. Uma desvantagem dessa arquitetura é sua falta de flexibilidade: conforme vimos em nosso exemplo, o número de *threads* trabalhadores no pool pode ser pequeno demais para tratar adequadamente da taxa de chegada de pedidos. Outra desvantagem é a grande quantidade de chaveamentos entre as *threads* de E/S e trabalhadores, pois eles manipulam uma fila compartilhada.

Na *arquitetura thread por pedido* (Figura 6.6a), a *thread* de E/S gera uma nova *thread* trabalhador para cada pedido e esse trabalhador se autoterminará quando tiver processado o pedido. Essa arquitetura tem a vantagem de que as *threads* não disputam uma fila compartilhada e a taxa de rendimento é potencialmente maximizada, pois a *thread* de E/S pode criar tantos trabalhadores quantos forem os pedidos pendentes. Sua desvantagem é a sobrecarga das operações de criação e destruição de *threads*.

a. *Thread* por pedido b. *Thread* por conexão c. *Thread* por objeto

Figura 6.6 Arquiteturas baseadas em *threads* (veja também a Figura 6.5).

A *arquitetura de thread por conexão* (Figura 6.6b) associa uma *thread* a cada conexão. O servidor cria uma nova *thread* trabalhador quando um cliente estabelece uma conexão e destrói a *thread* quando o cliente fecha a conexão. Nesse meio-tempo, o cliente pode fazer vários pedidos pela conexão, destinados a um ou mais objetos remotos. A *arquitetura de thread por objeto* (Figura 6.6c) associa uma *thread* a cada objeto remoto. Uma *thread* de E/S recebe pedidos e os enfileira para os trabalhadores, mas desta vez há uma fila por objeto.

Em cada uma dessas duas últimas arquiteturas, o servidor se beneficia das menores sobrecargas de gerenciamento de *thread*, se comparadas à arquitetura de *thread* por pedido. Sua desvantagem é que os clientes podem sofrer atrasos quando uma *thread* trabalhador tem vários pedidos pendentes, mas outra *thread* não tem trabalho a fazer.

Schmidt [1998] descreve variações dessas arquiteturas, assim como combinações delas, e discute suas vantagens e desvantagens com mais detalhes. A Seção 6.5 descreverá um modelo de uso de *threads* diferente, no contexto das invocações dentro de uma única máquina, no qual as *threads* clientes têm acesso ao espaço de endereçamento do servidor.

Threads em clientes ◊ As *threads* podem ser úteis para os clientes, assim como para os servidores. A Figura 6.5 também mostra um processo cliente com duas *threads*. A primeira *thread* gera resultados a serem passados para um servidor por meio de invocação a método remoto, mas não exige uma resposta. As invocações a métodos remotos normalmente bloqueiam o chamador, mesmo quando não há rigorosamente nenhuma necessidade de esperar. Esse processo cliente pode incorporar uma segunda *thread*, a qual realiza as invocações a métodos remotos e bloqueia, enquanto a primeira *thread* é capaz de continuar calculando mais resultados. A primeira *thread* coloca seus resultados em buffers, os quais são esvaziados pela segunda *thread*. Ela só é bloqueada quando todos os buffers estiverem cheios.

O caso dos clientes *multi-threaded* também fica evidente no exemplo de navegadores web. Os usuários sentem demoras substanciais enquanto páginas são procuradas; portanto, é fundamental que os navegadores manipulem vários pedidos de páginas web paralelamente.

Threads versus múltiplos processos ◊ A partir dos exemplos anteriores, podemos ver a utilidade das *threads*, as quais permitem que a computação seja sobreposta com E/S e, no caso de um multiprocessador, com outra computação. Entretanto, o leitor pode ter notado que a mesma sobreposição poderia ser obtida pelo uso de vários processos *single-threadeds*. Por que, então, o modelo do processo *multi-threaded* deve ser preferido? A resposta é dupla: as *threads* são computacionalmente mais baratas de serem criadas e gerenciadas do que os processos, e o compartilhamento de recursos pode ser obtido de forma mais eficiente entre *threads* do que entre processos, pois elas compartilham um ambiente de execução.

A Figura 6.7 mostra alguns dos principais componentes de estado que devem ser mantidos para ambientes de execução e para *threads*, respectivamente. Um ambiente de execução tem um espaço de endereçamento; interfaces de comunicação, como os soquetes; recursos de mais alto nível, como arquivos abertos e objetos de sincronização de *threads*, como os semáforos; e por fim, ele também lista as *threads* que possui. Uma *thread* tem uma prioridade de escalonamento, um estado de execução (como *BLOCKED* ou *RUNNABLE*), cópia dos valores dos registradores internos do processador,

Ambiente de execução	*Thread*
Tabelas de espaço de endereçamento	Registradores internos do processador salvos
Interfaces de comunicação, arquivos abertos	Prioridade e estado da execução (como *BLOCKED*)
Semáforos, outros objetos de sincronização	Informações do tratamento da interrupção de software
Lista de identificadores de *thread*	Identificador do ambiente de execução
Páginas do espaço de endereçamento residentes na memória; entradas de cache em hardware	

Figura 6.7 Estados associados aos ambientes de execução e as *threads*.

quando a *thread* está no estado *BLOCKED*, e o estado da *thread* relativo à execução de uma *interrupção de software*. Uma *interrupção de software* é um evento que faz uma *thread* ser interrompida (semelhante ao caso de uma interrupção de hardware). Se a *thread* tiver um tratador de interrupção, o controle será transferido para ela. Os sinais do UNIX são exemplos de interrupções de software.

A figura mostra que um ambiente de execução e as *threads* pertencentes a ele são associados às páginas pertencentes ao espaço de endereçamento mantido na memória principal, e os dados e instruções são mantidos em caches em hardware.

Podemos fazer um resumo da comparação entre processos e *threads*, como segue:

- Criar uma nova *thread* dentro de um processo existente é computacionalmente menos oneroso do que criar um processo.
- O chaveamento para uma *thread* diferente dentro de um mesmo processo é menos oneroso do que chavear entre *threads* pertencentes a processos diferentes.
- As *threads* dentro de um processo podem compartilhar dados e outros recursos conveniente e eficientemente, em comparação a distintos processos.
- Mas, além disso, as *threads* dentro de um processo não são protegidas umas das outras.

Considere o custo da criação de uma nova *thread* em um ambiente de execução existente. As principais tarefas são: alocar uma região para sua pilha; fornecer valores iniciais para os registradores internos do processador; e definir para o estado de execução (inicialmente, ele pode ser *SUSPENDED* ou *RUNNABLE*) e um valor para a prioridade inicial da *thread*. Como o ambiente de execução já existe, apenas um identificador para ele precisa ser posto no registro descritor de uma *thread* (o qual contém os dados necessários para gerenciar a execução da *thread*).

As sobrecargas computacionais associadas à criação de um processo são, em geral, consideravelmente maiores do que as da criação de uma nova *thread*, pois um novo ambiente de execução deve ser criado primeiro, incluindo suas tabelas de espaço de endereçamento. Anderson *et al.* [1991] citam um valor de cerca de 11 milissegundos para criar um novo processo UNIX e de cerca de 1 milissegundo para criar uma *thread* na arquitetura de processador CVAX executando o núcleo Topaz; em cada caso, o tempo medido inclui a nova entidade simplesmente realizando uma chamada a um procedimento vazio e depois terminando. Esses valores são fornecidos apenas como um parâmetro aproximado.

Quando a nova entidade realiza algum trabalho útil, em vez de chamar um procedimento vazio, existem também custos a longo prazo, os quais estão sujeitos a serem maiores para um novo processo do que para uma nova *thread* dentro de um processo existente. Em um núcleo que suporta memória virtual, o novo processo acarretará erros de falta de página quando dados e instruções forem referenciados pela primeira vez; inicialmente, as caches não conterão valores de dados para o novo processo e ele deverá povoar as entradas de cache enquanto for executando. Por outro lado, no caso da criação de *threads*, essas sobrecargas a longo prazo também podem ocorrer, mas elas tendem a ser menores. Quando a *thread* acessa código e dados que foram acessados recentemente por outras *threads* dentro do processo, ela tira proveito automaticamente de qualquer uso de cache, ou de memória principal, que tenha ocorrido anteriormente.

A segunda vantagem para o desempenho das *threads* está relacionada ao *chaveamento* entre *threads* – isto é, executar uma *thread*, em vez de outra, em determinado processador. Esse custo é o mais importante, pois ele pode acontecer muitas vezes durante a duração de uma *thread*. O chaveamento entre *threads* que compartilham o mesmo ambiente de execução é consideravelmente menos oneroso do que o chaveamento entre *threads* pertencentes a processos diferentes. As sobrecargas associadas ao chaveamento de *thread* são o escalonamento (a escolha da próxima *thread* a executar) e a troca de contexto.

Um contexto de processador compreende os valores dos registradores internos do processador, como o contador de programa e o domínio de proteção de hardware corrente – o espaço de endereçamento e o modo de proteção do processador (supervisor ou usuário). Uma *troca de contexto* é a transição que ocorre no chaveamento entre *threads*, ou quando uma única *thread* faz uma chamada de sistema ou, ainda, quando recebe uma exceção. Ela envolve o seguinte:

- O salvamento do estado original dos registradores internos do processador e a carga de um novo estado.
- Em alguns casos, uma transferência para um novo domínio de proteção – isso é conhecido como *transição de domínio*.

O chaveamento de *threads* que compartilham o mesmo ambiente de execução inteiramente em nível de usuário não envolve nenhuma transição de domínio e é relativamente barata. O chaveamento para o núcleo, ou para outra *thread* pertencente ao mesmo ambiente de execução por meio do núcleo, envolve transição de domínio. Portanto, o custo é maior, mas, se o núcleo for mapeado no espaço de endereçamento do processo, ele ainda será relativamente baixo. Entretanto, no chaveamento entre *threads* pertencentes a diferentes ambientes de execução existem sobrecargas maiores. O quadro a seguir explica as implicações do uso da cache de hardware para essas transições de domínio. Custos de prazos maiores, como atualizar entradas de cache de hardware e carregar páginas na memória principal, são mais propensos de ocorrerem quando ocorre tal transição de domínio. Os valores citados por Anderson *et al.* [1991] são de 1,8 milissegundos para o chaveamento entre processos UNIX e 0,4 milissegundos para o núcleo Topaz fazer o chaveamento entre *threads* pertencentes ao mesmo ambiente de execução. Custos ainda mais baixos (0,04 milissegundos) são obtidos, caso as *threads* sejam chaveadas em nível de usuário. Esses valores são dados apenas como um parâmetro aproximado; eles não medem os custos de uso de cache em prazos maiores.

No exemplo anterior, do processo cliente com duas *threads*, a primeira *thread* gera dados e os passa para a segunda *thread*, a qual faz uma invocação a método remoto ou uma chamada de procedimento remota; como as *threads* compartilham um espaço de endereçamento, não há necessidade de utilizar passagem de mensagem para transmitir os dados. As duas *threads* podem acessar os dados por meio de uma variável comum. Aqui residem a vantagem e o perigo de usar processos *multi-threaded*. A conveniência e a eficiência do acesso a dados compartilhados é uma vantagem. Isso é particularmente verdade para os servidores, conforme mostrou o exemplo de dados de arquivo armazenados em cache, dado anteriormente. Entretanto, as *threads* que compartilham um espaço de endereçamento, e que não são escritos em uma linguagem fortemente tipada, não são protegidas umas das outras. Uma *thread* pode arbitrariamente alterar dados usados por outra *thread*, causando um erro. Se for exigida tal proteção, então uma linguagem fortemente tipada deve ser usada, ou pode ser preferível utilizar múltiplos processos em vez de múltiplas *threads*.

Programação com *threads* ◊ A programação com *threads* é concorrente, conforme estudado tradicionalmente, por exemplo, na área de sistemas operacionais. Esta seção se refere aos seguintes conceitos de programação concorrente, que são totalmente explicados por Bacon [2002]: *condição de corrida*, *seção crítica* (Bacon chama isso de *região crítica*), *monitor*, *variável de condição*, *semáforo*.

A maior parte dos programas baseados em *threads* é feita em uma linguagem convencional, como C, que foi estendido para ter uma biblioteca de *threads*. O pacote C Threads, desenvolvido para o sistema operacional Mach, é um exemplo disso. Mais recentemente, o padrão POSIX Threads IEEE 1003.1c-1995, conhecido como *pthreads*, foi amplamente adotado. Boykin *et al.* [1993] descrevem o C Threads e o *pthreads* no contexto do Mach.

Algumas linguagens fornecem suporte direto para *threads*, incluindo Ada95 [Burns e Wellings 1998], Modula-3 [Harbison 1992] e, mais recentemente, Java [Oaks e Wong 1999]. Daremos aqui uma visão geral das *threads* Java.

O problema do *alias* ◊ Normalmente, as unidades de gerenciamento de memória incluem uma cache em hardware para acelerar a conversão entre endereços virtuais para endereços físicos, chamada de *translation lookaside buffer* (TLB). As TLBs, e também as caches de dados e instruções, ao usar endereços virtuais apresentam o *problema do alias*. O mesmo endereço virtual pode ser válido em dois espaços de endereçamentos diferentes, mas, em geral, se supõem que eles se refiram a diferentes dados físicos desses dois espaços de endereçamento. A menos que suas entradas sejam rotuladas (*tagged*) com um identificador de contexto, as TLBs e as caches não tem conhecimento disso e, portanto, podem conter dados incorretos. Assim, o conteúdo da TLB e da cache precisa ser invalidado quando ocorre um chaveamento entre espaços de endereçamento diferentes. As caches que são endereçadas fisicamente não sofrem do problema do alias, porém é mais comum usar endereços virtuais para caches, principalmente porque elas permitem que as pesquisas sejam sobrepostas com conversão de endereço.

Assim como em qualquer implementação de *threads*, a linguagem Java fornece métodos para criá-las, destruí-las e sincronizá-las. A classe Java *Thread* inclui o construtor e os métodos de gerenciamento listados na Figura 6.8. Os métodos de sincronização, *Thread* e *Object*, aparecem na Figura 6.9.

Ciclo de vida de uma *thread* ◊ Uma nova *thread* é criada na mesma máquina virtual Java (JVM) que da sua criadora, no estado *SUSPENDED*. Após se tornar *RUNNABLE* com o método *start()*, ela executa o método *run()* de um objeto fornecido em seu construtor. A JVM e as *threads* são todas executadas em um único processo no sistema operacional. As *threads* podem receber uma prioridade, de modo que as implementações Java que suportam prioridades executam uma *thread* em particular, em detrimento a qualquer outra *thread* com prioridade mais baixa. Uma *thread* termina quando retorna do método *run()* ou quando seu método *destroy()* é chamado.

Os programas podem gerenciar as *threads* em grupos. Toda *thread* pertence a um grupo, o qual é designado no momento de sua criação. Os grupos de *threads* são úteis quando vários aplicativos coexistem na mesma JVM. Um exemplo de uso de grupos é relacionado com a segurança: por padrão, uma *thread* de um grupo não pode executar operações de gerenciamento em uma *thread* de outro gru-

Thread(ThreadGroup group, Runnable target, String name)
Cria uma nova *thread* no estado *SUSPENDED*, a qual pertencerá a *group* e será identificado como *name*; a *thread* executará o método *run()* de *target*.

setPriority(int newPriority), getPriority()
Configura e retorna a prioridade da *thread*.

run()
A *thread* executa o método *run()* de seu objeto de destino, caso ele tenha um; caso contrário, ela executa seu próprio método *run()* (*Thread* implementa *Runnable*).

start()
Muda o estado da *thread* de *SUSPENDED* para *RUNNABLE*.

sleep(int millisecs)
Passa a *thread* para o estado *SUSPENDED* pelo tempo especificado.

yield()
Passa para o estado *READY* e ativa o escalonamento.

destroy()
Termina (destrói) a *thread*.

Figura 6.8 Construtor e métodos de gerenciamento de *threads* Java.

thread.join(int millisecs)
Bloqueia até a *thread* terminar, mas não mais que o tempo especificado.

thread.interrupt()
Interrompe a *thread*: faz com que ela retorne de uma invocação a método que causa bloqueio, como *sleep()*.

object.wait(long millisecs, int nanosecs)
Bloqueia a *thread* até que uma chamada feita para *notify()*, ou *notifyAll()*, em *object*, ative a *thread*, ou que a *thread* seja interrompida, ou ainda, que o tempo especificado tenha decorrido.

object.notify(), object.notifyAll()
Ativa, respectivamente, uma ou todas as *threads* que tenham chamado *wait()* em *object*.

Figura 6.9 Chamadas de sincronização de *thread* Java.

po. Assim, por exemplo, uma *thread* de aplicativo não pode interromper maliciosamente uma *thread* sistema de janelas (AWT).

Os grupos de *threads* também facilitam o controle das prioridades relativas das *threads* (nas implementações Java que suportam prioridades). Isso é útil para os navegadores que executam *applets* e para servidores web que executam programas chamados *servlets* [Hunter e Crawford 1998], os quais criam páginas web dinâmicas. Uma *thread* não privilegiada dentro de um *applet*, ou de um *servlet*, só pode criar uma nova *thread* que pertença ao seu próprio grupo ou a um grupo descendente criado dentro dela (as restrições exatas dependem do gerenciador de segurança – *SecurityManager* – que esteja em vigor). Os navegadores e servidores podem atribuir *threads* pertencentes aos diversos *applets* ou *servlets* a diferentes grupos, e configurar a prioridade máxima de cada grupo como um todo (incluindo os grupos descendentes). Não há meios de uma *thread* de *applet*, ou *servlet*, anular as prioridades do grupo configuradas pelas *threads* gerenciadoras, pois elas não podem ser modificadas por chamadas de *setPriority()*.

Sincronização de *threads* ◊ A programação de um processo *multi-threaded* exige bastante cuidado. Os principais problemas são o compartilhamento dos objetos e as técnicas usadas para coordenação e cooperação entre *threads*. As variáveis locais de cada *thread*, presentes nos métodos, são privativas – assim como suas pilhas. Entretanto, as *threads* não possuem cópias privativas de variáveis estáticas (classe), nem variáveis de instância de objeto.

Considere, por exemplo, as filas compartilhadas que descrevemos anteriormente nesta seção, em que as *threads* de E/S e as *threads* trabalhadores inserem e retiram pedidos. Em princípio, condições de corrida (*race conditions*) podem surgir quando as *threads* manipulam estruturas de dados, como as filas, concorrentemente. Os pedidos enfileirados podem ser perdidos ou duplicados, a não ser que as manipulações de ponteiro pelas *threads* sejam cuidadosamente coordenadas.

A linguagem Java fornece a palavra-chave *synchronized* para os programadores designarem a construção conhecida como *monitor* para a coordenação de *threads*. Os programadores designam métodos inteiros, ou blocos de código arbitrários, como pertencentes a um monitor associado a um objeto individual. A garantia do monitor é que no máximo uma *thread* pode ser executada dentro dele em certo momento. Em nosso exemplo, poderíamos serializar as tarefas das *threads* de E/S e trabalhadores, designando os métodos *addTo()* e *removeFrom()* na classe *Queue* como métodos *synchronized*. Assim, todos os acessos às variáveis dentro desses métodos seriam realizados com exclusão mútua, com relação às invocações desses métodos.

A linguagem Java permite que as *threads* sejam bloqueadas e liberadas por meio de objetos arbitrários que atuam como variáveis de condição. Uma *thread* que precisa bloquear até que ocorra uma certa condição, chama o método *wait()* de um objeto. Todos os objetos implementam esse método, pois ele pertence à classe raiz *Object*. Outra *thread* chama *notify()* para desbloquear no máximo uma *thread*, ou *notifyAll()*, para desbloquear todas as *threads* que estão esperando nesse objeto. Os dois métodos de notificação também pertencem à classe *Object*.

Como exemplo, quando uma *thread* trabalhador descobre que não existe nenhum pedido para processar, ela chama *wait()* na instância de *Queue*. Quando a *thread* de E/S adiciona, subseqüentemente, um pedido na fila, ela chama o método *notify()* da fila para ativar o trabalhador.

Os métodos de sincronização Java aparecem na Figura 6.9. Além das primitivas de sincronização que já mencionamos, o método *join()* bloqueia o chamador até o término de uma outra *thread*. O método *interrupt()* é útil para ativar prematuramente uma *thread* que esteja em espera. Todas as primitivas de sincronização padrão, como os semáforos, podem ser implementadas em Java. Mas é necessário cuidado, pois as garantias do monitor de Java se aplicam somente no código *synchronized* de um objeto; uma classe pode ter uma mistura de métodos *synchronized* e não-*synchronized*. Note também que o monitor implementado por um objeto Java tem apenas uma variável de condição implícita, enquanto, em geral, um monitor pode ter diversas variáveis de condição.

Escalonamento de *threads* ◊ Há uma distinção importante entre escalonamento preemptivo e não-preemptivo de *threads*. No *escalonamento preemptivo*, uma *thread* pode ser suspensa, em qualquer ponto de sua execução, para permitir a execução de outra *thread*. No *escalonamento não-preemptivo*, uma *thread* é executada até realizar uma operação, por exemplo, uma chamada de sistema, que a bloqueie e leve ao escalonamento de uma outra *thread*.

A vantagem do escalonamento não-preemptivo é que qualquer seção de código que não contenha uma operação que possa bloqueá-la é automaticamente uma seção crítica. Assim, as condições de corrida são convenientemente evitadas. Por outro lado, as *threads* cujo escalonamento é não-preemptivo não são apropriadas para tirar proveito de um multiprocessador, pois elas são executadas exclusivamente. Deve-se tomar cuidado com seções de código de execução longa que não contenham chamadas que provoquem escalonamento. Nesses casos, talvez o programador necessite prever a inserção de chamadas a *yield()*, cuja única função é permitir que outras *threads* sejam escalonadas e executadas. As *threads* cujo escalonamento é não-preemptivo também não são convenientes para aplicativos em tempo real, nos quais eventos devem ser processados dentro um tempo limite após sua ocorrência.

Por padrão, a linguagem Java não suporta processamento em tempo real, embora existam implementações para tempo real [www.rtj.org]. Por exemplo, os aplicativos multimídia que processam dados, como voz e vídeo, têm requisitos de tempo real tanto para comunicação como para processamento (por exemplo, filtragem e compactação) [Govindan e Anderson 1991]. O Capítulo 17 examinará os requisitos de escalonamento de *threads* para tempo real. O controle de processos em plantas industriais é outro exemplo de aplicações de tempo real. Em geral, cada aplicação de tempo real tem seus próprios requisitos de escalonamento de *threads*. Portanto, às vezes é desejável que os aplicativos implementem sua própria política de escalonamento. Para considerarmos isso, veremos agora a implementação das *threads*.

Implementação de *threads* ◊ Muitos núcleos de sistemas operacionais fornecem suporte nativo para processos *multi-threaded*, incluindo Windows, Linux, Solaris, Mach e Chorus. Esses núcleos fornecem chamadas de sistema para criação e gerenciamento de *threads* e escalonam as *threads* individualmente. Outros núcleos têm apenas a abstração de processo *single-threaded*. Nesse caso, o suporte a processos *multi-threaded* é então implementado em uma biblioteca de procedimento ligada aos programas aplicativos. Em tais casos, o núcleo não tem conhecimento dessas *threads* em nível de usuário e, portanto, não pode escaloná-las individualmente. A própria biblioteca de suporte a *threads* realiza o escalonamento. Uma *thread* que se bloqueia, ao realizar uma chamada de sistema, bloqueia todo o processo e, portanto, todas as *threads* dentro dele. Uma forma para evitar isso é usar apenas chamadas de sistema não bloqueantes (ou assíncronas). Analogamente, a implementação da biblioteca de *threads* pode utilizar temporizadores e recursos de interrupção de software fornecidos pelo núcleo para conceder fatias de tempo de execução entre as várias *threads*.

Quando não é fornecido suporte algum do núcleo para processos *multi-threaded*, a implementação de *threads* em nível de usuário apresenta os seguintes problemas:

- As *threads* pertencentes a um mesmo processo não podem tirar proveito de um multiprocessador.
- Uma *thread* que gera um erro de falta de página bloqueia o processo inteiro e, conseqüentemente, todas as *threads* dentro dele.

- As *threads* pertencentes a diferentes processos não podem ser escalonadas de acordo com um único esquema de prioridade relativa.

Por outro lado, as implementações de *threads* em nível de usuário têm vantagens significativas em relação às implementações em nível de núcleo (ou sistema):

- Certas operações de *threads* são significativamente menos dispendiosas. Por exemplo, o chaveamento entre *threads* pertencentes ao mesmo processo não envolve necessariamente uma chamada de sistema, que é uma tarefa relativamente onerosa no núcleo.
- Dado que o módulo de escalonamento das *threads* é implementado fora do núcleo, ele pode ser personalizado, ou alterado, de acordo com os requisitos específicos do aplicativo. Variações nos requisitos do escalonamento ocorrem geralmente devido às considerações específicas do aplicativo, como a natureza de tempo real do processamento de aplicações multimídia.
- Pode-se ter um número maior de *threads* do que poderia ser razoavelmente fornecido, por padrão, em um núcleo.

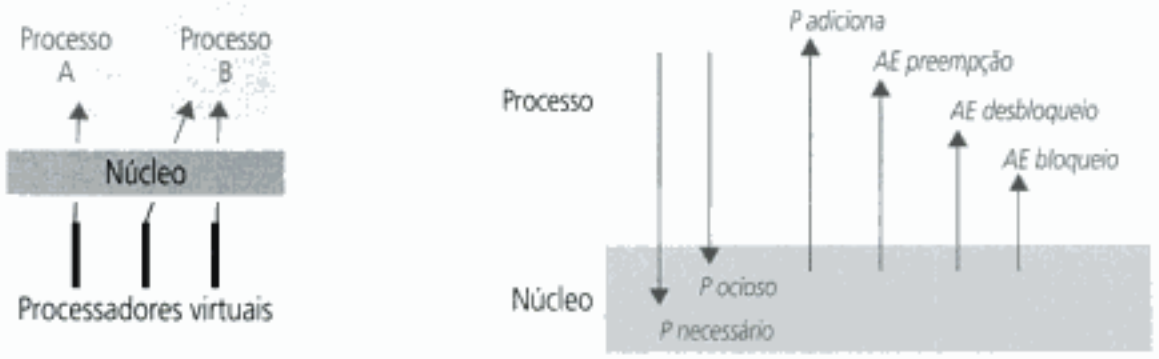
É possível combinar as vantagens das implementações de *threads* em nível de usuário e em nível de núcleo. Uma estratégia aplicada, por exemplo, no núcleo Mach [Black 1990], é permitir que o código em nível de usuário forneça sugestões sobre agendamentos de execução para o escalonador de *threads* do núcleo. Outra, adotada no sistema operacional Solaris 2, é uma forma hierárquica de escalonamento. Cada processo cria uma ou mais *threads* em nível de núcleo, conhecidos no Solaris como "processos leves". Também são suportadas *threads* em nível de usuário. Um escalonador em nível de usuário atribui cada *thread* em nível de usuário a uma *thread* em nível de núcleo. Esse esquema pode tirar proveito dos multiprocessadores e também se beneficia pelo fato de algumas operações de criação e chaveamento entre *threads* ocorrem em nível de usuário. A desvantagem do esquema é que ainda falta flexibilidade: se uma *thread em nível de núcleo* bloqueia, todas as *threads* em nível de usuário atribuídos a ela também serão impedidas de executar, independentemente de estarem aptos a isso.

Vários projetos de pesquisa têm desenvolvido escalonamento hierárquico para oferecer mais eficiência e flexibilidade. Isso inclui o trabalho feito nas ativações do escalonador [Anderson *et al.* 1991], no trabalho sobre multimídia de Govindan e Anderson [1991], no sistema operacional Psyche [Marsh *et al.* 1991], no núcleo Nemesis [Leslie *et al.* 1996] e no núcleo SPIN [Bershad *et al.* 1995]. A idéia que orienta esses projetos é que, o que um escalonamento em nível de usuário exige do núcleo não apenas um conjunto de *threads em nível de núcleo*, nos quais se possa mapear *threads* em nível de usuário, mas também que o núcleo notifique sobre *eventos* relevantes para a tomada de decisão. Descrevemos o projeto das ativações de escalonador para tornar isso mais claro.

O pacote FastThreads, de Anderson *et al.* [1991], é uma implementação de um sistema de escalonamento hierárquico baseada em eventos. Eles consideram como principais componentes do sistema um núcleo sendo executado em um computador com um ou mais processadores e um conjunto de programas aplicativos funcionando sobre ele. Cada processo aplicativo contém um escalonador em nível de usuário, o qual gerencia as *threads* dentro do processo. O núcleo é responsável por alocar *processadores virtuais* para os processos. O número de processadores virtuais atribuídos a um processo depende de fatores como os requisitos dos aplicativos, suas prioridades relativas e da demanda total nos processadores. A Figura 6.10a mostra um exemplo de uma máquina com três processadores, na qual o núcleo aloca um processador virtual para o processo *A*, executando uma tarefa de prioridade relativamente baixa, e dois processadores virtuais ao processo *B*. Eles são processadores *virtuais* porque o núcleo pode alocar, à medida que o tempo passa, diferentes processadores físicos para cada processo, enquanto mantém sua garantia da quantidade de processadores alocados.

O número de processadores virtuais atribuídos a um processo também pode variar. Os processos podem devolver um processador virtual que não precisam mais; eles também podem solicitar processadores virtuais extras. Por exemplo, se o processo *A* tiver solicitado um processador virtual extra e *B* terminar, então o núcleo poderá atribuir um processador para *A*.

A Figura 6.10b mostra que um processo notifica o núcleo quando ocorre um de dois tipos de evento: quando um processador virtual está "ocioso" e não é mais necessário ou quando um processador virtual extra é exigido.



A. Atribuição de processadores virtuais a processos

B. Eventos entre o escalonador em nível de usuário e o núcleo

Legenda: P = processador; AE = ativação do escalonador

Figura 6.10 Ativações do escalonador.

A Figura 6.10b também mostra que o núcleo notifica o processo, quando ocorre um de quatro tipos de evento. Uma *ativação do escalonador* (AE) é uma chamada do núcleo para um processo, a qual notifica o escalonador do processo sobre um evento. Esse tipo de "intervenção" de uma camada inferior (o núcleo) no código de uma camada superior é, às vezes, chamada de *upcall*. O núcleo cria uma AE carregando os registradores internos de um processador físico com um contexto que o faz iniciar a execução do processo em um endereço designado pelo escalonador em nível de usuário. Assim, uma AE também é uma unidade de alocação de fatia de tempo em um processador virtual. O escalonador em nível de usuário tem a tarefa de atribuir suas *threads READY* ao conjunto de AEs que estão em execução dentro dele. O número de AEs é no máximo igual ao número de processadores virtuais que o núcleo atribuiu ao processo.

Os quatro tipos de evento que o núcleo notifica para o escalonador em nível de usuário (o qual vamos nos referir simplesmente como "escalonador") são os seguintes:

Processador virtual alocado: o núcleo atribuiu um novo processador virtual para o processo e essa é a primeira fatia de tempo nele; o escalonador pode carregar a AE com o contexto de uma *thread READY*, a qual pode então recomeçar a execução.

AE bloqueio: uma AE foi bloqueada no núcleo e este está usando uma nova AE para notificar o escalonador. O escalonador configura o estado da *thread* correspondente como *BLOCKED* e pode alocar uma *thread READY* para a AE que está fazendo a notificação.

AE desbloqueio: uma AE que foi bloqueada no núcleo desbloqueia e está apta para executar em nível de usuário novamente; agora, o escalonador pode devolver a *thread* correspondente para a lista *READY*. Para criar a AE que faz a notificação, o núcleo aloca um novo processador virtual para o processo, ou faz a preempção de outra AE no mesmo processo. Neste último caso, ele também comunica o evento de preempção para o escalonador, o qual pode reavaliar sua alocação de *threads* para AEs.

AE preempção: o núcleo retirou a AE especificada do processo (embora ele possa fazer isso para alocar um processador para uma nova AE no mesmo processo); o escalonador coloca a *thread* preemptada na lista *READY* e reavalia a alocação de *threads*.

Esse esquema de escalonamento hierárquico é flexível, pois o escalonador em nível de usuário do processo pode alocar *threads* para AEs de acordo com políticas construídas sobre os eventos de baixo nível. O núcleo sempre se comporta da mesma maneira. Ele não tem influência sobre o comportamento do escalonador em nível de usuário, mas o ajuda através de suas notificações de evento e fornecendo o estado das *threads* bloqueadas e preemptadas. O esquema é potencialmente eficiente, pois nenhuma *thread* em nível de usuário permanece no estado *READY* caso haja um processador virtual para ela ser executada.

6.5 Comunicação e invocação

Nesta seção nos concentraremos nos aspectos de comunicação como parte da implementação do que chamamos genericamente de invocação – cujo propósito é efetuar uma operação sobre um recurso e, no caso de chamadas de procedimentos remotos, notificação de eventos e invocação a métodos remotos, em um espaço de endereçamento diferente de quem a executa.

Vamos abordar os problemas e conceitos de projeto do sistema operacional, fazendo as seguintes perguntas sobre o SO:

- Quais primitivas de comunicação ele fornece?
- Quais protocolos ele suporta e o quanto a implementação da comunicação é aberta?
- Quais passos são dados para tornar a comunicação a mais eficiente possível?
- Que suporte é fornecido para operações com alta latência e em modo desconectado?

Primitivas de comunicação ◊ Alguns núcleos projetados para sistemas distribuídos fornecem primitivas de comunicação customizadas para os tipos de invocação descritos no Capítulo 5. O Amoeba [Tanenbaum *et al.* 1990], por exemplo, fornece *doOperation*, *getRequest* e *sendReply* como primitivas. O Amoeba, o sistema V e o Chorus fornecem primitivas de comunicação em grupo. Colocar funcionalidade de comunicação em um nível relativamente alto no núcleo tem a vantagem da eficiência na programação de aplicativos. Se, por exemplo, a camada de *middleware* fornecer RMI sobre soquetes UNIX conectados (TCP), então um cliente deverá fazer duas chamadas de sistema de comunicação (*escrita* e *leitura* de soquete) para cada invocação remota. No Amoeba, seria exigida apenas uma chamada para *doOperation*. As economias na sobrecarga de chamada de sistema estão sujeitas a serem ainda maiores na comunicação em grupo.

Na prática, a camada de *middleware*, e não o núcleo, fornece a maior parte dos recursos de comunicação de alto nível encontrados nos sistemas atuais, incluindo RPC/RMI, notificação de evento e comunicação em grupo. Desenvolver esse software complexo como código em nível de usuário é muito mais simples do que desenvolvê-lo para o núcleo. Normalmente, os desenvolvedores implementam a camada de *middleware* em soquetes, fornecendo acesso aos protocolos Internet padrão – freqüentemente, soquetes TCP (com conexão), mas às vezes soquetes UDP (sem conexão). Os principais motivos para o uso de soquetes são a portabilidade e a interação: a camada de *middleware* deve executar no máximo possível dos sistemas operacionais amplamente usados. Todos os sistemas operacionais mais comuns, como UNIX e a família Windows, fornecem APIs de soquete semelhantes, dando acesso aos protocolos TCP e UDP.

Apesar do uso difundido de soquetes TCP e UDP, fornecidos pelos núcleos comuns, pesquisas continuam sendo feitas sobre primitivas de comunicação de menor custo, computacional e latência, em núcleos experimentais. Examinaremos melhor os problemas de desempenho na Seção 6.5.1.

Protocolos e sistemas abertos ◊ Um dos principais requisitos do sistema operacional é fornecer protocolos padrão que permitam a interligação em rede entre implementações de *middleware* sobre diferentes plataformas. Nos anos 80, vários núcleos de pesquisa incorporaram seus próprios protocolos de rede otimizados para interações de RPC – notadamente a RPC do Amoeba [van Renesse *et al.* 1989], VMTP [Cheriton 1986] e a RPC do Sprite [Ousterhout *et al.* 1988]. Entretanto, esses protocolos não foram usados fora de seus ambientes de pesquisa nativos. Em contraste, os projetistas dos núcleos Mach 3.0 e Chorus (assim como o L4 [Härtig *et al.* 1997]) decidiram deixar a escolha dos protocolos de interligação em rede totalmente aberta. Esses núcleos fornecem passagem de mensagens apenas entre processos locais e deixam o processamento do protocolo de rede para um servidor que é executado no núcleo.

Diante da necessidade diária de acesso à Internet, a compatibilidade com o TCP e UDP é exigida dos sistemas operacionais para todos os dispositivos interligados em rede, a não ser os menores. O sistema operacional ainda é obrigado a permitir que o *middleware* tire proveito dos novos protocolos de baixo nível. Por exemplo, os usuários querem usufruir das tecnologias sem fio, como a transmissão de raios infravermelhos e rádiofreqüência (RF), preferivelmente, sem ter de atualizar seus aplicativos. Isso exige que protocolos correspondentes, como o IrDA para interligação em rede com raios infraver-

melhos, o Bluetooth, ou HomeRF para interligação em rede com RF, possam ser facilmente integrados ao sistema operacional.

Normalmente, os protocolos são organizados em uma *pilha* de camadas (veja o Capítulo 3). Muitos sistemas operacionais permitem que novas camadas sejam integradas estaticamente, como um "driver" de protocolo instalado, como, por exemplo, para o IrDA. Em contraste, a *composição dinâmica de protocolo* é uma técnica pela qual uma pilha de protocolos pode se adaptar dinamicamente para atender os requisitos de um aplicativo em particular, e para utilizar as camadas físicas que estiverem disponíveis, dada a conectividade corrente da plataforma. Por exemplo, um navegador web sendo executado em um computador *notebook* deve tirar proveito de um enlace sem fio remoto enquanto o usuário está em trânsito e, depois, de uma conexão Ethernet mais rápida, quando volta ao escritório.

Outro exemplo de composição dinâmica de protocolo é o uso de um protocolo requisição-resposta customizado para uma camada de interligação em rede sem fio, para reduzir as latências de ida e volta. Tem-se verificado que as implementações de TCP padrão têm desempenho deficiente em redes sem fio [Balakrishnan *et al.* 1996], as quais tendem a exibir taxas mais altas de perda de pacotes do que as redes cabeadas. Em princípio, um protocolo de requisição-resposta, como o HTTP, poderia ser construído de modo a funcionar de forma mais eficiente entre nós sem fio, usando diretamente a camada de transporte sem fio, em vez de usar uma camada TCP intermediária.

O suporte para composição de protocolo apareceu no projeto de Streams no UNIX [Ritchie 1984]. Mais recentemente, o Horus [van Renesse *et al.* 1995] e o x-núcleo [Hutchinson e Peterson 1991] proveram a composição dinâmica de protocolos.

6.5.1 Desempenho de uma invocação

O desempenho de uma invocação é um fator crítico no projeto de sistemas distribuídos. Quanto mais os projetistas separam a funcionalidade entre espaços de endereçamentos, mais invocações remotas são exigidas. Os clientes e servidores podem executar milhões de operações relacionadas à invocação, de modo que pequenas frações de milissegundos contam nos custos de invocação. As tecnologias de rede continuam a melhorar, mas os tempos de invocação não têm diminuído proporcionalmente ao aumento da largura de banda da rede. Esta seção explicará como as sobrecargas de software freqüentemente predominam sobre as sobrecargas de rede nos tempos de invocação – pelos menos para o caso de uma rede local ou intranet. Isso, em contraste com uma invocação remota pela Internet – por exemplo, na busca de um recurso web. Na Internet, as latências de rede são altamente variáveis, mas em média são altas; a largura de banda é freqüentemente baixa e a carga do servidor muitas vezes predomina sobre os custos de processamento por requisição.

As implementações de RPC e RMI têm sido o assunto de vários estudos, devido à ampla aceitação desses mecanismos para processamento cliente-servidor de propósito geral. Muita pesquisa tem sido feita sobre invocações na rede e, particularmente, sobre como os mecanismos de invocação podem tirar proveito das redes de alto desempenho [Hutchinson *et al.* 1989, van Renesse *et al.* 1989, Schroeder e Burrows 1990, Johnson e Zwaenepoel 1993, von Eicken *et al.* 1995, Gokhale e Schmidt 1996]. Há também, conforme mostraremos, um importante caso especial de RPCs entre processos contidos no mesmo computador [Bershad *et al.* 1990, Bershad *et al.* 1991].

Custos de uma invocação ◊ Chamar um procedimento convencional, ou um método, fazer uma chamada de sistema, enviar uma mensagem, realizar uma chamada de procedimento remoto ou uma invocação a método remoto, são todos exemplos de mecanismos de invocação. Cada mecanismo faz com que código seja executado fora do escopo do procedimento, ou objeto, que fez a chamada. Em geral, cada um envolve a comunicação de argumentos para esse código e o retorno de valores para o chamador. Os mecanismos de invocação podem ser síncronos, como por exemplo, no caso das chamadas de procedimentos remotos ou convencionais, ou assíncronos.

As distinções importantes, relacionadas ao desempenho, entre os mecanismos de invocação, independentemente de serem síncronos ou não, são se eles implicam em uma transição de domínio (isto é, se eles ultrapassam um espaço de endereçamento), se envolvem comunicação em rede e se causam escalonamento e chaveamento de contexto. A Figura 6.11 mostra os casos particulares de uma chamada de sistema, de uma invocação remota entre processos contidos no mesmo computador e de uma invocação remota entre processos em diferentes nós no sistema distribuído.

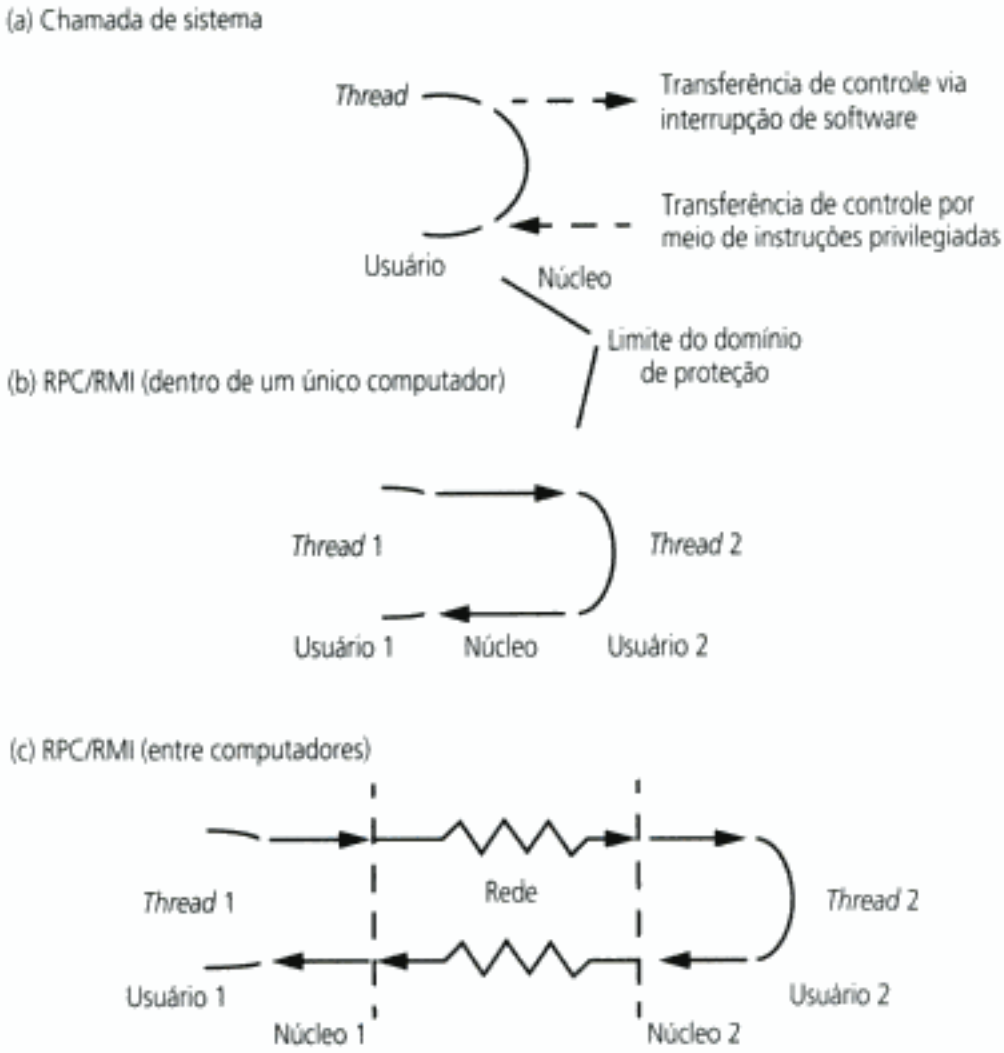


Figura 6.11 **Invocações entre espaços de endereçamentos.**

Invocação via rede ◊ Uma *RPC nula* (analogamente, uma *RMI nula*) é definida como uma RPC sem parâmetros que executa um procedimento nulo e não retorna valores. Sua execução envolve a troca de mensagens que transportam dados do sistema, mas não dados de usuário. O tempo de uma RPC nula entre processos de usuário em dois PCs de 500MHz, em uma rede local de 100 megabits/segundo, é da ordem de décimos de milissegundo. Em comparação, uma chamada de procedimento convencional nula pode demorar uma pequena fração de um microssegundo. Cerca de 100 bytes, no total, são passados pela rede para uma RPC nula. Com uma largura de banda bruta de 100 megabits/segundo, o tempo de transferência total da rede para esse volume de dados é de cerca de 0,01 milissegundos. Claramente, grande parte do *atraso* observado – o tempo de chamada total da RPC sentido por um cliente – leva em conta as ações do núcleo do sistema operacional e pelo código em tempo de execução da RPC em nível de usuário.

Os custos da invocação nula (RPC, RMI) são importantes porque eles medem uma sobrecarga fixa, a *latência*. Os custos da invocação aumentam com o tamanho dos argumentos e resultados, mas, em muitos casos, a latência é significativa se comparada com o restante do atraso.

Considere uma RPC que busca um volume de dados especificado em um servidor. Ela tem um único argumento de entrada, um valor inteiro, especificando o tamanho dos dados solicitados, e dois argumentos de resposta, um valor inteiro, especificando sucesso ou falha (o cliente pode ter fornecido um tamanho inválido) e, quando a chamada tem êxito, um array de bytes do servidor.

A Figura 6.12 mostra, esquematicamente, o atraso do cliente em relação ao tamanho dos dados solicitados. O atraso é aproximadamente proporcional ao tamanho, até que este chega a um limite, quan-

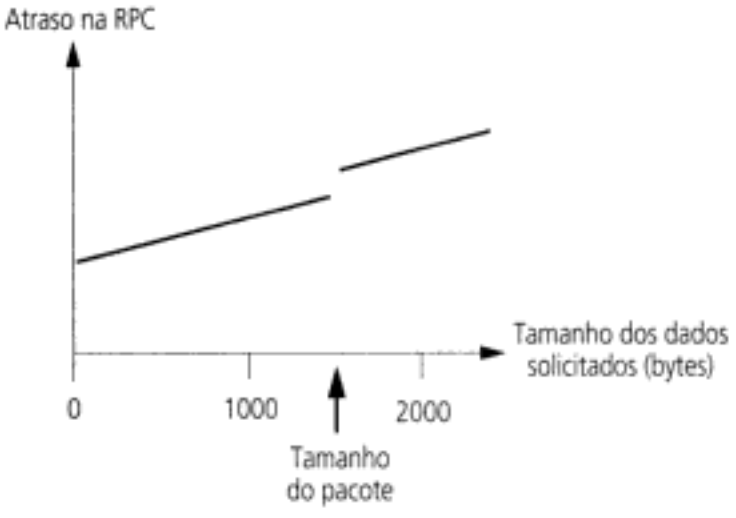


Figura 6.12 Atraso na RPC em relação ao tamanho do parâmetro.

do atinge praticamente o tamanho do pacote. Além desse limite, pelo menos um pacote extra precisa ser enviado para transportar os dados a mais. Dependendo do protocolo, mais um pacote pode ser usado para confirmar esse pacote extra. Saltos no gráfico ocorrem sempre que o número de pacotes aumenta.

O atraso não é o único valor de interesse para uma implementação de RPC: a largura de banda (ou *throughput*) da RPC também preocupa quando grandes volumes de dados são transferidos. Ela determina a taxa de transferência de dados entre computadores em uma única RPC. Se examinarmos a Figura 6.12, veremos que a largura de banda é relativamente baixa para pequenos volumes de dados, quando as sobrecargas de processamento fixas predominam. À medida que o volume de dados aumenta, a largura de banda sobe, já que essas sobrecargas se tornam globalmente menos significativas. Gokhale e Schmidt [1996] citam um *throughput* de cerca de 80 megabits/segundo na transferência de 64 quilobytes em uma RPC entre estações de trabalho usando uma rede ATM com largura de banda nominal de 155 megabits/segundo. Com cerca de 0,8 milissegundos para transferir 64 quilobytes, essa é a mesma ordem de grandeza do tempo citado anteriormente para uma RPC nula em uma rede Ethernet de 100 megabits/segundo.

Lembre-se de que as etapas em uma RPC são as seguintes (a RMI envolve etapas semelhantes):

- um *stub* cliente empacota os argumentos de chamada em uma mensagem, envia a mensagem de requisição, recebe e desempacota a resposta;
- no servidor, uma *thread* trabalhador recebe a requisição que chega, ou uma *thread* de E/S recebe a requisição e a repassa para uma *thread* trabalhador; em qualquer caso, o trabalhador chama o *stub* servidor apropriado;
- o *stub* servidor desempacota a mensagem de requisição, chama o procedimento designado, empacota e envia a resposta.

A seguir estão os principais componentes responsáveis pelo atraso da invocação remota, além dos tempos de transmissão na rede:

Empacotamento: o empacotamento e o desempacotamento, que envolvem a cópia e a conversão dos dados, se tornam uma sobrecarga significativa à medida que o volume de dados cresce.

Cópia de dados: potencialmente, mesmo após o empacotamento, os dados da mensagem são copiados várias vezes no andamento de uma RPC:

1. entre o limite usuário–núcleo, entre o espaço de endereçamento do cliente, ou do servidor, e os buffers do núcleo;
2. entre cada camada de protocolo (por exemplo, RPC/UDP/IP/Ethernet);
3. entre a interface de rede e os buffers do núcleo.

As transferências entre a interface de rede e a memória principal normalmente são feitas por acesso direto à memória (DMA – *Direct Memory Access*). O processador manipula as outras cópias.

Inicialização de pacotes: isso envolve a construção de todos os cabeçalhos do protocolo, incluindo as somas de verificação. Portanto, o custo é proporcional, em parte, ao volume de dados enviado.

Escalonamento de threads e troca de contexto: isso pode ocorrer como segue:

1. várias chamadas de sistema (isto é, trocas de contexto) são feitas durante uma RPC, pois os *stubs* ativam as operações de comunicação do núcleo;
2. uma ou mais *threads* de servidor são escalonadas;
3. se o sistema operacional possui um processo gerenciador de rede a parte, então cada operação de envio envolve uma troca de contexto para uma de suas *threads*.

Espera por confirmações: a escolha do protocolo de RPC pode influenciar o atraso, particularmente, quando grandes volumes de dados são enviados.

Um projeto cuidadoso do sistema operacional pode ajudar a reduzir alguns desses custos. O estudo de caso do projeto da RPC Firefly, disponível no endereço [www.cdk3.net/oss], mostra parte disso em detalhes, assim como técnicas que podem ser aplicadas dentro da implementação do *middleware*. Já mostramos como o suporte apropriado do sistema operacional para *threads* pode ajudar a reduzir os custos de usar *multi-threading*. O sistema operacional também pode ter impacto na redução das sobrecargas de cópia na memória, por meio de recursos de compartilhamento de memória.

Compartilhamento de memória ◊ As regiões compartilhadas (apresentadas na Seção 6.4) podem ser usadas para uma comunicação rápida entre um processo de usuário e o núcleo, ou entre processos de usuário. Os dados são trocados pela escrita e leitura na região compartilhada. Assim, os dados são passados de forma eficiente, sem necessidade de copiá-los no espaço de endereçamento do núcleo. Entretanto, chamadas de sistema e interrupções de software podem ser necessárias para sincronização – como quando o processo de usuário tiver escrito os dados que devem ser transmitidos, ou quando o núcleo tiver escrito dados para o processo de usuário consumir. É claro que uma região compartilhada é justificada apenas se for usada o suficiente para compensar o custo de sua configuração.

Mesmo com regiões compartilhadas, o núcleo ainda precisa copiar dados dos buffers na interface de rede. A arquitetura U-Net [von Eicken *et al.* 1995] permite que código em nível de usuário tenha acesso direto à própria interface de rede, para que ele possa transferir os dados para a rede sem nenhuma cópia.

Escolha do protocolo ◊ O atraso que um cliente sente durante as interações de requisição e resposta no TCP não é necessariamente pior do que para UDP e, às vezes, é melhor, particularmente para mensagens grandes. Entretanto, é preciso cuidado ao implementar interações de requisição e resposta em um protocolo como o TCP, que não foi especificamente projetado para esse propósito. Em particular, o comportamento dos buffers do TCP pode atrapalhar o bom desempenho, e suas sobrecargas de conexão o colocam em desvantagem, se comparado ao UDP, a não ser que mensagens suficientes sejam enviadas em uma única conexão para tornar a sobrecarga por pedido desprezível.

As sobrecargas de conexão do TCP são particularmente evidentes nas invocações web, pois o HTTP 1.0 estabelece uma conexão TCP separada para cada invocação. Os navegadores clientes são "travados", enquanto a conexão é estabelecida. Além disso, em muitos casos, o algoritmo de inicialização lenta (*slow-start*) do TCP tem o efeito de atrasar desnecessariamente a transferência de dados HTTP. O algoritmo de inicialização lenta opera de forma pessimista diante de um possível congestionamento da rede, permitindo que apenas uma pequena janela de dados seja enviada primeiro, antes que uma confirmação seja recebida. Nielson *et al.* [1997] discutem como o HTTP 1.1 utiliza as chamadas *conexões persistentes*, que são mantidas durante o curso de várias invocações. Assim, os custos iniciais de conexão são amortizados, desde que várias invocações sejam feitas no mesmo servidor web. Isso é provável de acontecer, pois os usuários freqüentemente buscam várias páginas no mesmo site, cada uma contendo várias imagens.

Nielson *et al.* também descobriram que modificar o comportamento padrão dos buffers do sistema operacional poderia ter um impacto significativo nos atrasos de uma invocação. Freqüentemente, é vantajoso reunir várias mensagens pequenas e depois enviá-las em conjunto, em vez de enviá-las em pacotes separados, devido à latência por pacote que descrevemos anteriormente. Por esse motivo,

o SO não envia, necessariamente, dados pela rede imediatamente após a chamada *write()* no soquete correspondente. O comportamento padrão do SO é esperar até que seu buffer esteja cheio, ou usar um *timeout*, como critério para enviar os dados pela rede, na esperança de que mais dados cheguem.

Nielson *et al.* descobriram que, no caso do HTTP 1.1, o comportamento padrão dos buffers do sistema operacional poderia causar atrasos significativos desnecessários, por causa dos *timeouts*. Para eliminar esses atrasos, eles alteraram as configurações do TCP no núcleo e forçaram o envio da rede nos limites das requisições HTTP. Esse é um bom exemplo de como um sistema operacional pode ajudar, ou atrapalhar, o *middleware* devido às políticas que implementa.

Invocação em um mesmo computador ◊ Bershad *et al.* [1990] relatam um estudo mostrando que, na instalação examinada, a maior parte das trocas entre espaços de endereçamentos ocorreu dentro de um mesmo computador e não, como poderia se esperar em uma instalação cliente-servidor, entre computadores. A tendência de colocar funcionalidade de serviço dentro de servidores em nível de usuário significa que cada vez mais invocações serão feitas para um processo local. Isso é particularmente verdade à medida que o uso de cache é exageradamente explorado quando os dados necessários para um cliente tendem a ser mantidos em um servidor local. O custo de uma RPC, dentro de um mesmo computador, está crescendo em importância como parâmetro de desempenho do sistema. Essas considerações sugerem que esse caso local deve ser otimizado.

A Figura 6.11 sugere que uma invocação entre espaços de endereçamentos seja implementada dentro de um computador exatamente como acontece entre computadores, exceto que a passagem de mensagens é local. Na verdade, freqüentemente, esse tem sido o modelo implementado. Bershad *et al.* [1990] desenvolveram um mecanismo de invocação mais eficiente para o caso de dois processos em uma mesma máquina, chamado *lightweight RPC* (*LRPC*), ou RPC leve. O projeto da LRPC é baseado em otimizações relativas à cópia de dados e ao escalonamento de *threads*.

Primeiramente, eles notaram que seria mais eficiente usar regiões de memória compartilhadas para a comunicação cliente-servidor, com uma região diferente (privativa) entre o servidor e cada um de seus clientes locais. Tal região contém uma ou mais *pilhas A* (de argumento) (veja a Figura 6.13). Em vez dos parâmetros do RPC serem copiados entre o núcleo e os espaços de endereçamentos dos usuários envolvidos, o cliente e o servidor são capazes de passar argumentos e valores de retorno diretamente, por meio dessa pilha A. A mesma pilha é usada pelos *stubs* cliente e servidor. Na LRPC, os argumentos são copiados uma vez: quando são empacotados na pilha A. Em uma RPC equivalente, eles são copiados quatro vezes: da pilha do *stub* cliente para uma mensagem; da mensagem para um buffer do núcleo; do buffer do núcleo para uma mensagem de servidor, e da mensagem para a pilha do *stub* servidor. Podem existir várias pilhas A em uma região compartilhada, pois várias *threads* do mesmo cliente podem chamar o servidor simultaneamente.

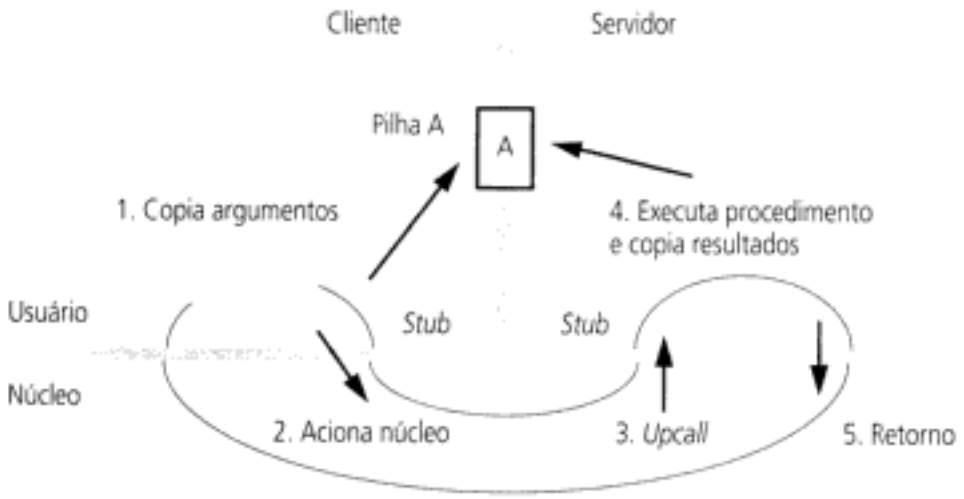


Figura 6.13 Uma chamada de procedimento remoto leve.

Bershad *et al.* também consideraram o custo do escalonamento das *threads*. Compare o modelo de chamada de sistema e das chamadas de procedimento remoto, na Figura 6.11. Quando ocorre uma chamada de sistema, a maioria dos núcleos não escalona uma nova *thread* para tratar dela, em vez disso, fazem uma troca de contexto na *thread* que fez a chamada para que ele manipule a chamada de sistema. Em uma RPC, um procedimento remoto pode existir em um computador diferente da *thread* cliente; portanto, uma *thread* diferente deve ser escalonada para executá-lo. No caso local, entretanto, pode ser mais eficiente a *thread* cliente – que, de outro modo, estaria no estado *BLOCKED* – chamar o procedimento no espaço de endereçamento do servidor.

Nesse caso, um servidor deve ser programado de forma diferente da maneira como descrevemos anteriormente. Em vez de configurar uma ou mais *threads*, as quais então "escutariam" as requisições nas portas, o servidor exporta um conjunto de procedimentos para serem chamados. As *threads* dos processos locais podem entrar no ambiente de execução do servidor, desde que comecem chamando um dos procedimentos exportados pelo servidor. Um cliente que precise invocar operações em um servidor deve primeiro vincular-se na interface do servidor (não mostrada na figura). Ele faz isso por meio do núcleo, o qual notifica o servidor. Quando o servidor tiver respondido ao núcleo com uma lista de endereços dos procedimentos permitidos, o núcleo, por sua vez, responderá ao cliente.

Uma invocação aparece na Figura 6.13. Uma *thread* cliente entra no ambiente de execução do servidor, primeiramente acionando o núcleo e apresentando a ele um recurso. O núcleo verifica esse pedido e só permite a troca de contexto para um procedimento válido no servidor; se ele for válido, o núcleo troca o contexto da *thread* para efetuar o procedimento no ambiente de execução do servidor. Quando o procedimento no servidor retorna, a *thread* retorna para o núcleo, o qual repassa o contexto da *thread* para o ambiente de execução do cliente. Note que os clientes e servidores empregam procedimentos *stub* para ocultar dos desenvolvedores dos aplicativos os detalhes que acabamos de descrever.

Discussão sobre a LRPC ◊ Não há muitas dúvidas de que a LRPC é mais eficiente do que a RPC para o caso local, desde que ocorram invocações suficientes para compensar os custos de gerenciamento de memória. Bershad *et al.* informam um fator três vezes menor para os atrasos da LRPC do que os da RPC executada de forma local.

A transparência da localização não é sacrificada na implementação de Bershad. Um *stub* cliente examina um conjunto de bits no momento da vinculação, que registra se o servidor é local ou remoto, e passa a usar LRPC ou RPC, respectivamente. O aplicativo não sabe qual é utilizada. Entretanto, a transparência da migração pode ser difícil de obter quando um recurso é transferido de um servidor local para um servidor remoto, ou *vice-versa*, devido à necessidade de trocar os mecanismos de invocação.

Em um trabalho posterior, Bershad *et al.* [1991] descrevem várias melhorias de desempenho, as quais são tratadas para operação em multiprocessador. As melhorias estão relacionadas com o fato de evitar o acionamento do núcleo e ao escalonamento dos processadores de maneira a evitar transições de domínio desnecessárias. Por exemplo, se um processador, alocado ao contexto de gerenciamento de memória do servidor, está ocioso no momento em que uma *thread* cliente tenta invocar um procedimento do servidor, então a *thread* deve ser transferida para esse processador. Isso evita uma transição de domínio; ao mesmo tempo, o processador do cliente pode ser reutilizado por outra *thread* no cliente. Essas melhorias envolvem uma implementação de escalonamento de *threads* em dois níveis (usuário e núcleo), conforme descrito na Seção 6.4.

6.5.2 Operação assíncrona

Já discutimos como o sistema operacional pode ajudar a camada de *middleware* a fornecer mecanismos de invocação remota eficientes. Mas também observamos que, no ambiente da Internet, os efeitos das altas latências, baixas larguras de banda e cargas de servidor altas, podem superar as vantagens oferecidas pelo SO. Podemos acrescentar a isso o fenômeno da desconexão e reconexão na rede, o qual pode ser considerado o causador de uma comunicação com latência extremamente alta. Os computadores móveis dos usuários não estão conectados na rede o tempo todo. Mesmo que tenham acesso remoto sem fio (por exemplo, usando GSM), os usuários podem ser desconectados peremptoriamente quando, por exemplo, o trem em que estão entra em um túnel.

Uma técnica comum para anular as latências altas é a operação assíncrona, que surge em dois modelos de programação: invocações concorrentes e invocações assíncronas. Esses modelos dizem respeito basicamente ao escopo de *middlewares*, em vez do projeto de núcleo de sistema operacional, mas é interessante considerá-los aqui, embora estejamos examinando o assunto do desempenho de uma invocação.

Fazendo invocações concorrentes ◊ No primeiro modelo, o *middleware* fornece apenas invocações bloqueantes, mas o aplicativo gera múltiplas *threads* para realizá-las concorrentemente.

Um bom exemplo de tal aplicativo é um navegador web. Normalmente, uma página web contém várias imagens. O navegador precisa buscar cada uma dessas imagens em uma requisição HTTP *GET* separada (pois os servidores web compatíveis com HTTP 1.0 padrão suportam apenas requisições para recursos únicos). O navegador não precisa obter as imagens em uma seqüência específica, de modo que ele faz requisições concorrentes – normalmente, até cerca de quatro por vez. Desse modo, o tempo que leva para concluir todas as requisições de imagem normalmente é menor do que o atraso que ocorre para fazer as requisições em série. Não apenas o atraso total da comunicação é menor, de modo geral, mas o navegador pode sobrepor computação, como a representação da imagem, com a comunicação.

A Figura 6.14 mostra as vantagens em potencial do fato de entrelaçar invocações (como as requisições HTTP) entre um cliente e um único servidor, em uma máquina com um só processador. No caso em série, o cliente empacota os argumentos, chama a operação *send* e depois espera até que a resposta do servidor chegue – depois do que, executa a operação *receive*, desempacota e, em seguida, processa os resultados. Após disso, ele pode fazer a segunda invocação.

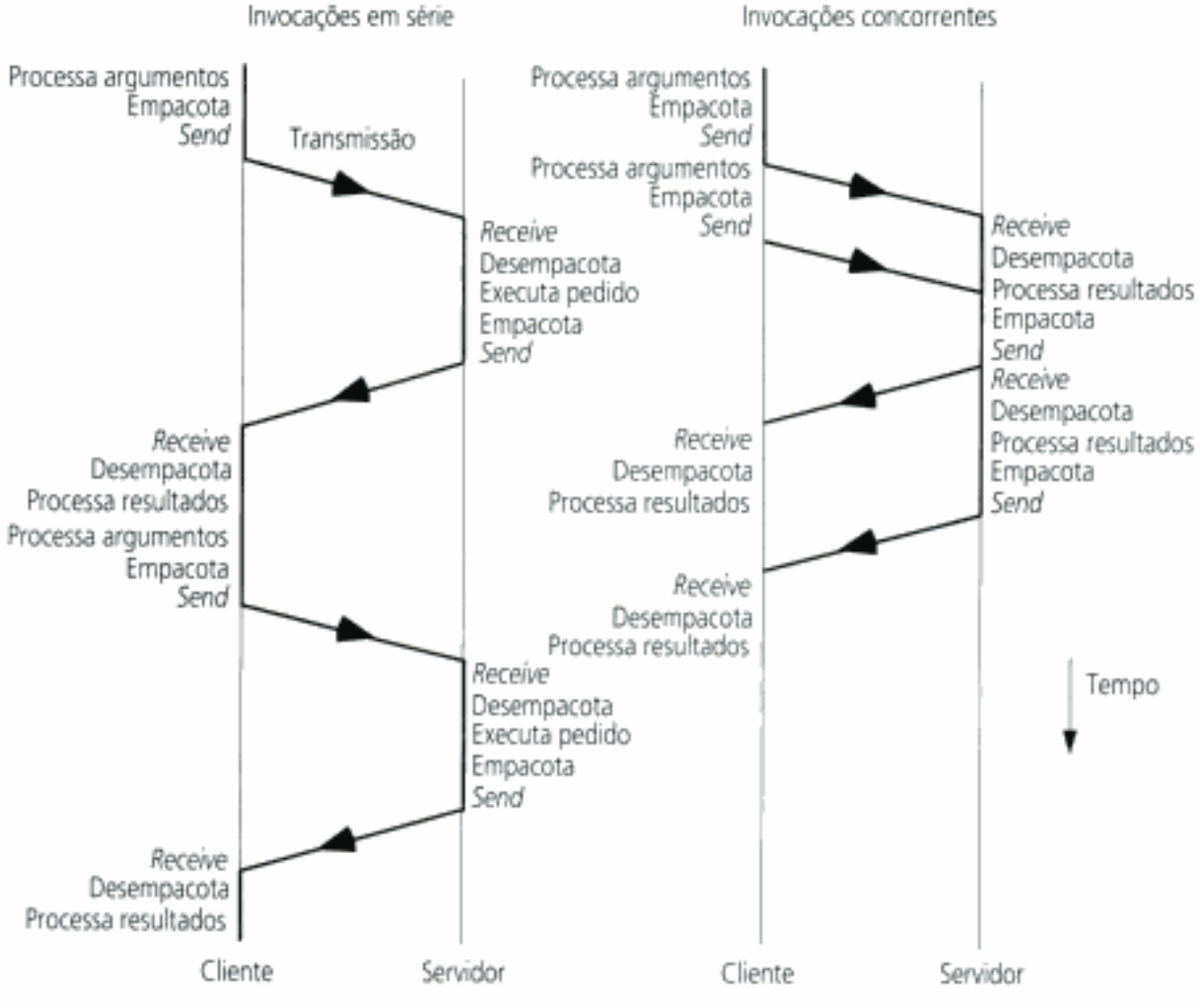


Figura 6.14 Tempos de invocações em série e concorrentes.

No caso concorrente, a primeira *thread* cliente empacota os argumentos e chama a operação *Send*. Então, a segunda *thread* faz imediatamente a segunda invocação. Cada *thread* espera para receber seus resultados. O tempo total gasto provavelmente é menor do que no caso em série, como mostra a figura. Vantagens semelhantes se aplicam, caso as *threads* clientes façam pedidos concorrentes para vários servidores; e se o cliente for executado em um multiprocessador, então um *throughput* ainda maior é totalmente possível, pois o processamento das duas *threads* também pode ser sobreposto.

Voltando ao caso particular do protocolo HTTP, o estudo de Nielson *et al.* [1997], a que nos referimos acima, também mediu os efeitos das invocações HTTP 1.1 concorrentes entrelaçadas (as quais eles chamam de *pipeline*) sobre conexões persistentes. Eles descobriram que o uso de *pipeline* reduziu o tráfego da rede e que podem trazer vantagens de desempenho para os clientes, desde que o sistema operacional forneça uma interface conveniente para esvaziar os buffers e assim modificar o comportamento padrão do TCP.

Invocações assíncronas ◊ Uma *invocação assíncrona* é aquela que é feita com o auxílio de uma chamada não bloqueante, a qual retorna assim que a mensagem de requisição da invocação tenha sido criada e esteja pronta para envio.

Às vezes, o cliente não exige nenhuma resposta (exceto, talvez, uma indicação de falha, caso o *host* de destino não possa ser atingido). Por exemplo, as invocações *sentido único* (*one-way*) do CORBA têm semântica *talvez*. Se necessário, o cliente usa uma chamada separada para receber os resultados da invocação. Por exemplo, o sistema de comunicação Mercury [Liskov e Shrira 1988] suporta invocação assíncrona. Uma operação assíncrona retorna um objeto chamado *promise* (promessa). Quando a invocação tem êxito, ou é considerada falha, o sistema Mercury coloca o status e os valores de retorno na promessa. O chamador usa a operação *claim* (reclamação) para obter os resultados da promessa. A operação *claim* bloqueia até que a promessa esteja pronta, quando então retorna os resultados ou exceções da chamada. A operação *ready* (pronto) está disponível para testar uma promessa sem bloquear – ela retorna um valor *true* ou *false*, de acordo com a promessa estar pronta ou bloqueada.

Invocações assíncronas persistentes ◊ Os mecanismos de invocação assíncronos tradicionais, como as invocações do Mercury e as invocações *de sentido único* do CORBA, são implementados em fluxos TCP e falham, caso seja desfeito o fluxo, isto é, se o enlace de rede cair ou se o *host* destino falhar.

Mas, uma forma mais desenvolvida do modelo de invocação assíncrona, que chamamos de *invocação assíncrona persistente*, está se tornando cada vez mais relevante por causa da operação em modo desconectado. Esse modelo é semelhante ao Mercury em termos das operações de programação que oferece, mas a diferença está em sua semântica de falhas. Um mecanismo de invocação convencional (síncrono ou assíncrono) é projetado para falhar após a ocorrência de determinado número de *timeouts*. Entretanto, *timeouts* de curto prazo freqüentemente não são apropriados quando ocorrem desconexões ou latências muito altas.

Um sistema de invoção assíncrona persistente tenta indefinidamente realizar a invocação até obter êxito ou falha, ou até que o aplicativo cancele a invocação. Um exemplo é a QRPC (*Queued* RPC – RPC enfileirada) do *toolkit* Rover para acesso móvel à informação [Joseph *et al.* 1997].

Conforme seu nome sugere, a QRPC enfileira as requisições de invocação enviadas em um *log* estável, enquanto não há conexão de rede, e agenda seu envio na rede para os servidores, quando há uma conexão. Analogamente, ela enfileira os resultados da invocação, no que podemos considerar que seja a "caixa de correio" de invocação do cliente, até que este reconecte e os colete. As requisições e resultados podem ser compactados ao serem enfileirados, antes de sua transmissão por uma rede de baixa largura de banda.

A QRPC pode tirar proveito de diferentes enlaces de comunicação para enviar uma requisição de invocação e receber a resposta. Por exemplo, uma requisição poderia ser enviada por meio de um enlace GSM, enquanto o usuário estivesse em trânsito, e depois a resposta seria enviada por um enlace Ethernet, quando o usuário conectasse seu dispositivo na intranet corporativa. Em princípio, o sistema de invocação pode armazenar os resultados perto do próximo ponto de conexão esperado do usuário.

O escalonador de rede do cliente funciona de acordo com vários critérios e não envia as invocações necessariamente na ordem FIFO. Os aplicativos podem atribuir prioridades para invocações individuais. Quando uma conexão se torna disponível, a QRPC avalia sua largura de banda e o custo de sua utilização. Ela envia primeiro as invocações de alta prioridade, e pode não enviar todas, caso o enlace seja

lento e dispendioso (como uma conexão remota sem fio), supondo que um enlace mais rápido e barato, como um Ethernet, finalmente se tornará disponível. Analogamente, a QRPC leva a prioridade em conta ao buscar resultados das invocações na caixa de correio em um enlace de largura de banda baixa.

A programação com um sistema de invocação assíncrono (persistente ou não) levanta o problema de como os usuários podem continuar usando os aplicativos em seus dispositivos clientes, enquanto os resultados das invocações ainda não são conhecidos. Por exemplo, o usuário pode estar se perguntando se teve êxito na atualização de um parágrafo em um documento compartilhado, ou se alguém fez uma atualização conflitante, como a exclusão do parágrafo. O Capítulo 15 examinará esse problema.

6.6 Arquiteturas de sistemas operacionais

Nesta seção, examinaremos uma arquitetura de núcleo de sistema operacional conveniente para sistemas distribuídos. Adotaremos uma estratégia de obedecer os primeiros princípios, começando com o requisito de sistema aberto e, tendo isso em mente, examinar as principais arquiteturas de núcleo que propusemos.

Um sistema distribuído aberto deve tornar possível:

- Em cada computador executar apenas o software necessário para desempenhar sua função particular na arquitetura do sistema distribuído; os requisitos do software podem variar entre, por exemplo, assistentes digitais pessoais e computadores servidores dedicados. Carregar módulos redundantes desperdiça recursos de memória.
- Permitir que o software (e o computador) que está implementando um serviço em particular seja trocado independentemente dos outros recursos.
- Possibilitar que sejam fornecidas alternativas para o mesmo serviço, quando isso é necessário para atender diferentes usuários ou aplicativos.
- Introduzir novos serviços sem prejudicar a integridade dos já existentes.

A separação dos *mecanismos* de gerenciamento de recursos das *políticas* de gerenciamento de recursos, que variam de um aplicativo para outro e de um serviço para outro, tem sido um princípio fundamental no projeto de sistemas operacionais há muito tempo [Wulf *et al.* 1974]. Por exemplo, dissemos que um sistema de escalonamento ideal forneceria mecanismos que permitiriam um aplicativo multimídia, como a videoconferência, atender sua demanda de tempo real, e coexistiria com uma aplicação que não usa tempo real, como a navegação web.

De preferência, o núcleo forneceria apenas os mecanismos mais básicos nos quais as tarefas de gerenciamento de recursos gerais seriam executadas em um nó. Os módulos servidores seriam carregados dinamicamente, conforme necessário, para implementar as políticas de gerenciamento de recursos necessário aos aplicativos que estivessem correntemente em execução.

Núcleos monolíticos e micronúcleos ◊ Existem dois exemplos importantes de projeto de núcleo de sistemas operacionais: *monolítica* e *micronúcleo*. Esses projetos diferem principalmente na decisão sobre qual funcionalidade pertence ao núcleo e qual é deixada para os processos servidores que podem ser carregados dinamicamente para execução. Embora os micronúcleos não sejam amplamente usados, é instrutivo entender suas vantagens e desvantagens comparadas com os núcleos normais encontrados atualmente.

O núcleo do sistema operacional UNIX tem sido chamado de *monolítico* (veja a definição no quadro a seguir). Esse termo se destina a sugerir os fatos de que ele é *um bloco maciço* – ele executa todas as funções básicas do sistema operacional e ocupa alguns megabytes de código e dados – e de que *não é diferenciado* – ele é codificado de maneira não modular. O resultado é que, em grande parte, ele é *intratável*: é difícil alterar qualquer componente de software individual para adaptá-lo aos requisitos variáveis. Outro exemplo de núcleo monolítico é o do sistema operacional de rede Sprite [Ousterhout *et al.* 1988]. Um núcleo monolítico pode conter alguns processos servidores que são executados dentro de seu espaço de endereçamento, incluindo servidores de arquivos e alguma capacidade de interligação em rede. O código executado por esses processos faz parte da configuração padrão do núcleo (veja a Figura 6.15).

Monolítico ◊ O Chambers 20th Century Dictionary dá a seguinte definição para *monólito* e *monolítico*. **monólito**, *s.* um pilar ou coluna feita de uma só pedra: tudo que se pareça um monólito em uniformidade, solidez ou em tratamento. – *adj.* monolítico pertencente ou parecido com um monólito: de um estado, uma organização etc., completamente maciço e não diferenciado; intratável por esse motivo.

Em contraste, no caso de um projeto de micronúcleo, o núcleo fornece apenas as abstrações mais básicas, principalmente as de espaços de endereçamento, *threads* e comunicação local entre processos; todos os demais serviços de sistema são fornecidos por servidores carregados dinamicamente, mais precisamente, apenas nos computadores do sistema distribuído que necessitam deles (Figura 6.15). Os clientes acessam esses serviços de sistema usando os mecanismos de invocação baseados em mensagens ao núcleo.

Dissemos anteriormente que os usuários podem não se interessar por sistemas operacionais que não executam seus aplicativos. Mas, além da característica extensível, os projetistas de micronúcleo têm outro objetivo: a simulação de sistemas operacionais padrão, como o UNIX [Armand *et al.* 1989, Golub *et al.* 1990, Härtig *et al.* 1997].

O posicionamento do micronúcleo – em sua forma mais geral – no projeto de sistema distribuído global é mostrado na Figura 6.16. O micronúcleo aparece como uma camada entre a camada de hardware e a camada composta pelos principais componentes de sistema, chamados de *subsistemas*. Se o desempenho for o principal objetivo, em vez da portabilidade, então o *middleware* pode usar diretamente os recursos do micronúcleo. Caso contrário, ele utilizará um subsistema de suporte para uma linguagem em tempo de execução, ou uma interface de sistema operacional de nível mais alto, fornecida por um subsistema de simulação do sistema operacional. Cada um deles, por sua vez, é implementado por uma combinação de procedimentos de biblioteca vinculados aos aplicativos e por um conjunto de serviços executados no micronúcleo.

Pode haver mais de uma interface de chamada de sistema – mais de um "sistema operacional" – apresentada ao programador em uma mesma plataforma. Um exemplo é a implementação do UNIX e do SO/2 no núcleo do sistema operacional distribuído Mach. Note que a simulação do sistema operacional é diferente da *virtualização* da máquina (veja o quadro a seguir).

Comparação ◊ As principais vantagens de um sistema operacional baseado em micronúcleo são sua capacidade de extensão e sua capacidade de impor modularidade através de limites de proteção de memória. Além disso, é mais provável que um núcleo relativamente pequeno esteja menos sujeito a erros do que um maior e mais complexo.

Figura 6.15 Núcleo monolítico e micronúcleo.

Figura 6.16 A função do micronúcleo.

A vantagem de um projeto monolítico é a relativa eficiência com que as operações podem ser invocadas. As chamadas de sistema podem ser mais dispendiosas do que os procedimentos convencionais, mas, mesmo usando as técnicas que examinamos na seção anterior, uma invocação entre espaços de endereçamento em nível de usuário no mesmo nó é ainda mais dispendiosa.

A falta de estrutura nos projetos monolíticos pode ser evitada pelo uso de técnicas de engenharia de software, como a disposição em camadas (usada no MULTICS [Organick 1972]) ou o projeto orientado a objetos (usado, por exemplo, no Choices [Campbell *et al.* 1993]). O Windows emprega uma combinação de ambos [Custer 1998]. Mas o Windows permanece "maciço" e a maior parte de sua funcionalidade não é feita para ser substituída rotineiramente. Mesmo um núcleo grande, modularizado, pode ser difícil de manter; e além disso, normalmente, eles fornecem suporte limitado para um sistema distribuído aberto. Quando os módulos são executados dentro de um mesmo espaço de endereçamento, usando-se uma linguagem, como C ou C++, que geram código eficiente, mas que permitem acessos arbitrários aos dados, é possível que o rigorismo de modularidade seja "quebrado" por programadores que buscam implementações eficientes e que um erro em um módulo corrompa os dados de outro.

Algumas estratégias mistas ◊ Dois dos micronúcleos originais, Mach [Acetta *et al.* 1986] e Chorus [Rozier *et al.* 1990], iniciaram executando servidores apenas como processos de usuário. A modularidade, então, é imposta pelo hardware, por meio de espaços de endereçamento. Onde os servidores fazem acesso direto ao hardware. Isso é viável através de chamadas de sistema especiais fornecidas para esses processos privilegiados, os quais mapeiam registradores de dispositivo e buffers em seus espaços de endereçamentos. O núcleo transforma interrupções em mensagens, permitindo assim que os servidores em nível de usuário as tratem.

Devido aos problemas de desempenho, os projetos de micronúcleo Chorus e Mach finalmente mudaram para permitir que os servidores sejam carregados dinamicamente no espaço de endereçamento do núcleo ou em nível de usuário. Em cada caso, os clientes interagem com os servidores usando as mesmas chamadas de comunicação entre processos. Assim, um desenvolvedor pode depurar um servidor em nível de usuário e, então, quando o desenvolvimento for julgado concluído, permitir que o servidor seja executado dentro do espaço de endereçamento do núcleo para otimizar o desempenho do sistema. Entretanto, nesse caso, tal servidor ameaçará a integridade do sistema, caso se descubra que ele ainda contém erros.

O projeto do sistema operacional SPIN [Bershad *et al.* 1995] refina o problema do compromisso entre eficiência e proteção, empregando recursos de linguagem na proteção. O núcleo e todos os módulos carregados dinamicamente, e enxertados no núcleo, são executados dentro de um único espaço de endereçamento. Mas todos são escritos em uma linguagem fortemente tipada (Modula-3), de modo que eles podem ser mutuamente protegidos. Os domínios de proteção dentro do espaço de endereçamento do núcleo são estabelecidos com o uso de espaços de nomes protegidos. Nenhum módulo enxertado no núcleo pode acessar um recurso, a não ser que tenha sido passada uma referência para ele; e a linguagem Modula-3 impõe a regra de que uma referência só pode ser usada para efetuar as operações permitidas pelo programador.

Virtualização ◊ A *virtualização* acontece onde o hardware de uma única máquina é alocado entre várias máquinas virtuais (imagens de hardware virtual), cada uma executando uma instância separada do sistema operacional. A virtualização começou com a arquitetura do IBM 370, cujo sistema operacional VM pode apresentar várias máquinas virtuais completas para diferentes programas em execução no mesmo computador. Mais recentemente, projetos como o Xen [Barham *et al.* 2003] desenvolveram *monitores de máquina virtual* para PCs comerciais. Um monitor de máquina virtual é uma camada de software entre o hardware e os sistemas operacionais comerciais, como o Linux ou o Windows. Ele permite que um único PC execute simultaneamente qualquer combinação de instâncias não modificadas desses sistemas operacionais e os isole e proteja uns dos outros. Implementações comerciais, como o VMWare, estão disponíveis para PCs e para os nós de *clusters*, permitindo que o trabalho seja alocado a eles de forma flexível.

Em uma tentativa de minimizar as dependências entre os módulos do sistema, os projetistas do SPIN escolheram um modelo baseado em eventos como mecanismo de interação entre os módulos enxertados no espaço de endereçamento do núcleo (veja na Seção 5.4 uma discussão sobre a programação baseada em eventos). O sistema define um conjunto de eventos básicos, como a chegada de um pacote da rede, interrupções de temporizadores, ocorrências de erros de falta página e alterações do estado das *threads*. Os componentes do sistema funcionam registrando-se como tratadores dos eventos que os afetam. Por exemplo, um escalonador se registraria para tratar de eventos semelhantes àqueles que estudamos no sistema de ativações de escalonador, na Seção 6.4.

Sistemas operacionais como o Nemesis [Leslie *et al.* 1996] exploram o fato de que, mesmo no nível do hardware, um espaço de endereçamento não é necessariamente um único domínio de proteção. O núcleo coexiste em um único espaço de endereçamento com todos os módulos de sistema carregados dinamicamente e com os aplicativos. Quando carrega um aplicativo, o núcleo coloca o código e os dados do aplicativo em regiões escolhidas entre as que estão disponíveis no momento da execução. O advento dos processadores com endereçamento de 64 bits tornou os sistemas operacionais de espaço de endereçamento único particularmente atraentes, pois esses processadores suportam espaços de endereçamento muito grandes, que podem acomodar muitos aplicativos.

O núcleo de um sistema operacional de espaço de endereçamento único configura os atributos de proteção em regiões individuais, dentro desse espaço, de forma a restringir o acesso do código em nível de usuário. O código em nível de usuário ainda é executado com o processador em um contexto de proteção em particular (determinado pelas configurações do processador e da unidade de gerenciamento de memória), o qual fornece acesso total às suas próprias regiões e, para outros, somente acesso compartilhado, seletivamente. A economia de um único espaço de endereçamento, comparada com o uso de vários, é que o núcleo nunca precisa esvaziar as caches ao implementar uma transição de domínio.

Alguns projetos de núcleo mais recentes, como o L4 [Härtig *et al.* 1997] e o Exonúcleo [Kaashoek *et al.* 1997], adotam a abordagem de que o que descrevemos como "micronúcleos" ainda contêm políticas demais. O L4 é um projeto de micronúcleo de "segunda-geração" que obriga os módulos de sistema carregados dinamicamente a serem executados em espaços de endereçamento em nível de usuário, mas ele otimiza a comunicação entre processos para compensar os custos envolvidos nisso. Ele diminui grande parte da complexidade do núcleo delegando o gerenciamento de espaços de endereçamento para servidores em nível de usuário. O Exonúcleo adota uma estratégia bastante diferente empregando bibliotecas em nível de usuário, em vez de servidores, para fornecer extensões adicionais. Ele fornece alocação protegida de recursos de nível extremamente baixo, como blocos de disco, e considera que todas as outras funcionalidades de gerenciamento de recursos – até o sistema de arquivos – sejam vinculadas aos aplicativos como bibliotecas.

Nas palavras de um projetista de micronúcleo [Liedtke 1996], "a história do micronúcleo está repleta de boas idéias e becos escuros". Ainda não há um veredicto sobre como projetar uma arquitetura de sistema operacional que seja suficientemente extensível, mas que funcione bem em relação aos projetos monolíticos.

6.7 Resumo

Este capítulo descreveu como o sistema operacional suporta a camada de *middleware* no fornecimento de invocações a recursos compartilhados. O sistema operacional (SO) fornece um conjunto de mecanismos nos quais diversas políticas de gerenciamento de recursos podem ser implementadas para atender aos requisitos locais e para tirar proveito dos aprimoramentos tecnológicos. O SO permite que os servidores encapsulem e protejam os recursos, enquanto deixa que os clientes os compartilhem de forma concorrente. Ele fornece os mecanismos necessários para os clientes invocarem operações em recursos.

Um processo consiste em um ambiente de execução e *threads*: um ambiente de execução é formado por um espaço de endereçamento, interfaces de comunicação e outros recursos locais, como os semáforos; uma *thread* é uma abstração de atividade executada dentro de um ambiente de execução. Os espaços de endereçamento precisam ser grandes e esparsos para suportarem compartilhamento e acesso mapeado a objetos, como por exemplo, arquivos. Novos espaços de endereçamentos podem ser criados a partir de regiões herdadas de processos pais. Uma técnica importante para copiar regiões é a cópia em escrita.

Os processos podem ter múltiplas *threads*, as quais compartilham o ambiente de execução. Os processos *multi-threadeds* nos permitem obter concorrência relativamente barata e explorar o paralelismo real disponível nos multiprocessadores. Eles são úteis tanto para clientes como para servidores. As implementações recentes de *threads* permitem o escalonamento em duas camadas: o núcleo dá acesso a vários processadores, enquanto o código em nível de usuário trata dos detalhes das políticas de escalonamento.

O sistema operacional fornece primitivas básicas para passagem de mensagem e mecanismos para comunicação por meio de memória compartilhada. A maioria dos núcleos inclui comunicação de rede como um recurso básico; outros fornecem apenas comunicação local e deixam a comunicação de rede para os servidores, os quais podem implementar uma variedade de protocolos de comunicação. Esse é um compromisso de desempenho em relação à flexibilidade.

Discutimos as invocações remotas e levamos em conta a diferença entre sobrecargas decorrentes diretamente da rede e as ocasionadas pela execução do código do sistema operacional. Descobrimos que a proporção do tempo total relacionado ao software é relativamente grande para uma invocação nula, mas diminui proporcionalmente com o tamanho dos argumentos da requisição. As principais sobrecargas envolvidas em uma invocação, que são candidatas à otimização, são o empacotamento, a cópia de dados, a inicialização de pacotes, o escalonamento e o chaveamento de contexto de *threads*, e o protocolo de controle de fluxo utilizado. A invocação entre espaços de endereçamento em um mesmo computador é um caso especial importante, e descrevemos as técnicas de gerenciamento de *threads* e passagem de parâmetros usadas na RPC leve.

Existem duas estratégias principais de arquitetura do núcleo: núcleos monolíticos e micronúcleos. A principal diferença entre elas reside em onde é traçada a linha entre o gerenciamento de recursos pelo núcleo e o gerenciamento de recursos realizado por servidores carregados dinamicamente (e, normalmente, em nível de usuário). Um micronúcleo precisa suportar pelo menos a noção de processo e comunicação entre processos. Ele suporta subsistemas de simulação do sistema operacional, assim como subsistemas de suporte à linguagem e outros serviços, como aqueles para processamento em tempo real.

Exercícios

6.1 Discuta cada uma das tarefas de encapsulamento, processamento concorrente, proteção, transformação de nomes, comunicação de parâmetros e resultados, e escalonamento, no caso do serviço de arquivos do UNIX (ou de outro núcleo com que você esteja familiarizado). *página 201–203*

6.2 Por que algumas interfaces de sistema são implementadas por chamadas de sistema dedicadas (ao núcleo) e outras por chamadas de sistema baseadas em mensagens? *página 201–203*

6.3 Smith decide que qualquer *thread* em seus processos deve ter sua própria pilha protegida – todas as outras regiões em um processo seriam totalmente compartilhadas. Isso faz sentido? *página 205*

6.4 Os tratadores de um sinal (interrupção de software) devem pertencer a um processo ou a uma *thread*? *página 205*

6.5 Discuta o problema da atribuição de nomes aplicada às regiões de memória compartilhada. *página 206–207*

6.6 Sugira um esquema para balancear a carga em um conjunto de computadores. Você deve discutir:

i) quais requisitos de usuário ou sistema são atendidos por tal esquema;

ii) para quais categorias de aplicativos ele é conveniente;

iii) como medir a carga e com que precisão; e

iv) como monitorar a carga e escolher a localização de um novo processo. Suponha que os processos não podem migrar.

Como seu projeto seria afetado se os processos pudessem migrar entre computadores? Você esperaria que a migração de processo tivesse um custo significativo? *página 207–208*

6.7 Explique a vantagem da cópia de região com *cópia na escrita* para o UNIX, onde uma chamada *fork* normalmente é acompanhada de uma chamada *exec*. O que deve acontecer se uma região que foi copiada usando *cópia na escrita* for ela própria copiada? *página 209–210*

6.8 Um servidor de arquivos usa cache e obtém uma taxa de acertos de 80%. As operações no servidor custam 5 ms de tempo da CPU, quando o servidor encontra o bloco solicitado na cache, e leva mais 15 ms de tempo de E/S de disco, quando não encontra. Explicando todas as suposições que fizer, faça uma estimativa da capacidade de *throughput* (pedidos médios/seg), se ele for:

i) *single-threaded*;

ii) usar duas *threads* executando em um único processador;

iii) usar duas *threads* executando em um computador com dois processadores. *página 209–211*

6.9 Compare a arquitetura *multi-threaded* de pool de trabalhadores, com a arquitetura de uma *thread* por pedido. *página 211–212*

6.10 Quais operações de *thread* têm custo mais significativo? *página 212–213*

6.11 Um *spin lock* (veja Bacon [2002]) é uma variável booleana acessada por meio de uma instrução *test-and-set* atômica, que é usada para obter exclusão mútua. Você usaria *spin lock* para obter exclusão mútua entre *threads* em um computador com um só processador? *página 216–217*

6.12 Explique o que o núcleo deve fornecer para uma implementação de *threads* em nível de usuário, como o Java no UNIX. *página 217*

6.13 Os erros de falta de página representam um problema para implementações de *threads* em nível de usuário? *página 217*

6.14 Explique os fatores que motivam a estratégia de escalonamento mista do projeto de "ativações do escalonador" (em vez de escalonamento em nível de usuário ou em nível de núcleo pura). *página 217–218*

6.15 Por que um pacote de *threads* deve estar interessado nos eventos que bloqueiam ou desbloqueiam uma *thread*? Por que ele deve estar interessado no evento de preempção pendente de um processador virtual? (Dica: outros processadores virtuais podem continuar a ser alocados.) *página 218–219*

6.16 O tempo de transmissão na rede é responsável por 20% de uma RPC nula e por 80% de uma RPC que transmite 1024 bytes de usuário (menor do que o tamanho de um pacote de rede). Qual será a porcentagem dos tempos dessas duas operações se a rede for migrada de 10 megabits/segundo para 100 megabits/segundo? *página 221–222*

6.17 Uma RMI "nula", que não recebe parâmetros, chama um procedimento vazio e não retorna valores, "travando" o chamador por 2,0 milissegundos. Explique o que contribui para esse tempo.

No mesmo sistema RMI, cada 1K de dados do usuário adiciona mais 1,5 milissegundos. Um cliente deseja buscar 32K de dados de um servidor de arquivos. Ele deve usar uma RMI de 32K ou 32 RMIs de 1K? *página 221–222*

6.18 Quais fatores identificados no custo de uma invocação remota também aparecem na passagem de mensagens? *página 222–223*

6.19 Explique como uma região compartilhada poderia ser usada por um processo para ler dados gravados pelo núcleo. Inclua em sua explicação o que seria necessário para a sincronização. *página 223–224*

6.20 i) Um servidor ativado por chamadas de procedimento leves pode controlar o grau de concorrência dentro dele?

ii) Explique por que e como um cliente é impedido de chamar código arbitrário dentro de um servidor na RPC leve.

iii) A LRPC expõe os clientes e servidores a maiores riscos de interferência mútua do que a RPC convencional (dado o compartilhamento da memória)? *página 224–225*

6.21 Um cliente faz RMIs em um servidor. O cliente leva 5 ms para computar os argumentos de cada requisição e o servidor leva 10 ms para processar cada requisição. O tempo de processamento do SO local para cada operação *send* ou *receive* é de 0,5 ms e o tempo da rede para transmitir cada requisição ou resposta é de 3 ms. O empacotamento ou desempacotamento leva 0,5 ms por mensagem.

Faça uma estimativa do tempo que leva para o cliente gerar e retornar 2 pedidos (i) se for *single-threaded* e (ii) se tiver duas *threads* que podem fazer pedidos concorrentemente em um único processador. Há necessidade de RMI assíncrona, se os processos forem *multi-threaded*? *página 226–227*

6.22 Explique o que é política de segurança e quais são os mecanismos correspondentes no caso de um sistema operacional multiusuário como o UNIX. *página 228–229*

6.23 Explique os requisitos de ligação de programa que devem ser satisfeitos se um servidor precisa ser carregado dinamicamente no espaço de endereçamento do núcleo e como eles diferem do caso da execução de um servidor em nível de usuário. *página 229–230*

6.24 Como uma interrupção poderia ser comunicada a um servidor em nível de usuário? *página 231–232*

6.25 Em certo computador, estimamos que, independentemente do SO que execute, o escalonamento de *threads* custa cerca de 50 ms, uma chamada de procedimento nula custa 1 ms, uma troca de contexto para o núcleo custa 20 ms e uma transição de domínio custa 40 ms. Para o Mach e para o SPIN, faça uma estimativa do custo para um cliente chamar um procedimento nulo carregado dinamicamente. *página 231–232*

7 Segurança

Há uma necessidade generalizada de medidas para garantir a privacidade, a integridade e a disponibilidade dos recursos em sistemas distribuídos. Os ataques contra a segurança assumem as formas de intromissão, mascaramento, falsificação e negação de serviço. Os projetistas de sistemas distribuídos seguros devem enfrentar interfaces de serviço expostas e redes inseguras em um ambiente onde os invasores provavelmente têm conhecimento dos algoritmos usados.

A criptografia fornece a base para a autenticação de mensagens, assim como sua privacidade e integridade; são necessários protocolos de segurança cuidadosamente projetados para explorá-la. A escolha dos algoritmos de criptografia e o gerenciamento de chaves são fundamentais para a eficácia, desempenho e utilidade dos mecanismos de segurança. A criptografia de chave pública torna fácil a distribuição de chaves de criptografia, mas seu desempenho é inadequado para a criptografia de grandes volumes de dados, situação onde a criptografia de chave secreta é mais conveniente. Protocolos mistos, como o TLS (*Transport Layer Security*), estabelecem um canal seguro usando criptografia de chave pública e negociam chaves secretas, depois essas chaves são utilizadas nas trocas de dados subseqüentes.

A informação digital pode ser assinada, produzindo certificados digitais. Os certificados permitem que seja estabelecida confiança entre usuários e organizações.

7.1 Introdução

Na Seção 2.3.3, apresentamos um modelo simples para examinar os requisitos de segurança em sistemas distribuídos. Concluímos que a necessidade de mecanismos de segurança em sistemas distribuídos surge do desejo de compartilhar recursos. (Geralmente, os recursos que não são compartilhados podem ser protegidos por meio de seu isolamento do acesso externo.) Se considerarmos os recursos compartilhados como objetos, então o requisito é proteger contra todas as formas de ataque concebíveis, todos os processos que encapsulam objetos compartilhados e todos os canais de comunicação usados para interagir com eles. O modelo apresentado na Seção 2.3.3 forneceu um bom ponto de partida para a identificação dos requisitos de segurança. Eles podem ser resumidos como segue:

- Os processos encapsulam recursos (tanto objetos em nível de linguagem de programação como recursos definidos pelo sistema) e permitem que os clientes os acessem por meio de interfaces. Os principais (usuários ou outros processos) são autorizados a trabalhar com os recursos. Os recursos devem ser protegidos contra acesso não autorizado (Figura 2.13).
- Os processos interagem por meio de uma rede que é compartilhada por muitos usuários. Inimigos (invasores) podem acessar a rede. Eles podem copiar, ou tentar ler, qualquer mensagem transmitida pela rede, e podem injetar na rede mensagens arbitrárias endereçadas para qualquer destino, e dando a entender que são provenientes de qualquer fonte (Figura 2.14).

A necessidade de proteger a integridade e a privacidade da informação, e de outros recursos pertencentes a indivíduos e organizações é generalizada, tanto no mundo físico quanto no digital. Ela surge do desejo de compartilhar recursos. No mundo físico, as organizações adotam *políticas de segurança* que proporcionam o compartilhamento de recursos dentro de limites especificados. Por exemplo, uma empresa pode permitir a entrada em seus prédios para seus funcionários e visitantes autorizados. Uma política de segurança para documentos pode especificar grupos de funcionários que podem acessar classes de documentos, ou pode ser definida para documentos e usuários individuais.

As políticas de segurança são impostas com a ajuda de *mecanismos de segurança*. Por exemplo, o acesso a um prédio pode ser controlado por uma recepcionista que distribui crachás para os visitantes autorizados, reforçado pela presença de um guarda de segurança e por travas eletrônicas de portas. O acesso a documentos em papel normalmente é controlado pela ocultação e pela distribuição restrita. No mundo eletrônico, a distinção entre políticas e mecanismos de segurança é igualmente importante; sem ela, seria difícil determinar se um sistema em particular é seguro. As políticas de segurança são independentes da tecnologia usada, assim como, por si só, o uso de uma tranca em uma porta não garante a segurança de um prédio, a não ser que exista uma política para sua utilização (por exemplo, dizendo que a porta será trancada quando ninguém estiver guardando a entrada). Analogamente, os mecanismos de segurança que vamos descrever não garantem em si a segurança de um sistema. Na Seção 7.1.2 esboçaremos os requisitos de segurança em vários cenários simples de comércio eletrônico, ilustrando a necessidade de políticas nesse contexto.

O assunto deste capítulo é o uso de mecanismos para proteger dados e outros recursos em sistemas distribuídos e ao mesmo tempo possibilitar interações entre os computadores envolvidos pelas políticas de segurança. Os mecanismos que vamos descrever são projetados para fazer valer as políticas de segurança contra os ataques mais determinados.

O papel da criptografia ◊ A criptografia digital fornece a base para a maioria dos mecanismos de segurança de computador, mas é importante notar que segurança do computador e criptografia são assuntos distintos. Criptografia é a arte de codificar informação em um formato que apenas os destinatários apropriados possam acessá-la. A criptografia também pode ser empregada para fornecer uma prova da autenticidade de informações de maneira semelhante ao uso de assinaturas nas transações convencionais.

A criptografia tem uma história longa e fascinante. A necessidade militar de comunicação segura, e a necessidade correspondente de um inimigo de interceptá-la e decodificá-la, levou ao investimento de muito esforço intelectual por parte de alguns dos melhores matemáticos de sua época. Os leitores que estiverem interessados em explorar essa história encontrarão uma leitura cativante nos livros sobre o assunto de David Kahn [Kahn 1967, 1983, 1991] e Simon Singh [Singh 1999]. Whitfield Diffie, um dos inventores da criptografia de chave pública, escreveu com conhecimento de causa sobre a história e a política recentes da criptografia [Diffie 1988, Diffie e Landau 1998].

Mas foi apenas recentemente que a criptografia saiu do esconderijo a que era obrigada a ficar anteriormente pelos poderes políticos e militares que controlavam seu desenvolvimento e seu uso. Atualmente ela é assunto livre de pesquisas feitas por uma comunidade grande e ativa, com resultados publicados em muitos livros, periódicos e conferências. A publicação do livro *Applied Cryptography*, de Schneier [Schneier 1996], foi um marco na abertura do conhecimento no setor. Esse foi o primeiro livro a publicar muitos algoritmos importantes com código-fonte – um passo corajoso, pois quando a primeira edição foi lançada, em 1994, o status jurídico de tal publicação não estava claro. O livro de Schneier continua sendo a referência definitiva sobre a maioria dos aspectos da criptografia moderna. Um livro mais recente, do qual Schneier é co-autor [Ferguson e Schneier 2003], fornece uma introdução excelente sobre criptografia e computador, e contém um panorama discursivo de praticamente todos os algoritmos e técnicas importantes em uso corrente, incluindo vários publicados desde o lançamento do primeiro livro de Schneier. Além disso, Menezes *et al.* [1997] fornecem um bom manual prático, com uma forte base teórica, e a Network Security Library [www.secinf.net] é uma excelente fonte on-line de conhecimento prático e experiência.

O livro *Security Engineering* de Ross Anderson [Anderson 2001] também é importante. Ele está repleto de lições práticas sobre o projeto de sistemas seguros, extraídas de situações reais e falhas na segurança de sistemas.

Essa nova abertura é em grande parte o resultado do tremendo crescimento no interesse por aplicações não militares da criptografia e dos requisitos de segurança dos sistemas computacionais distribuídos. Isso resultou na existência, pela primeira vez, de uma comunidade independente de pesquisadores em criptografia fora do domínio militar.

Ironicamente, a abertura da criptografia para acesso e uso público resultou em uma grande melhoria nas técnicas de criptografia, tanto em seu poder de suportar ataques de inimigos quanto na conveniência com que elas podem ser implantadas. A criptografia de chave pública é um dos frutos dessa abertura. Como outro exemplo, notamos que o algoritmo de criptografia DES, que foi adotado e usado pelos órgãos militares e governamentais dos EUA, era inicialmente um segredo militar. Sua publicação final e os esforços bem-sucedidos de violá-la resultaram no desenvolvimento de algoritmos de criptografia de chave secreta muito mais poderosos.

Outra reviravolta útil foi o desenvolvimento de uma terminologia e de uma estratégia comuns. Um exemplo desta última é a adoção de um conjunto de nomes familiares para os protagonistas (principais) envolvidos nas transações que precisam se tornar seguras. O uso de nomes de pessoas para principais e invasores ajuda a esclarecer e a reavivar as descrições dos protocolos de segurança e dos ataques em potencial sobre eles, o que é um passo importante na identificação de suas deficiências. Os nomes mostrados na Figura 7.1 são usados extensivamente na literatura sobre segurança e vamos usá-los livremente aqui. Não conseguimos descobrir suas origens; a ocorrência mais antiga que conhecemos está no artigo original sobre criptografia de chave pública RSA [Rivest *et al.* 1978]. Um divertido comentário sobre seu uso pode ser encontrado em Gordon [1984].

7.1.1 Ameaças e ataques

Algumas ameaças são óbvias – por exemplo, na maioria dos tipos de rede local é fácil construir e executar um programa em um computador conectado a rede para obter cópias das mensagens transmitidas entre outros computadores. Outras ameaças são mais sutis – se os clientes não conseguem autenticar servidores, um programa poderia se instalar no lugar de um servidor de arquivos autêntico e, com isso, obter cópias de informações confidenciais que os clientes enviam inadvertidamente para ele armazenar.

Além do perigo da perda, ou de danos na informação ou nos recursos, por meio de violações diretas, pedidos indenizatórios podem ser feitos contra o proprietário de um sistema que não esteja totalmente seguro. Para evitar tais reivindicações, o proprietário deve estar em uma posição favorável para invalidá-las, mostrando que seu sistema é seguro contra tais violações ou produzindo um registro de todas as transações durante o período em questão. Um caso comum é o problema do saque fantasma em caixas eletrônicos. A melhor resposta que um banco pode dar a tal reivindicação é fornecer um registro da transação com a assinatura digital do correntista, de maneira que não possa ser falsificada por outra pessoa.

Alice	Primeiro participante
Bob	Segundo participante
Carol	Participante em protocolos de três ou quatro partes
Dave	Participante em protocolos de quatro partes
Eve	Bisbilhoteiro
Mallory	Invasor mal-intencionado
Sara	Um servidor

Figura 7.1 Nomes comumente usados para protagonistas nos protocolos de segurança.

O principal objetivo da segurança é restringir o acesso às informações e recursos apenas para os principais que estejam autorizados a ter acesso. As ameaças contra a segurança caem em três classes amplas:

Vazamento – a aquisição de informações por destinatários não autorizados;

Falsificação – a alteração não autorizada da informação;

Vandalismo – interferência na operação correta de um sistema, sem ganho para o invasor.

Os ataques nos sistemas distribuídos dependem da obtenção do acesso aos canais de comunicação existentes, ou do estabelecimento de novos canais que sejam mascarados como conexões autorizadas. (Usamos o termo *canal* para nos referirmos a qualquer mecanismo de comunicação entre processos.) Os métodos de ataque podem ainda ser classificados de acordo com a maneira como um canal é empregado inadequadamente:

Intromissão – obter cópias de mensagens sem autorização.

Mascaramento – enviar ou receber mensagens usando a identidade de outro principal, sem sua autorização.

Falsificação de mensagem – interceptar mensagens e alterar seu conteúdo, antes de passá-las para o destinatário desejado. O *ataque do homem no meio* (*man-in-the-middle*) é uma forma de falsificação de mensagem na qual o invasor intercepta a primeira mensagem em uma troca de chaves de criptografia para estabelecer um canal seguro. O invasor substitui por chaves comprometidas que permitem a ele decodificar as mensagens subseqüentes, antes de recodificá-las com as chaves corretas e passá-las adiante.

Reprodução – armazenar mensagens interceptadas e enviá-las em uma data posterior. Este ataque pode ser eficiente mesmo com mensagens autenticadas e codificadas.

Negação de serviço – saturar um canal, ou outro recurso, com mensagens para impedir (negar) o acesso de outras pessoas.

Teoricamente, esses são os perigos; mas como os ataques são realizados na prática? Os ataques bem-sucedidos dependem da descoberta de brechas na segurança dos sistemas. Infelizmente, elas são muito comuns nos sistemas atuais, e não são necessariamente pouco conhecidas. Cheswick e Bellovin [1994] identificam 42 deficiências que consideram riscos sérios em sistemas e componentes normalmente usados na Internet. Elas variam desde adivinhação de senha até ataques nos programas que executam o protocolo de sincronização de tempo ou no envio e recepção de correspondência eletrônica. Algumas delas têm levado a ataques bem-sucedidos e bastante divulgados [Stoll 1989, Spafford 1989] e muitas têm sido exploradas para propósitos nocivos ou criminosos.

Quando a Internet e os sistemas conectados a ela foram projetados, a segurança não era uma prioridade. Os projetistas provavelmente não tinham nenhuma idéia da escala com a qual a Internet cresceria, e o projeto básico de sistemas operacionais, como o UNIX, é anterior ao advento das redes de computadores. Conforme veremos, a incorporação de medidas de segurança precisa ser cuidadosamente pensada no estágio de projeto básico, e o material deste capítulo se destina a fornecer a base para tal pensamento.

Focalizamos as ameaças aos sistemas distribuídos que surgem da exposição de seus canais de comunicação e suas interfaces. Para muitos sistemas, essas são as únicas ameaças que precisam ser consideradas (além daquelas que surgem de erro humano – os mecanismos de segurança não podem atuar contra uma senha mal escolhida, ou que seja negligentemente revelada). Entretanto, nos sistemas que incluem programas móveis, e sistemas cuja segurança é particularmente sensível ao vazamento de informações, existem mais fontes de ameaças.

Ameaças em código móvel ◊ Várias linguagens de programação desenvolvidas recentemente foram projetadas para permitir que programas sejam carregados em um processo a partir de um servidor remoto e depois executados localmente. Nesse caso, as interfaces internas e os objetos dentro de um processo em execução podem ficar expostos a ataques feitos a partir de um código móvel.

Java é a linguagem mais usada que permite esse tipo de facilidade, e os projetistas prestam considerável atenção ao projeto, à construção da linguagem e aos mecanismos de carga remota, em um esforço para restringir a exposição (o modelo *sandbox* de proteção contra código móvel).

A máquina virtual do Java, a JVM (*Java Virtual Machine*), foi projetada tendo o código móvel em vista. Ela fornece para cada aplicativo seu próprio ambiente de execução. Cada ambiente tem um gerenciador de segurança que determina quais recursos estão disponíveis para esse aplicativo. Por exemplo, o gerenciador de segurança pode impedir que um aplicativo leia e escreva em arquivos, ou que possua acesso limitado às conexões de rede. Uma vez configurado, o gerenciador de segurança não pode ser substituído. Quando um usuário executa um programa, como um navegador, que carrega código móvel por *download* para ser executado localmente, em seu nome, ele não tem nenhum bom motivo para acreditar que o código vá se comportar de maneira responsável. Na verdade, existe o perigo de fazer *download* e executar um código nocivo que remova arquivos ou acesse informações privadas. Para proteger os usuários contra código não confiável, a maioria dos navegadores especifica que os *applets* não podem acessar arquivos locais, impressoras ou soquetes de rede. Alguns aplicativos baseados em código móvel são capazes de atribuir vários níveis de confiança no código carregado por *download*. Nesse caso, os gerenciadores de segurança são configurados de forma a dar mais direitos de acessos aos recursos locais.

A JVM adota mais duas medidas para proteger o ambiente local:

1. as classes carregadas por *download* são armazenadas separadamente das classes locais, impedindo que elas substituam as classes locais com versões espúrias;
2. é verificada a validade dos *bytecodes*. Um *bytecode* Java válido é composto de instruções da máquina virtual Java de um conjunto específico. As instruções também são verificadas para garantir que não produzirão certos erros quando o programa for executado, como o acesso a endereços de memória inválidos.

A segurança da linguagem Java foi o assunto de muitas pesquisas, no curso das quais se tornou claro que os mecanismos originais adotados não estavam livres de brechas [McGraw e Felden 1999]. As brechas identificadas foram corrigidas e o sistema de proteção Java foi refinado para permitir que código móvel acessasse recursos locais, quando autorizado a fazer isso [java.sun.com V].

Apesar da inclusão de mecanismos de verificação de tipo e de validação de código, os mecanismos de segurança incorporados nos sistemas de código móvel ainda não possuem eficientemente o mesmo nível de confiança que aqueles usados para proteger canais e interfaces de comunicação. Isso porque a construção de um ambiente para a execução de programas apresenta muitas oportunidades de erros, e é difícil ter certeza de que todos foram evitados. Volpano e Smith [1999] apontaram que uma estratégia alternativa, baseada em provas de que o comportamento do código móvel é sadio, poderia ser uma solução melhor.

Vazamento de informações ◊ Se a transmissão de uma mensagem entre dois processos puder ser observada, uma informação poderá ser colhida a partir de sua simples existência – por exemplo, uma avalanche de mensagens para um operador sobre uma ação em particular poderia indicar um alto nível de negociação dessa ação. Existem formas muito mais sutis de vazamento de informações, algumas nocivas e outras provenientes de erro involuntário. O potencial de vazamento surge quando os resultados de uma computação podem ser observados. Nos anos 70, foi feito um trabalho na prevenção desse tipo de ameaça à segurança [Denning e Denning 1977]. A estratégia adotada foi atribuir níveis

de segurança às informações e aos canais, e analisar o fluxo de informações nos canais com o objetivo de garantir que informações de mais alto nível não pudessem fluir para canais de mais baixo nível. Um método para o controle seguro de fluxos de informação foi descrito pela primeira vez por Bell e LaPadula [1975]. A extensão dessa estratégia para sistemas distribuídos com desconfiança mútua entre os componentes é o assunto de pesquisa recente [Myers e Liskov 1997].

7.1.2 Tornando transações eletrônicas seguras

Muitos usos da Internet na indústria, no comércio e em várias outras áreas, envolvem transações que dependem fundamentalmente da segurança. Por exemplo:

E-mail: embora os sistemas de e-mail não tenham incluído originalmente suporte para segurança, existem muitos empregos de e-mail no qual o conteúdo das mensagens deve ser mantido em segredo (por exemplo, no envio de um número de cartão de crédito), ou que o conteúdo e o remetente de uma mensagem devem ser autenticados (por exemplo, ao enviar um lance por e-mail em um leilão). A segurança com criptografia baseada nas técnicas descritas neste capítulo é, agora, incluída em muitos clientes de correio eletrônico.

Aquisição de bens e serviços: atualmente, tais transações são muito comuns. Os compradores escolhem bens e pagam por eles usando a web; os bens são entregues por meio de um mecanismo de distribuição apropriado. Software e outros produtos digitais (como gravações e vídeos) podem ser distribuídos por meio de *download* pela Internet. Bens materiais, como livros, CDs e quase todos os outros tipos de produtos, também são comercializados por vendedores na Internet; eles são fornecidos por meio de um serviço de distribuição.

Transações bancárias: atualmente, os bancos eletrônicos oferecem aos usuários praticamente todas as facilidades fornecidas pelos bancos convencionais. Eles podem consultar seus saldos e extratos, transferir dinheiro entre contas, estabelecer pagamentos automáticos regulares, etc.

Microtransações: a Internet se presta ao fornecimento de pequenos volumes de informação e outros serviços para muitos clientes. Por exemplo, atualmente, a maioria das páginas da web não é paga, mas o desenvolvimento da web como um meio de publicação de alta qualidade seguramente depende do grau com que os fornecedores de informação podem receber pagamentos dos consumidores dessa informação. O uso da Internet para voz e videoconferência fornece outro exemplo de serviço que provavelmente será fornecido somente quando for pago pelos usuários finais. O preço de tais serviços pode chegar a apenas uma fração de um centavo e os custos e taxas relacionados com o pagamento devem ser proporcionalmente baixas. Em geral, esquemas baseados no envolvimento de um banco, ou operadora de cartão de crédito, para cada transação, não podem conseguir isso.

Transações como essas só podem ser realizadas com segurança quando são protegidas por políticas e mecanismos de segurança apropriados. O comprador deve ser protegido contra a exposição de seus códigos de crédito (números de cartão) durante a transmissão e contra um vendedor fraudulento, que receba o pagamento sem nenhuma intenção de fornecer os bens. Os vendedores devem receber o pagamento antes de liberar os bens e, para produtos carregados por *download*, eles devem garantir que apenas o cliente que pagou obtenha os dados de forma que possam ser utilizados. A proteção exigida deve ser obtida a um custo que seja razoável em comparação com o valor da transação.

Políticas de segurança para vendedores e compradores na Internet levam aos seguintes requisitos para tornar seguras as compras na web:

1. Autenticação do vendedor para o comprador, para que este possa ter confiança de que está em contato com um servidor operado pelo vendedor com quem pretende negociar.
2. Impedir que o número do cartão de crédito, e outros detalhes do pagamento do comprador, caiam nas mãos de outra pessoa e garantir que eles sejam transmitidos do comprador para o vendedor sem alteração.
3. Se os bens estiverem em uma forma conveniente para *download*, certificar-se de que seu conteúdo seja enviado para o comprador, sem alteração e sem exposição para terceiros.

A identidade do comprador normalmente não é exigida pelo vendedor (exceto para o propósito de enviar bens, no caso de não poderem ser carregados por *download*). O vendedor desejará verificar se o comprador tem fundos suficientes para pagar pela compra, mas isso normalmente é feito pela exigência do pagamento pelo banco do comprador, antes do envio dos bens.

As necessidades de segurança das transações bancárias usando uma rede aberta são semelhantes às das transações de compra, com o comprador, como correntista, e o banco como vendedor, mas aqui certamente há necessidade de:

4. Autenticar a identidade do correntista para o banco, antes de dar a ele acesso à sua conta.

Note que, nessa situação, é importante para o banco garantir que o correntista não possa negar sua participação em uma transação. *Não-repúdio* é o nome dado a esse requisito.

Além dos requisitos anteriores, que são impostos pelas políticas de segurança, existem alguns requisitos de sistema. Eles surgem da própria escala da Internet, que torna impraticável exigir que os compradores estabeleçam relacionamentos especiais com os vendedores (por exemplo, registrando chaves de criptografia para uso posterior, etc.). Deve ser possível para um comprador concluir uma transação segura com um vendedor, mesmo que não tenha havido nenhum contato anterior entre o comprador e o vendedor, e sem o envolvimento de terceiros. Técnicas como o uso de *cookies* – registros de transações anteriores armazenados no *host* cliente do usuário – têm deficiências de segurança óbvias; *desktop* e *hosts* móveis são freqüentemente localizados em ambientes físicos inseguros.

Devido à importância da segurança para o comércio na Internet, e ao rápido crescimento desse comércio, optamos por ilustrar o uso de técnicas de segurança com criptografia descrevendo, na Seção 7.6, o protocolo de segurança padrão *de facto* usado na maior parte do comércio eletrônico – o TLS (*Transport Layer Security*). Uma descrição do Millicent, um protocolo especificamente projetado para microtransações, pode ser encontrada no endereço www.cdk4.net/security.

O comércio na Internet é uma aplicação importante das técnicas de segurança, mas certamente não é a única. Ela é necessária onde quer que computadores sejam usados por indivíduos ou organizações para armazenar e comunicar informações importantes. O uso de e-mail codificado para comunicação privada entre indivíduos é um caso questionável que tem sido o assunto de considerável discussão política. Vamos nos referir a esse debate na Seção 7.5.2.

7.1.3 Projetando sistemas seguros

Nos últimos anos, passos enormes foram dados no desenvolvimento de técnicas de criptografia e em sua aplicação, apesar do projeto de sistemas seguros continuar sendo uma tarefa inerentemente difícil. No centro desse dilema está o fato de que o objetivo do projetista é excluir *todos* os ataques e brechas possíveis. A situação é semelhante à do programador cujo objetivo deve ser excluir todos os erros de seu programa. Em nenhum dos casos há um método concreto para garantir os objetivos durante o projeto. Projeta-se para os melhores padrões disponíveis e aplica-se análise e verificações informais. Uma vez concluído o projeto, uma opção é a validação formal. O trabalho feito na validação formal dos protocolos de segurança produziu alguns resultados importantes [Lampson *et al.* 1992, Schneider 1996, Abadi e Gordon 1999]. Uma descrição de um dos primeiros passos nessa direção, a lógica de autenticação BAN [Burrows *et al.* 1990] e sua aplicação, pode ser encontrada no endereço www.cdk4.net/security.

Segurança significa evitar desastres e minimizar os acidentes. Ao se projetar tendo em vista a segurança é necessário supor o pior. O quadro a seguir mostra um conjunto de suposições e diretrizes de projeto bastante úteis. Essas suposições sustentam o pensamento que há por trás das técnicas que vamos descrever neste capítulo.

Para demonstrar a validade dos mecanismos de segurança empregados em um sistema, os projetistas do sistema devem primeiro construir uma lista de ameaças – os métodos pelos quais as políticas de segurança poderiam ser violadas – e mostrar que cada uma delas é evitada pelos mecanismos empregados. Essa demonstração pode assumir a forma de argumento informal, ou melhor, pode assumir a forma de uma prova lógica.

Provavelmente, nenhuma lista de ameaças será exaustiva; portanto, métodos de auditoria também devem ser usados para detectar violações em aplicações sensíveis à segurança. Eles são muito simples de implementar, caso seja sempre gravado um registro (*log*) seguro das ações em um sistema sensível quanto a segurança, com detalhes dos usuários que estão executando as ações e suas permissões.

Suposições de pior caso e diretrizes de projeto

As interfaces são expostas: os sistemas distribuídos são compostos de processos que oferecem serviços ou compartilham informações. Suas interfaces de comunicação são necessariamente abertas (para permitir que novos clientes as acessem) – um invasor pode enviar uma mensagem para qualquer interface.

As redes são inseguras: por exemplo, as fontes de mensagem podem ser falsificadas – as mensagens podem parecer terem sido enviadas por Alice, quando na verdade foram enviadas por Mallory. Pode haver *spoofing* dos endereços dos *hosts* – Mallory pode se conectar na rede com o mesmo endereço de Alice e receber cópias das mensagens destinadas a ela.

Limite do tempo de vida e escopo de cada segredo: quando uma chave secreta é gerada pela primeira vez, podemos ter certeza de que ela não foi comprometida. Quanto mais a utilizamos e mais ela se torna conhecida, maior é o risco. O uso de segredos, como senhas e chaves secretas compartilhadas, deve ter um período de validade (limite do tempo de vida) e o compartilhamento deve ser restrito.

Algoritmos e código de programa estão disponíveis para os invasores: quanto maior e mais amplamente distribuído é um segredo, maior é o risco de sua exposição. Os algoritmos de criptografia secretos são totalmente inadequados para os ambientes atuais de redes de larga escala. A melhor prática é publicar os algoritmos usados para criptografia e autenticação, e contar com o segredo das chaves de criptografia. Isso ajuda a garantir que os algoritmos sejam poderosos, lançando-os abertamente para o escrutínio de outras pessoas.

Os invasores podem ter acesso a grandes recursos: o custo do poder de computação está diminuindo rapidamente. Devemos supor que os invasores terão acesso aos maiores e mais poderosos computadores que serão projetados durante o ciclo de vida de um sistema e, então, acrescentar algumas ordens de grandeza para admitir evoluções inesperadas.

Minimizar a base confiável: as partes de um sistema que são responsáveis pela implementação de sua segurança e de *todos os componentes de hardware e software com os quais elas contam*, precisam ser confiáveis – isso é freqüentemente referido como *base de computação confiável*. Qualquer defeito, ou erro de programação, nessa base confiável pode produzir deficiências de segurança; portanto, devemos ter como objetivo minimizar seu tamanho. Por exemplo, não se deve confiar em programas aplicativos para proteger dados de seus usuários.

Um *log* de segurança conterá uma seqüência de registros com indicação de tempo das ações dos usuários. No mínimo, os registros incluirão a identidade de um principal, a operação executada (por exemplo, excluir arquivo, atualizar registro de conta), a identidade do objeto que sofreu a operação e uma indicação de tempo. Onde violações particulares são suspeitas, os registros podem ser ampliados para incluir utilização de recurso físico (largura de banda de rede, periféricos) ou o processo de registro pode ter como objetivo operações sobre objetos em particular. A análise subseqüente pode ser estatística ou baseada em pesquisa. Mesmo quando nenhuma violação é suspeita, com o passar do tempo, as estatísticas podem ser comparadas para ajudar a descobrir tendências ou eventos incomuns.

O projeto de sistemas seguros é um exercício de ponderação dos custos em relação às ameaças. A série de técnicas que podem ser implantadas para proteger processos e tornar a comunicação entre processos segura é forte o suficiente para suportar quase qualquer ataque, mas seu uso acarreta custos e inconveniências:

- há um custo (em esforço computacional e na utilização da rede) por seu uso. Os custos devem ser ponderados em relação às ameaças;
- medidas de segurança especificadas de forma inadequada podem excluir usuários legítimos da execução das ações necessárias.

Tais compromissos são difíceis de identificar sem comprometer a segurança, e pode parecer que estão em conflito com o conselho do primeiro parágrafo desta subseção, mas o poder das técnicas de

segurança pode ser quantificado e selecionado com base no custo estimado de um ataque a elas. Um exemplo são as técnicas de custo relativamente baixo empregadas no protocolo Millicent para pequenas transações comerciais, descrito no endereço www.cdk4.net/security.

Como ilustração das dificuldades e acidentes que podem advir no projeto de sistemas seguros, examinaremos, na Seção 7.6.4, aquelas que surgiram no projeto de segurança incorporado no padrão de interligação em rede IEEE 802.11 WiFi.

7.2 Visão geral das técnicas de segurança

O objetivo desta seção é apresentar ao leitor algumas das técnicas e mecanismos mais importantes para tornar seguros sistemas e aplicativos distribuídos. Aqui, os descreveremos informalmente, reservando as descrições mais rigorosas para as Seções 7.3 e 7.4. Vamos usar os nomes dos principais (protagonistas) apresentados na Figura 7.1 e as notações para itens cifrados e assinados mostradas na Figura 7.2.

7.2.1 Criptografia

Criptografia é o processo de codificar uma mensagem de maneira a ocultar seu conteúdo. A criptografia moderna inclui vários algoritmos de segurança para cifrar e decifrar mensagens baseados no uso de segredos chamados *chaves*. Uma chave de criptografia é um parâmetro usado em um algoritmo de criptografia de tal maneira que a criptografia não possa ser revertida sem o conhecimento da chave.

K_A	Chave secreta de Alice
K_B	Chave secreta de Bob
K_{AB}	Chave secreta compartilhada entre Alice e Bob
K_{Apriv}	Chave privada de Alice (conhecida apenas por Alice)
K_{Apub}	Chave pública de Alice (publicada por Alice para todos lerem)
$\{M\}_K$	Mensagem M cifrada com a chave K
$[M]_K$	Mensagem M assinada com a chave K

Figura 7.2 Notações de criptografia.

Existem duas classes principais de algoritmo de criptografia em uso geral. A primeira usa *chaves secretas compartilhadas* – o remetente e o destinatário devem compartilhar o conhecimento da chave e ela não deve ser revelada a mais ninguém. A segunda classe de algoritmos de criptografia usa *pares de chave pública/privada* – o remetente de uma mensagem usa uma *chave pública* – que já foi publicada pelo destinatário – para cifrar a mensagem. O destinatário usa uma *chave privada* correspondente, para decifrar a mensagem. Embora muitos principais possam inspecionar a chave pública, apenas o destinatário pode decifrar a mensagem, pois somente ele possui a chave privada.

As duas classes de algoritmo de criptografia são extremamente úteis e são amplamente usadas na construção de sistemas distribuídos seguros. Normalmente, os algoritmos de criptografia de chave pública exigem de 100 a 1000 vezes mais poder de processamento do que os algoritmos de chave secreta, mas existem situações onde sua conveniência supera essa desvantagem.

7.2.2 Usos da criptografia

A criptografia desempenha três papéis importantes na implementação de sistemas seguros. Os apresentaremos aqui em esboço, por meio de alguns cenários simples. Em seções posteriores deste capítulo, descreveremos esses e outros protocolos com mais detalhes, tratando de alguns problemas não resolvidos que são apenas destacados aqui.

Em todos os nossos cenários a seguir, podemos supor que Alice, Bob e todos os outros participantes já concordaram com os algoritmos de criptografia que desejam usar e têm implementações deles. Também supomos que as chaves secretas ou privadas que eles detêm podem ser armazenadas com segurança para impedir que invasores as obtenham.

Segredo e integridade ◊ A criptografia é usada para manter o segredo e a integridade da informação, quando ela é exposta a ataques em potencial; por exemplo, durante a transmissão em redes vulneráveis à intromissão e à falsificação da mensagem. Esse uso da criptografia corresponde à sua função tradicional em atividades militares e de inteligência. Ele explora o fato de que uma mensagem cifrada com uma chave de criptografia em particular só pode ser decifrada por um destinatário que conheça a chave para decifrar correspondente. Assim, ele mantém o segredo da mensagem cifrada, desde que a chave para decifrar não seja *comprometida* (exposta a quem não é participante da comunicação) e que o algoritmo de criptografia seja poderoso o suficiente para anular todas as tentativas possíveis de violá-lo. A criptografia também mantém a integridade da informação cifrada, já é possível incluir e verificar algum tipo de informação redundante, como uma soma de verificação.

Cenário 1. Comunicação secreta com uma chave secreta compartilhada: Alice deseja enviar algumas informações secretamente para Bob. Alice e Bob compartilham uma chave secreta K_{AB}.

1. Alice usa K_{AB} e uma função de criptografia consensual $C(K_{AB}, M)$ para cifrar e enviar qualquer número de mensagens $\{M_i\}K_{AB}$ para Bob. (Alice pode usar K_{AB}, desde que seja seguro supor que K_{AB} não foi comprometida.)
2. Bob lê as mensagens cifradas e as decifra usando a função correspondente $D(K_{AB}, M)$.

Agora, Bob pode ler a mensagem original M. Se a mensagem fizer sentido quando for decifrada por Bob, ou melhor, se ela incluir algum valor acordado entre Alice e Bob, como uma soma de verificação da mensagem, então Bob saberá que a mensagem é de Alice e que não foi falsificada. Mas ainda existem alguns problemas:

Problema 1: como Alice pode enviar uma chave compartilhada K_{AB} para Bob com segurança?

Problema 2: como Bob sabe que qualquer $\{M_i\}$ não é uma cópia de uma mensagem cifrada anteriormente por Alice que foi capturada por Mallory e reproduzida posteriormente? Mallory não precisa ter a chave K_{AB} para realizar esse ataque – ele pode simplesmente copiar o padrão de bits que representa a mensagem e enviá-la para Bob posteriormente. Por exemplo, se a mensagem fosse um pedido para pagar alguém em dinheiro, Mallory poderia enganar Bob, fazendo-o pagar duas vezes.

Posteriormente neste capítulo, vamos mostrar como esses problemas podem ser resolvidos.

Autenticação ◊ A criptografia é usada no suporte de mecanismos de autenticação da comunicação entre pares de principais. Um principal que cifra uma mensagem com êxito, usando uma chave em particular, pode supor que a mensagem é autêntica se ela contiver uma soma de verificação correta, ou algum outro valor esperado, se for usado o modo de criptografia com encadeamento de blocos (Seção 7.3). Ele pode concluir que o remetente da mensagem possuía a chave de cifrar correspondente e, daí, deduzir a identidade do remetente, caso a chave seja conhecida apenas pelas duas partes. Assim, se as chaves forem mantidas em segredo, uma ação de decifrar bem-sucedida autenticará a mensagem como sendo proveniente de um remetente em particular.

Cenário 2. Comunicação autenticada com um servidor: Alice deseja acessar arquivos mantidos por Bob em um servidor de arquivos na rede local da organização onde ela trabalha. Sara é um servidor de autenticação gerenciado com segurança. Sara distribui senhas para os usuários e contém as chaves secretas correntes de todos os principais do sistema que atende (geradas pela aplicação de alguma transformação na senha do usuário). Por exemplo, ele conhece a chave K_A de Alice e K_B de Bob. Em nosso cenário, nos referimos a um *tíquete*. Um tíquete é um item cifrado emitido por um servidor de autenticação que contém a identidade do principal para quem ele é emitido e uma chave compartilhada, gerada para a sessão de comunicação corrente.

1. Alice envia uma mensagem (não cifrada) para Sara, informando sua identidade e solicitando um tíquete para acessar Bob.

2. Sara envia uma resposta para Alice, cifrada com K_A, consistindo em um tíquete (a ser enviado para Bob com cada pedido de acesso a arquivo) cifrado com K_B e uma nova chave secreta K_{AB} para uso na comunicação com Bob. Então, a resposta recebida por Alice é como a seguinte: $\{\{Ticket\}K_B, K_{AB}\}K_A$.
3. Alice decifra a resposta usando K_A (que ela gera a partir de sua senha, usando a mesma transformação; a senha não é transmitida pela rede e, uma vez que tenha sido usada, ela é excluída do armazenamento local para evitar seu comprometimento). Se Alice tiver a chave K_A derivada da senha correta, ela obterá um tíquete válido para usar o serviço de Bob e uma nova chave secreta para uso na comunicação com Bob. Alice não pode decifrar nem falsificar o tíquete, pois ele está cifrado por K_B. Se o destinatário não for Alice, então ele não saberá a senha de Alice; portanto, não poderá decifrar a mensagem.
4. Alice envia o tíquete para Bob, junto com sua identidade e um pedido R para acessar um arquivo: $\{Ticket\}K_B, Alice, R$.
5. O tíquete, originalmente criado por Sara, é na verdade $\{\{K_{AB}, Alice\}K_B$. Bob decifra o tíquete usando sua chave K_B. Portanto, Bob obtém a identidade autêntica de Alice (com base no conhecimento da senha de Alice, compartilhado entre Alice e Sara) e uma nova chave secreta compartilhada K_{AB} para uso na interação com Alice. (Ela é chamada de *chave de sessão*, pois pode ser usada com segurança por Alice e Bob para uma seqüência de interações.)

Esse cenário é uma versão simplificada do protocolo de autenticação originalmente desenvolvido por Roger Needham e Michael Schroeder [1978] e usado subseqüentemente no sistema Kerberos, desenvolvido e usado no MIT [Steiner *et al.* 1988], que será descrito na Seção 7.6.2. Em nossa descrição simplificada desse protocolo não há nenhuma proteção contra a reprodução de mensagens de autenticação antigas. Essa e algumas outras deficiências serão tratadas em nossa descrição do protocolo Needham–Schroeder completo, na Seção 7.6.1.

O protocolo de autenticação que descrevemos depende do conhecimento anterior, por parte do servidor de autenticação Sara, das chaves K_A e K_B de Alice e de Bob. Isso é possível em uma única organização, onde Sara é executado em um computador fisicamente seguro e gerenciado por um principal confiável que gera os valores iniciais das chaves e os transmite para os usuários por meio de um canal seguro separado. Mas ele não é apropriado para comércio eletrônico ou outras aplicações remotas, onde o uso de um canal separado é extremamente inconveniente e o requisito de uma terceira pessoa confiável, irreal. A criptografia de chave pública nos salva desse dilema.

A utilidade dos desafios (*challenges*): um aspecto importante da ruptura de Needham e Schroeder, de 1978, foi a percepção de que a senha de um usuário não precisa ser enviada para um serviço de autenticação (e, assim, exposta na rede) sempre que for usada. Em vez disso, eles apresentaram o conceito de *desafio*. Isso pode ser visto no passo 2 de nosso cenário anterior, onde o servidor Sara emite um tíquete para Alice, *cifrado na chave secreta K_A de Alice*. Isso constitui um desafio, pois Alice não pode usar o tíquete, a não ser que consiga decifrá-lo, e ela só pode fazer isso se puder determinar K_A, que é derivada da senha de Alice. Um impostor dizendo-se ser Alice seria anulado nesse ponto.

Cenário 3. Comunicação autenticada com chaves públicas: supondo que Bob tenha gerado um par de chaves pública/privada, o seguinte diálogo permite que Bob e Alice estabeleçam uma chave secreta compartilhada K_{AB}:

1. Alice acessa um serviço de distribuição de chaves para obter um *certificado de chave pública*, fornecendo a chave pública de Bob. Ele é chamado de certificado porque é assinado por uma autoridade confiável – uma pessoa, ou organização, amplamente conhecida como sendo confiável. Após verificar a assinatura, ela lê a chave pública K_{Bpub} de Bob no certificado. (Discutiremos a construção e o uso de certificados de chave pública na Seção 7.2.3.)
2. Alice cria uma nova chave compartilhada K_{AB} e a cifra usando K_{Bpub} com um algoritmo de chave pública. Ela envia o resultado para Bob, junto com um nome (*keyname*) que identifica exclusivamente um par de chaves pública/privada (pois Bob pode ter vários deles). Então, Alice envia *keyname*, $\{K_{AB}\}K_{Bpub}$ para Bob.

3. Bob seleciona a chave privada K_{Bpriv} correspondente em seu conjunto de chaves privadas e a utiliza para decifrar a mensagem e obter K_{AB}. Note que a mensagem de Alice para Bob poderia ter sido corrompida, ou falsificada, quando estava em trânsito. A conseqüência seria simplesmente que Bob e Alice não compartilhariam a mesma chave K_{AB}. Se isso for problema, ele pode ser resolvido pela adição de um valor, ou de um string, previamente combinado na mensagem, como os nomes ou os endereços de e-mail de Bob e de Alice, os quais Bob pode verificar após decifrá-la.

O cenário anterior ilustra o uso da criptografia de chave pública para distribuir uma chave secreta compartilhada. Essa técnica é conhecida como *protocolo de criptografia misto* e é muito utilizada, pois explora recursos úteis tanto dos algoritmos de criptografia de chave pública como os de chave secreta.

Problema: essa troca de chave é vulnerável a ataques de homem no meio (*man-in-the-middle*). Mallory pode interceptar o pedido inicial de Alice para o serviço de distribuição de chaves para obter o certificado de chave pública de Bob, e enviar uma resposta contendo sua própria chave pública. Então, ele poderá interceptar todas as mensagens subseqüentes. Em nossa descrição anterior, nós nos defendemos desse ataque exigindo que o certificado de Bob fosse assinado por uma autoridade conhecida. Para se proteger desse ataque, Alice deve garantir que o certificado de chave pública de Bob seja assinado com uma chave pública (conforme descrito a seguir) que ela tenha recebido de uma maneira totalmente segura.

Assinaturas digitais ◊ A criptografia é usada para implementar um mecanismo conhecido como *assinatura digital*. Isso simula a função das assinaturas convencionais, verificando com um outro elemento se uma mensagem, ou um documento, é uma cópia inalterada do que foi produzido pelo signatário.

As técnicas de assinatura digital são baseadas em um vínculo irreversível na mensagem ou documento de um segredo conhecido apenas pelo signatário. Isso pode ser obtido cifrando-se a mensagem – ou melhor, uma forma compactada da mensagem chamada *resumo* (*digest*), usando uma chave conhecida apenas pelo signatário. Um resumo é um valor de comprimento fixo, calculado pela aplicação de uma *função de resumo segura* (*secure digest function*). A função de resumo segura é semelhante a uma função de soma de verificação, mas é muito improvável que ela produza um valor de resumo semelhante para duas mensagens diferentes. O resumo resultante, cifrado, atua como uma assinatura que acompanha a mensagem. Geralmente, é usada criptografia de chave pública para isso: o criador gera uma assinatura com sua chave privada; a assinatura pode ser decifrada por qualquer destinatário usando a chave pública correspondente. Há um requisito adicional: o verificador deve ter certeza de que a chave pública é realmente do principal que diz ser o signatário – isso é tratado com o uso de certificados de chave pública, descritos na Seção 7.2.3.

Cenário 4. Assinaturas digitais com uma função de resumo segura: Alice quer assinar um documento M para que todo destinatário subseqüente possa verificar se ela é sua criadora. Assim, quando Bob acessar posteriormente o documento assinado, após recebê-lo por meio de qualquer rota e de qualquer fonte (por exemplo, ele poderia ser enviado em uma mensagem ou ser recuperado de um banco de dados), poderá verificar se Alice é a criadora.

1. Alice calcula um resumo de comprimento fixo do documento $Digest(M)$.
2. Alice cifra o resumo com sua chave privada, a anexa a M e torna o resultado M, $\{Digest(M)\}K_{Apriv}$ disponível para os usuários pretendidos.
3. Bob obtém o documento assinado, extrai M e calcula $Digest(M)$.
4. Bob decifra $\{Digest(M)\}K_{Apriv}$ usando a chave pública K_{Apub} de Alice e compara o resultado com seu $Digest(M)$ calculado. Se eles corresponderem, a assinatura é válida.

7.2.3 Certificados

Um certificado digital é um documento contendo uma declaração (normalmente curta) assinada por um principal. Ilustraremos o conceito com um cenário.

1. Tipo de certificado:	Número de conta
2. Nome:	Alice
3. Conta:	6262626
4. Autoridade certificadora:	Banco Bob
5. Assinatura:	$\{Digest(field\ 2 + field\ 3)\}K_{Bpriv}$

Figura 7.3 Certificado da conta bancária de Alice.

Cenário 5. O uso de certificados: Bob é um banco. Quando seus clientes estabelecem contato, eles precisam ter certeza de que estão falando com Bob (o banco), mesmo que nunca tenham entrado em contato com ele antes. Bob precisa autenticar seus clientes, antes de dar a eles acesso às suas contas.

Por exemplo, Alice poderia achar útil obter um certificado de seu banco, informando o número de sua conta (Figura 7.3). Alice poderia usar esse certificado para fazer compras como forma de garantir que tem uma conta no Banco Bob. O certificado é assinado na chave privada K_{Bpriv} de Bob. Uma vendedora, Carol, pode aceitar tal certificado para cobrar produtos na conta de Alice, desde que ela possa validar a assinatura no campo 5. Para isso, Carol precisa ter a chave pública de Bob e certificar-se de que ela é autêntica para evitar a possibilidade de que Alice possa assinar um certificado falso, associando seu nome à conta de outra pessoa. Para realizar esse ataque, Alice simplesmente geraria um novo par de chaves $K_{B'pub}$, $K_{B'priv}$ e as usaria para gerar um certificado falsificado, dando a entender que seria proveniente do Banco Bob.

O que Carol precisa é de um certificado informando a chave pública de Bob, assinado por uma autoridade conhecida e confiável. Vamos supor que Fred represente a Federação dos Bancos, cuja função, entre outras, é certificar as chaves públicas dos bancos. Então, Fred emitiria um certificado de chave pública para Bob (Figura 7.4).

É claro que esse certificado depende da autenticidade da chave pública K_{Fpub} de Fred; portanto, temos um problema de autenticidade recursivo – Carol só pode contar com esse certificado se puder ter certeza de que conhece a chave pública K_{Fpub} autêntica de Fred. Podemos quebrar essa recursividade garantindo que Carol obtenha K_{Fpub} por algum meio no qual ela possa ter confiança – ela poderia recebê-la por intermédio de um representante de Fred, ou poderia receber uma cópia assinada dele, ou ainda, de alguém que conhece e em quem confia, dizendo que a recebeu diretamente de Fred. Nosso exemplo ilustra um encadeamento de certificados – neste caso, com dois vínculos.

Já mencionamos um dos problemas que surgem com os certificados – a dificuldade de escolher uma autoridade confiável a partir da qual um encadeamento de autenticações possa começar. Raramente a confiança é absoluta; portanto, a escolha de uma autoridade deve depender do objetivo do certificado. Outros problemas surgem com o risco das chaves privadas serem comprometidas (descobertas) e com o comprimento permitido para um encadeamento de certificados – quanto maior o encadeamento, maior o risco de um vínculo fraco.

Desde que se tenha o cuidado de tratar dessas questões, os encadeamentos de certificados são uma base importante para o comércio eletrônico e outros tipos de transação real. Eles ajudam a tratar do problema da escala: existem seis bilhões de pessoas no mundo; então, como podemos construir um ambiente eletrônico no qual possamos estabelecer as credenciais de todas elas?

1. Tipo de certificado:	Chave pública
2. Nome:	Banco Bob
3. Chave pública:	K_{Bpub}
4. Autoridade certificadora:	Fred – Federação dos Bancos
5. Assinatura:	$\{Digest(field\ 2 + field\ 3)\}K_{Fpriv}$

Figura 7.4 Certificado de chave pública do Banco Bob.

Os certificados podem ser usados para estabelecer a autenticidade de muitos tipos de declaração. Por exemplo, os membros de um grupo ou associação talvez quisessem manter uma lista de e-mails aberta apenas para os membros do grupo. Uma boa maneira de fazer isso seria o gerente do quadro de associados (Bob) emitir um certificado de associado $(S,Bob,\{Digest(S)\}K_{Bpriv})$ para cada membro, onde S é uma declaração da forma *Alice é membro da Sociedade dos Amigos* e K_{Bpriv} é a chave privada de Bob. Um membro solicitando sua inscrição na lista de e-mails da Sociedade dos Amigos teria que fornecer uma cópia desse certificado para o sistema de gerenciamento da lista, o qual verificaria o certificado antes de efetivar a inscrição.

Para tornar os certificados úteis, duas coisas são necessárias:

- um formato padrão e uma representação para eles, de modo que os emitentes e os usuários do certificado possam ter êxito em construí-los e interpretá-los;
- acordo sobre a maneira pela qual os encadeamentos de certificados são construídos e, em particular, sobre a noção de autoridade confiável.

Vamos voltar a esses requisitos na Seção 7.4.4.

Às vezes, há necessidade de revogar um certificado – por exemplo, Alice poderia deixar de ser membro da Sociedade dos Amigos, mas ela e outros membros provavelmente continuariam a manter cópias armazenadas de seus certificados de associados. Seria dispendioso, ou mesmo impossível, rastrear e excluir todos esses certificados, e não é fácil invalidar um certificado – seria necessário notificar todos os destinatários possíveis sobre um certificado revogado. A solução usual para esse problema é incluir uma data de expiração no certificado. Quem receber um certificado expirado deve rejeitá-lo e o dono do certificado deve solicitar sua renovação. Se for exigida uma revogação mais rápida, então deve-se recorrer a um dos mecanismos mais complicados mencionados.

7.2.4 Controle de acesso

Esboçaremos aqui os conceitos nos quais é baseado o controle de acesso aos recursos em sistemas distribuídos e as técnicas pelas quais ele é implementado. A base conceitual para proteção e controle de acesso foi estabelecida muito claramente em um artigo clássico de Lampson [1971] e detalhes de implementações não-distribuídas podem ser encontrados em muitos livros sobre sistemas operacionais [Stallings 1998b].

Historicamente, a proteção de recursos em sistemas distribuídos é específica ao serviço. Os servidores recebem mensagens de pedido da forma *<op, principal, recurso>*, onde *op* é a operação solicitada, *principal* é uma identidade, ou um conjunto de credenciais, do principal que está fazendo o pedido e *recurso* identifica o recurso no qual a operação deve ser aplicada. O servidor deve primeiro autenticar a mensagem de pedido e as credenciais do principal e depois aplicar o controle de acesso, recusando qualquer pedido para o qual o principal solicitante não tenha os direitos de acesso necessários para efetuar a operação solicitada no recurso especificado.

Nos sistemas distribuídos orientados a objetos pode haver muitos tipos de objeto nos quais o controle de acesso deve ser aplicado; e as decisões freqüentemente são específicas do aplicativo. Por exemplo, Alice só pode fazer um saque por dia em sua conta bancária, enquanto Bob pode fazer três. As decisões de controle de acesso normalmente são deixadas para o código em nível de aplicativo, mas é fornecido suporte genérico para grande parte do mecanismo que suporta as decisões. Isso inclui a autenticação de principais, a assinatura e autenticação dos pedidos e o gerenciamento de credenciais e informações sobre os direitos de acesso.

Domínios de proteção ◊ Um domínio de proteção é um ambiente de execução compartilhado por um conjunto de processos: ele contém um conjunto de pares *<recurso, direitos>*, listando os recursos que podem ser acessados por todos os processos em execução dentro do domínio e especificando as operações permitidas em cada recurso. Normalmente, um domínio de proteção é associado a um principal – quando um usuário se conecta, sua identidade é autenticada e um domínio de proteção é criado para os processos que ele executará. Conceitualmente, o domínio contém todos os direitos de acesso que o principal possui, incluindo todos os direitos que ele adquire através da participação como membro de vários grupos. Por exemplo, no UNIX, o domínio de proteção de um processo é determi-

nado pelos identificadores de usuário e grupo anexados ao processo no momento do *login*. Os direitos são especificados em termos das operações permitidas. Por exemplo, um arquivo poderia ser lido e gravado por um processo e somente lido por outro.

O domínio de proteção é apenas uma abstração. Duas implementações alternativas são comumente usadas em sistemas distribuídos. São elas as *listas de capacidades* (*capabilities*) e as *listas de controle de acesso*.

Listas de capacidades: um conjunto de capacidades é mantido por processo, de acordo com o domínio em que ele estiver localizado. A capacidade é um valor binário que atua como uma chave de acesso, permitindo que seu possuidor execute certas operações em um recurso especificado. Para uso em sistemas distribuídos, onde as capacidades devem ser impossíveis de falsificar, elas assumem uma forma como a seguinte:

Identificador de recurso	Um identificador exclusivo para o recurso pretendido
Operações	Uma lista das operações permitidas no recurso
Código de autenticação	Uma assinatura digital tornando a capacidade impossível de ser falsificada

Os serviços só fornecem capacidades para os clientes quando estes tiverem sidos autenticados como sendo pertencentes ao domínio de proteção alegado. A lista de operações na capacidade é um subconjunto das operações definidas para o recurso pretendido e freqüentemente é codificada como um mapa de bits. Diferentes capacidades são usadas para diferentes combinações de direitos de acesso para o mesmo recurso.

Quando as capacidades são usadas, os pedidos dos clientes assumem a forma *<op, userid, capacidade>*. Isto é, eles incluem uma capacidade para o recurso a ser acessado, em vez de um simples identificador, dando ao servidor uma prova imediata de que o cliente está autorizado a acessar o recurso identificado pela capacidade, com as operações especificadas por ela. Uma verificação de controle de acesso em um pedido que esteja acompanhado por uma capacidade envolve a sua validação e a verificação de que a operação solicitada está no conjunto permitido pela capacidade. Essa característica é a principal vantagem das capacidades – elas constituem uma chave de acesso auto-suficiente, exatamente como uma chave física de uma fechadura de porta é uma chave de acesso para o prédio que a fechadura protege.

As capacidades compartilham dois inconvenientes das chaves de uma fechadura física:

Roubo de chave: qualquer um que possua a chave de um prédio pode usá-la para ganhar acesso, seja proprietário autorizado da chave ou não – a pessoa pode ter roubado a chave, ou a obtido de alguma maneira fraudulenta.

O problema da revogação: a autorização para manter uma chave muda com o tempo. Por exemplo, o portador pode deixar de ser funcionário do dono do prédio, mas poderia reter a chave, ou uma cópia dela, e usá-la de uma maneira não autorizada.

As únicas soluções disponíveis para esses problemas das chaves físicas são (1) prender o portador da chave ilícita – o que nem sempre é possível a tempo de evitar que ele cause danos – ou (2) mudar a fechadura e redistribuir chaves para todos os proprietários – uma operação deselegante e dispendiosa.

Os problemas análogos das capacidades são claros:

- As capacidades podem, por falta de cuidado, ou como resultado de um ataque de intromissão, cair nas mãos de principais que não sejam aqueles para os quais elas foram emitidas. Se isso acontecer, os servidores não terão poderes para impedir que sejam usados ilicitamente.
- É difícil cancelar capacidades. O status do portador pode mudar e seus direitos de acesso devem mudar da mesma maneira, mas ele ainda pode usar a capacidade.

Soluções para esses dois problemas, baseadas na inclusão de informações identificando o portador e em tempos limites, em conjunto com listas de capacidades revogadas têm sido propostas e desenvolvidas [Gong 1989, Hayton *et al.* 1998]. Embora aumentem a complexidade de um conceito

simples, as capacidades continuam sendo uma técnica importante; por exemplo, elas podem ser usadas em conjunto com as listas de controle de acesso para otimizar o controle do acesso repetido a um mesmo recurso, e elas fornecem um mecanismo mais organizado para a implementação de delegação (veja a Seção 7.2.5).

É interessante notar a semelhança entre capacidades e certificados. Considere o certificado de posse de conta bancária de Alice, apresentado na Seção 7.2.3. Ele difere das capacidades descritas aqui somente porque não há uma lista de operações permitidas e porque o emitente é identificado. Os certificados e as capacidades podem ser conceitos indistintos em algumas circunstâncias. O certificado de Alice poderia ser considerado uma chave de acesso à conta bancária de Alice para executar todas as operações permitidas aos correntistas, desde que o solicitante possa comprovar que é mesmo Alice.

Listas de controle de acesso: cada recurso possui uma lista associada com entradas na forma *<domínio, operações>*, fornecendo as operações permitidas para cada domínio que tenha acesso ao recurso. Um domínio pode ser especificado pelo identificador de um principal, ou ser uma expressão que pode ser usada para determinar a participação de um principal como membro do domínio. Por exemplo, *o proprietário deste arquivo* é uma expressão que pode ser avaliada pela comparação da identidade do principal solicitante com a identidade do proprietário armazenada com um arquivo.

Esse é o esquema adotado na maioria dos sistemas de arquivos, incluindo UNIX e Windows NT, onde um conjunto de bits de permissão de acesso é associado a cada arquivo, e os domínios para os quais as permissões são concedidas são definidos pela referência às informações de membro armazenadas em cada arquivo.

Os pedidos para servidores assumem a forma *<op, principal, recurso>*. Para cada pedido, o servidor autentica o principal e verifica se a operação solicitada está incluída na entrada associada a este principal na lista de controle de acesso do recurso relevante.

Implementação ◊ As assinaturas digitais, credenciais e certificados de chave pública fornecem a base de criptografia para o controle de acesso seguro. Canais seguros oferecem vantagens de desempenho, permitindo que vários pedidos sejam manipulados sem a necessidade de verificação repetida dos principais e das credenciais [Wobber *et al.* 1994].

Tanto CORBA como Java oferecem APIs de segurança. O suporte para controle de acesso é um de seus principais objetivos. A linguagem Java fornece suporte para objetos distribuídos para gerenciar seu próprio controle de acesso com as classes Principal, Signer e ACL, e métodos padrão para autenticação e suporte para certificados, validação de assinaturas e verificações de controle de acesso. Também são suportadas criptografia de chave secreta e de chave pública. Farley [1998] apresenta uma boa introdução para esses recursos da linguagem Java. A proteção dos programas Java que incluem código móvel é baseada no conceito de domínio de proteção – código local e código carregado por *download* recebem diferentes domínios de proteção para sua execução. Pode haver um domínio de proteção para cada fonte de *download*, com direitos de acesso para diferentes conjuntos de recursos locais, dependendo do nível de confiança depositado no código carregado.

O CORBA oferece uma especificação para um serviço de segurança (*Security Service*) [Blakley 1999, OMG 2002b] com um modelo para ORBs, para fornecer comunicação segura, autenticação, controle de acesso com credenciais, ACLs e auditoria; esses itens serão melhor descritos na Seção 17.3.4.

7.2.5 Credenciais

Credenciais são um conjunto de evidências fornecidas por um principal ao solicitar acesso a um recurso. No caso mais simples, um certificado de uma autoridade relevante informando a identidade do principal é suficiente, e isso seria usado para verificar as permissões do principal em uma lista de controle de acesso (veja a Seção 7.2.4). Freqüentemente, isso é tudo que é exigido, ou fornecido, mas o conceito pode ser generalizado para tratar com requisitos muito mais sutis.

Não é conveniente exigir que os usuários interajam com o sistema e autentiquem a si mesmos sempre que sua autorização for exigida para executar uma operação em um recurso protegido. Em vez disso, é introduzida a noção de que uma credencial *representa* um principal. Assim, o certificado de chave pública de um usuário representa esse usuário – qualquer processo que receba um pedido autenticado com a chave privada do usuário pode supor que o pedido foi feito por esse usuário.

A idéia de *representa* pode ser levada bem mais adiante. Por exemplo, em uma tarefa cooperativa poderia ser exigido que certas ações sigilosas só fossem executadas com a autorização de dois membros da equipe; nesse caso, o principal que está solicitando a ação enviaria sua própria credencial de identificação e uma credencial de endosso de outro membro da equipe, junto com uma indicação de que elas devem ser consideradas em conjunto na verificação das credenciais.

Analogamente, para votar em uma eleição, um pedido de voto seria acompanhado de um certificado de eleitor, assim como de um certificado de identificação. Um certificado de delegação permite que um principal atue em nome de outro e assim por diante. Em geral, uma verificação de controle de acesso envolve a avaliação de uma fórmula lógica combinando os certificados fornecidos. Lampson *et al.* [1992] desenvolveram uma lógica de autenticação abrangente para uso na avaliação da autoridade de *representa* realizada por um conjunto de credenciais. Wobber *et al.* [1994] descrevem um sistema que suporta exatamente essa estratégia geral. Mais trabalhos sobre formas úteis de credenciais para uso em tarefas cooperativas reais podem ser encontrados em [Rowley 1998].

As credenciais baseadas na função (*role-based credentials*) parecem particularmente úteis no projeto de esquemas práticos de controle de acesso [Sandhu *et al.* 1996]. Conjuntos de credenciais baseadas na função são definidos para organizações, ou para tarefas cooperativas, e direitos de acesso em nível de aplicativo são construídos com referência a eles. Então, funções podem ser atribuídas a principais específicos, por meio da geração de um certificado de função associando um principal a uma função nomeada em uma tarefa ou organização específica [Coulouris *et al.* 1998].

Delegação ◊ Uma forma particularmente útil de credencial é aquela que autoriza um principal, ou a um processo atuando para um principal, executar uma ação com a autoridade de outro principal. Pode surgir uma necessidade de delegação em qualquer situação onde um serviço precise acessar um recurso protegido para completar uma ação em nome de seu cliente. Considere o exemplo de um servidor de impressão que aceita pedidos para imprimir arquivos. Seria um desperdício de recursos copiar o arquivo; portanto, o nome do arquivo é passado para o servidor de impressão, e o arquivo é acessado pelo servidor em nome do usuário que está fazendo o pedido. Se o arquivo for protegido contra leitura, isso não funcionará, a não ser que o servidor de impressão possa adquirir direitos temporários para ler o arquivo. A delegação é um mecanismo projetado para resolver problemas como esse.

A delegação pode ser obtida usando-se um certificado de delegação ou uma capacidade. O certificado é assinado pelo principal solicitante e ele autoriza outro principal (o servidor de impressão, em nosso exemplo) a acessar um recurso nomeado (o arquivo a ser impresso). Nos sistemas que as suportam, as capacidades podem obter o mesmo resultado sem a necessidade de identificar os principais – a capacidade de acessar um recurso pode ser passada em um pedido para um servidor. A capacidade é um conjunto de direitos codificados, impossível de falsificar, para acessar o recurso.

Quando direitos são delegados, é comum restringi-los a um subconjunto dos direitos mantidos pelo principal emitente para que o principal delegado não possa fazer mal uso deles. Em nosso exemplo, o certificado poderia ter um tempo limitado para reduzir o risco de que o código do servidor de impressão fosse subseqüentemente comprometido e o arquivo exposto a terceiros. O serviço de segurança (*Security Service*) CORBA inclui um mecanismo para a delegação de direitos baseado em certificados, com suporte para a restrição dos direitos transmitidos.

7.2.6 *Firewalls*

Os *firewalls* foram apresentados e descritos na Seção 3.4.8. Eles protegem intranets, realizando ações de filtragem em comunicações recebidas e enviadas. Aqui, discutiremos suas vantagens e inconvenientes como mecanismos de segurança.

Em um mundo ideal, a comunicação sempre se daria entre processos mutuamente confiáveis e seriam sempre usados canais seguros. Existem muitos motivos pelos quais esse ideal não é atingido, alguns corrigíveis, mas outros inerentes à natureza aberta dos sistemas distribuídos, ou resultantes de erros que estão presentes na maior parte do software. A facilidade com que as mensagens de pedido podem ser enviadas para qualquer servidor, em qualquer parte, e o fato de que muitos servidores não são projetados para suportar ataques maldosos de *hackers*, ou erros acidentais, torna fácil o vazamento de informações destinadas a serem confidenciais. Elementos indesejáveis também podem penetrar na

rede de uma organização, permitindo que programas *worm* e vírus entrem em seus computadores. Veja [web.mit.edu II] para uma crítica mais detalhada dos *firewalls*.

Os *firewalls* produzem um ambiente de comunicação local no qual toda a comunicação externa é interceptada. As mensagens são encaminhadas para o destinatário local pretendido apenas para comunicações explicitamente autorizadas.

O acesso às redes internas pode ser controlado por *firewalls*, mas o acesso a serviços públicos na Internet é irrestrito, pois seu objetivo é oferecer serviços para uma ampla gama de usuários. O uso de *firewalls* não oferece nenhuma proteção contra ataques internos em uma organização e é deficiente em seu controle de acesso externo. Há necessidade de mecanismos de segurança mais refinados, permitindo que usuários individuais compartilhem informações com outros usuários selecionados, sem comprometer a privacidade e a integridade. Abadi *et al.* [1998] descrevem uma estratégia para o uso de acesso a dados privados da web para usuários externos, baseada em um mecanismo de *túnel* web que pode ser integrado com um *firewall*. Para usuários confiáveis e autenticados, ela oferece acesso a servidores web internos por intermédio de um *proxy* seguro baseado no protocolo HTTPS (HTTP sobre TLS).

Os *firewalls* não são particularmente eficazes contra ataques de negação de serviço, como aquele baseado no *spoofing* de IP que descrevemos na Seção 3.4.2. O problema é que a avalanche de mensagens gerada por tais ataques sobrecarrega qualquer ponto de defesa único, como um *firewall*. Qualquer solução para avalanches de mensagens recebidas deve ser aplicada em um nível bem mais acima do destino. As soluções baseadas no uso de mecanismos de qualidade de serviço para restringir o fluxo de mensagens da rede a um nível que o destino possa tratá-las parecem os mais promissores.

7.3 Algoritmos de criptografia

Uma mensagem é cifrada pelo remetente aplicando alguma regra para transformar a mensagem de *texto puro* (qualquer seqüência de bits) em um *texto cifrado* (uma seqüência de bits diferente). O destinatário deve conhecer a regra inversa para transformar o texto cifrado no texto puro original. Outros principais não podem decifrar a mensagem, a menos que conheçam a regra inversa. A transformação da criptografia é definida com duas partes, uma *função C* e uma *chave K*. A mensagem *M* cifrada resultante é escrita como $\{M\}_K$.

$$C(K, M) = \{M\}_K$$

A função de criptografia *C* define um algoritmo que transforma itens de dados de texto puro em dados cifrados, combinando-os com a chave e transpondo-os de uma maneira fortemente dependente do valor da chave. Podemos considerar um algoritmo de criptografia como a especificação de uma grande família de funções, das quais um membro em particular é selecionado por determinada chave. A decifração é feita usando-se a função inversa *D*, que também recebe uma chave como parâmetro. Para a criptografia de chave secreta, a chave usada para decifrar é a mesma para cifrar:

$$D(K, E(K, M)) = \{M\}$$

Devido ao uso simétrico das chaves, a criptografia de chave secreta é freqüentemente referida como *criptografia simétrica*, enquanto a criptografia de chave pública é referida como *assimétrica*, pois as chaves usadas para cifrar e decifrar são diferentes, conforme veremos a seguir. Na próxima seção, vamos descrever várias funções de criptografia amplamente usadas dos dois tipos.

Algoritmos simétricos ◊ Se eliminarmos da consideração o parâmetro chave, definindo $F_K([M]) = C(K, M)$, então, uma propriedade das funções fortes de criptografia é que $F_K([M])$ é relativamente fácil de calcular, enquanto a função inversa, $F^{-1}_K([M])$, é tão difícil de calcular que não é exeqüível. Tais funções são conhecidas como funções de mão única. A eficácia de qualquer método para cifrar informações depende do uso de uma função de criptografia F_K que tenha essa propriedade da mão única. É isso que faz a proteção contra ataques projetados para descobrir *M*, dado $\{M\}_K$.

No caso de algoritmos simétricos bem projetados, como aqueles descritos na próxima seção, seu poder contra tentativas de descobrir *K*, dado um texto puro *M* e o texto cifrado correspondente $\{M\}_K$, depende do tamanho de *K*. Isso porque a forma geral mais eficiente de ataque é a mais grosseira, co-

nhecida como *ataque de força bruta*. A estratégia da força bruta é examinar todos os valores possíveis de K, calculando $C(K, M)$ até que o resultado corresponda ao valor de $\{M\}_K$ que já é conhecido. Se K tiver N bits, então tal ataque exigirá 2^{N-1} iterações, em média, e um máximo de 2^N iterações, para encontrar K. Assim, o tempo para violar K é exponencial ao número de bits em K.

Algoritmos assimétricos ◊ Quando é usado um par de chaves pública/privada, as funções de mão única são exploradas de outra maneira. A exeqüibilidade de um esquema de chave pública foi proposta pela primeira vez por Diffie e Hellman [1976], como um método de criptografia que elimina a necessidade de confiança entre as partes comunicantes. A base de todos os esquemas de chave pública é a existência de *funções de alçapão* (*trap-door functions*). Uma função de alçapão é uma função de mão única com uma saída secreta – ela é fácil de calcular em uma direção, mas impossível de calcular seu inverso, a não ser que um segredo seja conhecido. Diffie e Hellman foram os primeiros a sugerir a possibilidade de encontrar tais funções e usá-las de forma prática na criptografia. Desde então, vários esquemas de chave pública foram propostos e desenvolvidos. Todos dependem do uso, como funções de alçapão, de funções envolvendo grandes números.

O par de chaves necessário para os algoritmos assimétricos é derivado de uma raiz comum. Para o algoritmo RSA, descrito na Seção 7.3.2, a raiz é um par de números primos muito grandes, escolhidos arbitrariamente. A derivação do par de chaves a partir da raiz é uma função de mão única. No caso do algoritmo RSA, os números primos são multiplicados – um cálculo que leva apenas alguns segundos, mesmo para os números primos muito grandes utilizados. Evidentemente, o produto resultante, N, é muito maior do que os multiplicandos. Esse uso da multiplicação é uma função de mão única, pois é praticamente impossível, em termos computacionais, inferir os multiplicandos originais a partir do produto – isto é, decompor o produto em fatores.

Um dos pares de chaves é usado para criptografia. Para o RSA, a função de criptografia oculta o texto puro, tratando cada bloco de bits como um número binário e elevando-o à potência da chave, módulo N. O número resultante é o bloco de texto cifrado correspondente.

O tamanho de N, e de pelo menos um dos pares de chaves, é muito maior do que o tamanho seguro para chaves simétricas para garantir que N não possa ser decomposto em fatores. Por esse motivo, o potencial de ataques de força bruta no RSA é pequeno; sua resistência aos ataques depende da impossibilidade de decompor N em fatores. Vamos discutir os tamanhos seguros para N na Seção 7.3.2.

Cifras de bloco ◊ A maioria dos algoritmos de criptografia opera em blocos de dados de tamanho fixo; 64 bits é um tamanho popular para os blocos. Uma mensagem é subdividida em blocos; se necessário, o último bloco é preenchido no comprimento padrão e cada bloco é cifrado independentemente. O primeiro bloco estará disponível para transmissão assim que tiver sido cifrado.

Para uma cifra de bloco simples, o valor de cada bloco de texto cifrado não depende dos blocos precedentes. Isso constitui uma deficiência, pois um invasor pode reconhecer padrões repetidos e inferir seu relacionamento com o texto puro. A integridade das mensagens também não é garantida, a não ser que seja usada uma soma de verificação ou um mecanismo de resumo seguro. A maioria dos algoritmos de cifra de bloco emprega encadeamento de blocos de cifra, ou simplesmente, CBC, (*Cypher Block Chaining*) para superar essas deficiências.

Encadeamento de blocos de cifra: no modo de encadeamento de blocos de cifra, cada bloco de texto puro é combinado com o bloco de texto cifrado precedente, usando a operação ou-exclusivo (XOR) antes de ser cifrado (Figura 7.5). Na decifração, cada bloco é decifrado e aplicado nele uma função XOR com o bloco cifrado precedente (que deve ter sido armazenado para esse propósito), para obter o novo bloco de texto puro. Isso funciona porque a operação XOR é sua própria inversa – duas aplicações dela produzem o valor original.

O CBC se destina a impedir que partes idênticas de texto puro sejam cifradas em partes idênticas de texto cifrado. Mas existe uma deficiência no início de cada seqüência de blocos – se abrirmos conexões cifradas para dois destinos e enviarmos a mesma mensagem, as seqüências cifradas de blocos serão as mesmas e um intruso poderá obter algumas informações úteis a partir disso. Para impedir isso, precisamos inserir um trecho de texto puro diferente na frente de cada mensagem. Tal texto é chamado de *vetor de inicialização*. Um valor de hora (*timestamp*) constitui um bom vetor de inicialização, obrigando cada mensagem a começar com um bloco de texto puro diferente. Isso, combinado com a operação CBC, resultará em diferentes textos de cifra, mesmo para dois textos puros idênticos.

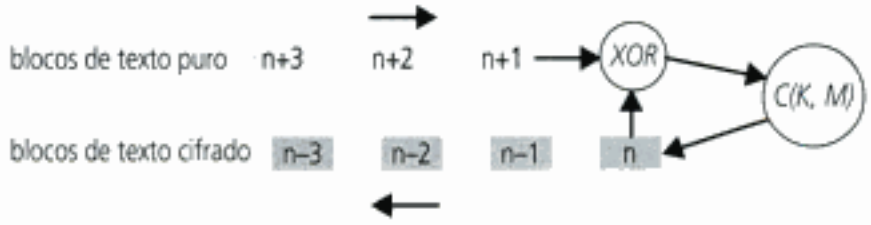


Figura 7.5 Encadeamento de blocos de cifra.

O uso do modo CBC está restrito à criptografia de dados que são transferidos em uma conexão confiável. A decifração falhará se quaisquer blocos de texto cifrado forem perdidos, pois o processo de decifração não conseguirá decifrar mais nenhum bloco. Portanto, ele é inconveniente para uso em aplicações como as descritas no Capítulo 15, nas quais alguns dados perdidos podem ser tolerados. Em tais circunstâncias, deve ser usada uma cifra de fluxo (*stream*).

Cifras de fluxo (*stream*) ◊ Para algumas aplicações, como codificar conversas telefônicas, a criptografia em blocos é inadequada, pois os fluxos de dados são produzidos em tempo real, em pequenos trechos. As amostras de dados podem ser de apenas 8 bits, ou mesmo de um bit, e seria um desperdício preencher cada uma delas com 64 bits antes de cifrá-las e transmiti-las. As cifras de fluxos são algoritmos de criptografia que podem fazer criptografia de forma incremental, convertendo texto puro em texto cifrado, um bit por vez.

Isso parece difícil de conseguir, mas na verdade é muito simples converter um algoritmo de cifra de bloco para usar como uma cifra de fluxo. O truque é construir um *gerador de fluxo de chaves*. Um fluxo de chaves é uma seqüência de bits de comprimento arbitrário que pode ser usada para ocultar o conteúdo de um fluxo de dados, aplicando a operação XOR do fluxo de chaves com o fluxo de dados (Figura 7.6). Se o fluxo de chaves for seguro, então o fluxo de dados cifrado resultante também será.

A idéia é semelhante a uma técnica usada pelos serviços de inteligência para frustrar intrusos, onde um ruído branco é produzido para ocultar a conversa em uma sala, enquanto sua gravação é feita. Se o som da sala onde há a conversa e o ruído branco forem gravados separadamente, a conversa poderá ser reproduzida sem ruído, subtraindo-se a gravação do ruído branco da gravação da sala onde há a conversa.

Um gerador de fluxo de chaves pode ser construído pela iteração de uma função matemática sobre um intervalo de valores de entrada para produzir um fluxo contínuo de valores de saída. Os valores de saída são então concatenados para compor blocos de texto puro, e os blocos são cifrados usando uma chave compartilhada pelo remetente e pelo receptor. O fluxo de chaves pode ser ainda mais dissimulado pela aplicação do CBC. Os blocos cifrados resultantes são usados como fluxo de chaves. Em princípio, qualquer função que resulte em uma variedade de valores não inteiros diferentes serve, mas geralmente um gerador de números aleatórios é usado com um valor inicial combinado entre o emissor e o receptor. Para manter a qualidade de serviço do fluxo de dados, os blocos de fluxo de chaves devem ser produzidos pouco antes de serem usados, e o processo que os gera não deve exigir um trabalho de processamento tal que o fluxo de dados seja retardado.

Assim, em princípio, fluxos de dados em tempo real podem ser cifrados com a mesma segurança que os dados colocados em lotes, desde que esteja disponível um poder de processamento suficiente

Figura 7.6 Cifra de fluxo.

para cifrar o fluxo de chaves em tempo real. É claro que alguns dispositivos que poderiam tirar proveito da criptografia em tempo real, como por exemplo, os telefones móveis, não são equipados com processadores muito poderosos e, nesse caso, talvez seja necessário reduzir a segurança do algoritmo de fluxo de chaves.

Projeto de algoritmos de criptografia ◊ Existem muitos algoritmos de criptografia bem projetados, de modo que $C(K, M) = \{M\}_K$ oculte o valor de M e torne praticamente impossível recuperar K mais rapidamente do que pela força bruta. Todos os algoritmos de criptografia contam com as manipulações de preservação das informações de M, usando princípios baseados na teoria da informação [Shannon 1949]. Schneier [1996] descreve os princípios da *mistura* e da *difusão* de Shannon para ocultar o conteúdo de um bloco de texto cifrado M, combinando-o com uma chave K de tamanho suficiente para submetê-lo à prova contra ataques de força bruta.

Mistura: operações não destrutivas, como XOR e deslocamento circular (*circular shifting*), são usadas para combinar cada bloco de texto puro com a chave, produzindo um novo padrão de bits que oculta o relacionamento entre os blocos em M e $\{M\}_K$. Se os blocos forem maiores do que alguns caracteres, isso anulará a análise baseada no conhecimento de freqüências de caractere. (A máquina WWII German Enigma usava blocos de uma letra encadeados e isso podia ser anulado pela análise estatística.)

Difusão: normalmente, existe repetição e redundância no texto puro. A difusão elimina os padrões regulares resultantes da transposição de partes de cada bloco de texto puro. Se for usado o CBC, a redundância também será distribuída por todo um texto maior. As cifras de fluxo não podem usar difusão, pois não existem blocos.

Nas duas próximas seções, descreveremos o projeto de vários algoritmos práticos importantes. Todos foram projetados de acordo com os princípios anteriores, foram sujeitos a análise rigorosa e são considerados seguros contra todos os ataques conhecidos com uma considerável margem de segurança. Com exceção do algoritmo TEA, que será apresentado para propósitos ilustrativos, os algoritmos descritos aqui estão entre os mais utilizados em aplicações onde é exigida uma segurança forte. Em alguns deles permanecem algumas pequenas deficiências, ou áreas de preocupação; o espaço aqui não nos permite descrever todas essas preocupações e o leitor deve consultar Schneier [1996] para obter mais informações. Resumiremos e compararemos a segurança e o desempenho dos algoritmos na Seção 7.5.1.

Os leitores que não precisarem entender o funcionamento dos algoritmos de criptografia podem omitir as seções 7.3.1 e 7.3.2.

7.3.1 Algoritmos de chave secreta (simétricos)

Muitos algoritmos de criptografia foram desenvolvidos e publicados nos últimos anos. Schneier [1996] descreve mais de 25 algoritmos simétricos, muitos dos quais ele identifica como seguros contra ataques conhecidos. Aqui, temos espaço para descrever apenas três deles. Escolhemos o primeiro, o TEA, pela simplicidade de seu projeto e implementação, e o utilizamos para fornecer uma ilustração concreta da natureza de tais algoritmos. Vamos discutir os algoritmos DES e IDEA com menos detalhes. O DES foi, por muitos anos, o padrão nacional americano, mas agora ele tem principalmente interesse histórico, pois suas chaves de 56 bits são pequenas demais para resistir aos ataques de força bruta feitos com os modernos computadores. O IDEA usa uma chave de 128 bits e é um dos algoritmos de criptografia de bloco simétrico mais eficazes, sendo uma boa escolha comprovada para criptografia de grandes volumes de dados.

Em 1997, O *National Institute for Standards and Technology* (NIST) dos EUA divulgou um convite para propostas de um algoritmo para substituir o DES como um novo padrão de criptografia avançada (*Advanced Encryption* Standard – AES) daquele país. Em outubro de 2000, foi escolhido o vencedor dentre 21 algoritmos enviados por profissionais da área de criptografia de 11 países. O algoritmo vencedor, Rijndael, foi escolhido por sua combinação de poder e eficiência. Mais informações sobre ele serão dadas a seguir.

TEA ◊ Os princípios de projeto dos algoritmos simétricos apresentados anteriormente são bem ilustrados no *Tiny Encryption Algorithm*, desenvolvido na Universidade de Cambridge [Wheeler e Needham 1994]. A função de criptografia, programada em C, aparece em sua totalidade na Figura 7.7.

```
void encrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];                          1
    unsigned long delta = 0x9e3779b9, sum = 0; int n;                2
    for (n= 0; n < 32; n++) {                                        3
        sum += delta;                                                4
        y += ((z << 4) + k[0]) ^ (z+sum) ^ ((z >> 5) + k[1]);        5
        z += ((y << 4) + k[2]) ^ (y+sum) ^ ((y >> 5) + k[3]);        6
    }
    text[0] = y; text[1] = z;                                        7
}
```

Figura 7.7 Função de criptografia TEA.

O algoritmo TEA usa várias passagens de adição de inteiros, combinados com XOR (o operador ^) e deslocamentos lógicos em nível de bit (<< e >>) para obter mistura e difusão dos padrões de bit no texto puro. O texto puro é um bloco de 64 bits representado como dois inteiros de 32 bits no vetor *text[]*. A chave tem 128 bits de comprimento, representada como quatro inteiros de 32 bits.

Em cada uma das 32 passagens, as duas metades do texto são repetidamente combinadas com as partes deslocadas da chave e uma com as outras, nas linhas 5 e 6. O uso da operação XOR e das partes deslocadas do texto proporciona a mistura e o deslocamento. A troca das duas partes do texto proporciona a difusão. A constante *delta*, que não se repete, é combinada, em cada ciclo, com cada parte do texto para ocultar a chave. Isso para evitar o caso dela poder ser revelada por uma seção de texto que não varie. A função de decifração é a inversa da função de criptografia e aparece na Figura 7.8.

Esse programa curto fornece criptografia de chave secreta de forma segura e razoavelmente rápida. Ele é bem mais rápido do que o algoritmo DES, e o caráter conciso do programa se presta à otimização e à implementação do hardware. A chave de 128 bits é segura contra ataques de força bruta. Estudos realizados por seus autores e por outros pesquisadores revelaram apenas duas deficiências muito pequenas, as quais eles trataram em uma nota subseqüente [Wheeler e Needham 1997].

Para ilustrar seu uso, a Figura 7.9 mostra um programa simples que utiliza TEA para cifrar ou decifrar dois arquivos previamente abertos (usando a biblioteca C *stdio*).

DES ◊ O DES (*Data Encryption Standard*) [National Bureau of Standards 1977] foi desenvolvido pela IBM e, subseqüentemente, adotado como padrão nacional nos EUA para aplicações governamentais e comerciais. Nesse padrão, a função de criptografia mapeia uma entrada de texto puro de 64 bits em uma saída cifrada de 64 bits, usando uma chave de 56 bits. O algoritmo tem 16 estágios dependentes de chave, conhecidos como *passagens*, nos quais os dados a serem cifrados são rotacionados por um número de bits determinado pela chave e por três transposições independentes de chave. O algoritmo demorava muito para ser executado por software nos computadores dos anos 70 e 80, e por

```
void decrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];
    unsigned long delta = 0x9e3779b9, sum = delta << 5; int n;

    for (n= 0; n < 32; n++) {
        z -= ((y << 4) + k[2]) ^ (y + sum) ^ ((y >> 5) + k[3]);
        y -= ((z << 4) + k[0]) ^ (z + sum) ^ ((z >> 5) + k[1]);
        sum -= delta;

    }
    text[0] = y; text[1] = z;
}
```

Figura 7.8 Função de decifração TEA.

```
void tea(char mode, FILE *infile, FILE *outfile, unsigned long k[]) {
/* mode é 'c' para cifrar, 'd' para decifrar, k[] é a chave.*/
    char ch, Text[8]; int i;

    while(!feof(infile)) {
        i = fread(Text, 1, 8, infile);     /* lê 8 bytes de infile para Text */
        if (i <= 0) break;
        while (i < 8) { Text[i++] = ' ';}       /* preenche o último bloco com espaços */
        switch (mode) {
        case 'c':
            encrypt(k, (unsigned long*) Text); break;
        case 'd':
            decrypt(k, (unsigned long*) Text); break;
        }
        fwrite(Text, 1, 8, outfile);  /* grava 8 bytes de Text em outfile */
    }
}
```

Figura 7.9 TEA em uso.

isso foi implementado diretamente em hardware VLSI. Por ser um chip VLSI, além de realizar rapidamente o cálculo DES, permitiu sua fácil incorporação em interfaces de rede e em outros hardware de comunicação.

Em junho de 1997, ele foi violado com êxito em um ataque de força bruta amplamente divulgado. O ataque foi realizado no contexto de uma competição para demonstrar a falta de segurança da criptografia com chaves menores do que 128 bits [www.rsasecurity.com I]. Para isso, um consórcio de usuários da Internet executou um programa aplicativo cliente em vários computadores (PCs e outras estações de trabalho). Eles iniciaram com 1000 máquinas clientes e aumentaram seu número até atingir 14.000 máquinas[Curtin e Dolske 1998].

O programa cliente tinha como objetivo violar uma chave particular usada em uma amostra conhecida de texto puro/texto cifrado e depois usá-la para decifrar uma mensagem de desafio secreta. Os clientes interagiam com um único servidor, o qual coordenava o trabalho deles, divulgando a cada cliente os intervalos de valores de chave a verificar e recebendo deles relatórios de progresso. O computador cliente típico executava o programa cliente como uma atividade em *background* e tinha um desempenho aproximadamente igual ao de um processador Pentium de 200 MHz. A chave foi violada em cerca de 12 semanas, após aproximadamente 25% dos 2^{56} ou 6×10^{16} valores possíveis terem sido verificados. Em 1998, foi desenvolvida uma máquina pela Electronic Frontier Foundation [EFF 1998] que conseguiu violar com êxito chaves DES em cerca de três dias.

Embora ainda seja usado em muitos aplicativos comerciais e outros, em sua forma básica, o DES deve ser considerado obsoleto para a proteção de informações, a não ser as de baixo valor. Uma solução freqüentemente usada é conhecida como *triple-DES* (ou *3DES*) [ANSI 1985, Schneier 1996]. Ela envolve a aplicação do DES três vezes, com duas chaves K_1 e K_2:

$$C_{3DES}(K_1, K_2, M) = C_{DES}(K_1, D_{DES}(K_2, C_{DES}(K_1, M)))$$

Isso proporciona uma resistência contra ataques de força bruta equivalente a um comprimento de chave de 112 bits, já prevendo um certo aumento futuro do poder computacional das máquinas. Entretanto, tal algoritmo tem o inconveniente do mal desempenho, resultante da aplicação tripla de um algoritmo que já é lento pelos padrões modernos.

IDEA ◊ O *International Data Encryption Algorithm* foi desenvolvido, no início dos anos 90 [Lai e Massey 1990, Lai 1992], como um sucessor do DES. Assim como o TEA, ele usa uma chave de 128 bits para cifrar blocos de 64 bits. Seu algoritmo é baseado na álgebra de grupos e tem oito passagens de XOR, adição módulo 2^{16} e multiplicação. Tanto para o DES como para o IDEA, a mesma função é usada para cifrar e decifrar: uma propriedade útil para algoritmos que são implementados em hardware.

O poder do IDEA foi amplamente analisado e nenhuma deficiência significativa foi encontrada. Ele cifra e decifra com aproximadamente três vezes a velocidade do DES.

RC4 ◊ O RC4 é uma cifra de fluxo desenvolvida por Ronald Rivest [Rivest 1992a]. As chaves podem ter qualquer comprimento de até 256 bytes. O RC4 é fácil de implementar [Schneier 1996, pp. 397–8] e cifra e decifra cerca de dez vezes mais rápido que o DES. Conseqüentemente, ele foi amplamente adotado em vários produtos, incluindo as redes IEEE 802.11 WiFi, mas uma deficiência que permitia aos invasores violar algumas chaves foi descoberta por Fluhrer *et al.* [2001] e isso levou a um reprojeto da segurança do padrão 802.11 (veja a Seção 7.6.4 para mais detalhes).

AES ◊ O algoritmo Rijndael, selecionado para se tornar o algoritmo padrão de criptografia avançada pelo NIST dos EUA, foi desenvolvido por Joan Daemen e Vincent Rijmen [Daemen e Rijmen 2000, 2002]. A cifra tem comprimento de bloco e de chave variável, com especificações para chaves com comprimento de 128, 192 ou 256 bits para cifrar blocos com comprimento de 128, 192 ou 256 bits. Tanto o comprimento do bloco como o comprimento da chave podem ser ampliados por múltiplos de 32 bits. O número de passagens no algoritmo varia de 9 a 13, dependendo dos tamanhos da chave e do bloco. O Rijndael pode ser implementado eficientemente em uma grande variedade de processadores e em hardware.

7.3.2 Algoritmos de chave pública (assimétricos)

Apenas alguns esquemas de chave pública foram desenvolvidos até agora. Eles dependem do uso de funções de alçapão em grandes números para produzir as chaves. As chaves K_c e K_d representam um par de grandes números e a função de criptografia executa uma operação, como a exponenciação, em M, usando um deles. A decifração é uma função semelhante, usando a outra chave. Se a exponenciação usar aritmética de módulo, pode ser mostrado que o resultado é igual ao valor original de M; isto é:

$$D(K_d, C(K_c, M)) = M$$

Um principal que queira participar de uma comunicação segura com outros compõe um par de chaves, K_c e K_d, e mantém em segredo a chave de decifração K_d. A chave de criptografia K_c é tornada de conhecimento público para uso de quem quiser se comunicar. A chave de criptografia K_c pode ser vista como uma parte da função de criptografia de mão única C, e a chave de decifração K_d é o conhecimento secreto que permite ao principal p reverter a criptografia. Qualquer portador de K_c (que está amplamente disponível) pode cifrar mensagens $\{M\}\ K_c$, mas somente o principal que tem a K_d secreta pode reverter a operação.

O uso de funções de grandes números implica em elevados custos de processamento para o cálculo das funções C e D. Posteriormente, veremos que esse é um problema que faz com que as chaves públicas sejam usadas apenas nos estágios iniciais das sessões de comunicação seguras. Certamente, o algoritmo RSA é o algoritmo de chave pública mais conhecido, e o descreveremos com alguns detalhes aqui. Outra classe de algoritmos é baseada em funções derivadas do comportamento das curvas elípticas em um plano. Esses algoritmos oferecem, com o mesmo nível de segurança, a possibilidade de funções de criptografia e decifração menos dispendiosas, mas sua aplicação prática está menos avançada e vamos tratar deles apenas sucintamente.

RSA ◊ O projeto de cifra de chave pública de Rivest, Shamir e Adelman (RSA) [Rivest *et al.* 1978] é baseado no uso do produto de dois números primos muito grandes (maiores do que 10^{100}), contando com o fato de que a determinação dos fatores primos de tais números grandes é tão difícil, em termos computacionais, que se torna efetivamente impossível de calcular.

Apesar das extensivas investigações, nenhuma falha foi encontrada e, agora, o RSA é amplamente usado. A seguir apresentamos um esboço do método. Para encontrar um par de chaves c, d:

1. Escolha dois números primos grandes, P e Q (cada um maior do que 10^{100}) e forme
   $N = P \times Q$
   $Z = (P - 1) \times (Q - 1)$
2. Para d, escolha qualquer número que seja relativamente primo de Z (isto é, de modo que d não tenha fatores comuns com Z).

Ilustramos os cálculos envolvidos usando valores inteiros pequenos para P e Q:

$P = 13, Q = 17 \rightarrow N = 221, Z = 192$

$d = 5$

3. Para encontrar c, resolva a equação:
   $c \times d = 1 \ mod \ Z$

Isto é, $c \times d$ é o menor elemento divisível por d na série Z+1, 2Z+1, 3Z+1, ...

$c \times d = 1 \ mod \ 192 = 1, 193, 385, ...$

385 é divisível por d

$c = 385/5 = 77$

Para cifrar texto usando o método RSA, o texto puro é dividido em blocos iguais de k bits de comprimento, onde $2^k < N$ (isto é, de modo que o valor numérico de um bloco é sempre menor do que N; nas aplicações práticas, k normalmente está no intervalo de 512 a 1024).

$k = 7$, pois $2^7 = 128$

A função para cifrar um único bloco de texto puro M é:

$C'(c,N,M) = M^c \ mod \ N$

para uma mensagem M, o texto cifrado é $M^{77} \ mod \ 221$

A função para decifrar um bloco de texto cifrado C, para produzir o bloco de texto puro original, é:

$D'(d,N,C) = C^d \ mod \ N$

Rivest, Shamir e Adelman provaram que C' e D' são inversos mútuos (isto é, $C'(D'(x)) = D'(C'(x)) = x$) para todos os valores de P no intervalo $0 \leq P \leq N$.

Os dois parâmetros c,N podem ser considerados como uma chave para a função de criptografia e, analogamente, d,N representam uma chave para a função de decifração. Portanto, podemos escrever $K_c = <c,N>$ e $K_d = <d,N>$, e obtermos as funções de criptografia $C(K_c, M) = \{M\}_K$ (a notação aqui indicando que a mensagem cifrada só pode ser decifrada pelo portador da chave privada K_d) e $D(K_d, \{M\}_K) = M$.

É interessante notar uma deficiência em potencial de todos os algoritmos de chave pública – como a chave pública está disponível para invasores, eles podem gerar mensagens cifradas facilmente. Assim, eles podem tentar decifrar uma mensagem desconhecida, cifrando exaustivamente seqüências de bits arbitrárias, até obterem uma correspondência com a mensagem de destino. Esse ataque, conhecido como *ataque de texto puro escolhido*, é anulado garantindo-se que todas as mensagens sejam maiores do que o comprimento da chave, de modo que essa forma de ataque de força bruta é menos exeqüível do que um ataque direto na chave.

O destinatário da informação secreta pretendido deve publicar, ou distribuir de outra forma, o par $<c,N>$, enquanto mantém d em segredo. A publicação de $<c,N>$ não compromete o segredo de d, pois qualquer tentativa de determinar d exige o conhecimento dos números primos originais P e Q e eles só podem ser obtidos pela decomposição dos fatores de N. A decomposição dos fatores de grandes números (lembramos que P e Q foram escolhidos para serem $> 10^{100}$; portanto, $N > 10^{200}$) é extremamente demorada, mesmo em computadores de desempenho muito elevado. Em 1978, Rivest *et al.* concluíram que decompor em fatores um grande número como 10^{200} demoraria mais de quatro bilhões de anos, com o melhor algoritmo conhecido, em um computador que executasse um milhão de instruções por segundo. Um cálculo semelhante para os computadores atuais reduziria esse tempo para cerca de um milhão de anos.

A RSA Corporation publicou uma série de desafios para fatorar números de mais de 100 dígitos decimais [www.rsasecurity.com II]. Quando este livro estava sendo produzido, números de até 174 dígitos decimais (576 dígitos binários) tinham sido decompostos em fatores com êxito; portanto, cla-

ramente o uso do algoritmo RSA com chaves de 512 bits é inaceitavelmente deficiente para muitos propósitos. A RSA Corporation (dona das patentes do algoritmo RSA) recomenda um comprimento de chave de pelo menos 768 bits, ou cerca de 230 dígitos decimais, para segurança a longo prazo (~ 20 anos). Em algumas aplicações, são usadas chaves de até 2.048 bits.

Os cálculos anteriores pressupõem que são usados os melhores algoritmos de fatoração atualmente disponíveis. O RSA e outras formas de criptografia assimétrica que usam multiplicação de números primos como função de mão única serão vulneráveis se, e quando, um algoritmo de decomposição em fatores mais rápido for descoberto.

Algoritmos de curva elíptica ◊ Um método para gerar pares de chaves pública/privada com base nas propriedades das curvas elípticas foi desenvolvido e testado. Os detalhes completos podem ser encontrados no livro de Menezes dedicado ao assunto [Menezes 1993]. As chaves são derivadas de um ramo diferente da matemática e, ao contrário do RSA, sua segurança não depende da dificuldade de decompor grandes números em fatores. As chaves obtidas são mais curtas, seguras e as demandas de processamento para cifrar e decifrar são menores do que para o RSA. Os algoritmos de criptografia de curva elíptica provavelmente serão adotados mais amplamente no futuro, especialmente em sistemas como aqueles que incorporam dispositivos móveis, os quais têm recursos de processamento limitados. A matemática relevante envolve algumas propriedades bastante complexas das curvas elípticas e está fora dos objetivos deste livro.

7.3.3 Protocolos de criptografia mistos

A criptografia de chave pública é conveniente para o comércio eletrônico porque não há necessidade de um mecanismo de distribuição de chaves seguro. (Há necessidade de autenticar as chaves públicas, mas isso é muito menos oneroso, exigindo apenas o envio de um certificado de chave pública com a chave.) Mas os custos de processamento da criptografia de chave pública são altos demais até para cifrar mensagens de tamanho médio, normalmente encontradas no comércio eletrônico. A solução adotada na maioria dos sistemas distribuídos de larga escala é usar um esquema de criptografia misto, no qual a criptografia de chave pública é usada para autenticar as partes e para cifrar a troca de chaves secretas, as quais são usadas para toda a comunicação subseqüente. Vamos descrever a implementação de um protocolo misto no estudo de caso sobre TLS, na Seção 7.6.3.

7.4 Assinaturas digitais

Assinaturas digitais fortes são um requisito essencial para os sistemas seguros. Elas são necessárias para certificar determinadas informações; por exemplo, para fornecer declarações confiáveis, vinculando as identidades dos usuários às suas chaves públicas, ou vinculando alguns direitos de acesso e funções às identidades dos usuários.

A necessidade de assinaturas em muitos tipos de transações comerciais e pessoais é inquestionável. Assinaturas manuscritas têm sido usadas como uma maneira de verificação desde que os documentos existem. Elas são usadas para atender as necessidades dos destinatários do documento, de verificar se ele é:

Autêntico: convence o destinatário de que o signatário deliberadamente assinou o documento e que ele não foi alterado por ninguém.

Impossível de falsificar: fornece uma prova de que o signatário, e ninguém mais, deliberadamente assinou o documento. A assinatura não pode ser copiada, nem colocada em outro documento.

Impossível de repudiar: o signatário não pode negar que o documento foi assinado por ele.

Na realidade, nenhuma dessas propriedades desejáveis da assinatura é inteiramente obtida com as assinaturas convencionais – pois falsificações e cópias são difíceis de detectar, os documentos podem ser alterados após sua assinatura e os signatários, às vezes, são enganados, assinando um documento

involuntária ou inconscientemente. Entretanto, desejamos conviver com sua imperfeição, devido à dificuldade de fraudar e ao risco de detecção. Assim como as assinaturas manuscritas, as assinaturas digitais dependem do vínculo de um atributo único e secreto do signatário com um documento. No caso das assinaturas manuscritas, o segredo é o padrão manuscrito do signatário.

As propriedades dos documentos digitais, mantidas em arquivos armazenados ou em mensagens, são completamente diferentes dos documentos em papel. Os documentos digitais são extremamente fáceis de gerar, copiar e alterar. Simplesmente anexar a identidade do criador, seja como um string de texto, uma fotografia ou uma imagem manuscrita, não tem nenhum valor para propósitos de verificação.

O que é necessário é uma maneira de vincular irrevogavelmente a identidade de um signatário à seqüência de bits inteira que representa um documento. Isso deve satisfazer o primeiro requisito acima, o da autenticidade. Assim como acontece com as assinaturas manuscritas, a data de um documento não pode ser garantida por uma assinatura. O destinatário de um documento assinado sabe apenas que ele foi assinado antes de tê-lo recebido.

A respeito do não-repúdio, há um problema que não surge com as assinaturas manuscritas: e se o signatário revelar deliberadamente sua chave privada e, subseqüentemente, negar que o assinou, alegando que existem outras pessoas que poderiam ter feito isso, porque a chave não era realmente privada? Foram desenvolvidos alguns protocolos para tratar desse problema, sob o título de *assinaturas digitais inegáveis* [Schneier 1996], mas eles aumentam a complexidade consideravelmente.

Um documento com uma assinatura digital pode ser consideravelmente mais resistente à falsificação do que outro com assinatura manuscrita. Mas a palavra 'original' tem pouco significado com referência aos documentos digitais. Conforme veremos em nossa discussão sobre as necessidades do comércio eletrônico, as assinaturas digitais sozinhas não podem, por exemplo, impedir um saque duplo no caixa eletrônico – outras medidas são necessárias para evitar isso. Vamos descrever agora duas técnicas para assinar documentos de forma digital, vinculando a identidade de um principal ao documento. Ambas dependem do uso de criptografia.

Assinatura digital ◊ Um documento, ou mensagem eletrônica M, pode ser assinada por um principal A cifrando-se uma cópia de M com uma chave K_A e anexando-se uma cópia de texto puro de M e o identificador de A. O documento assinado consiste então em: M, A, $[M]k_A$. A assinatura pode ser verificada por um principal que, subseqüentemente, recebe o documento para conferir se ele foi originado por A e se seu conteúdo, M, não foi alterado.

Se uma chave secreta for usada para cifrar o documento, apenas os principais que compartilharem o segredo poderão verificar a assinatura. Mas, se for usada criptografia de chave pública, então o signatário usará sua chave privada e qualquer um que possua a chave pública correspondente poderá verificar a assinatura. Isso é melhor que as assinaturas convencionais e atende uma variedade mais ampla das necessidades do usuário. A verificação de assinaturas prossegue de diferentes formas, dependendo se é usada criptografia de chave secreta, ou de chave pública, para produzir a assinatura. Descreveremos os dois casos nas Seções 7.4.1 e 7.4.2.

Funções de resumo (*digest*) ◊ As funções de resumo também são chamadas de *funções de hashing seguras* e denotadas como $H(M)$. Elas devem ser cuidadosamente projetadas para garantir que $H(M)$ seja diferente de $H(M')$ para todos os prováveis pares de mensagens M e M'. Se houver quaisquer pares de mensagens M e M' diferentes, tal que $H(M) = H(M')$, então um principal fraudulento poderia enviar uma cópia assinada de M, mas quando confrontada com ela, reivindicar que M' foi originalmente enviada e que ela deve ter sido alterada em trânsito. Discutiremos algumas funções de *hashing* seguras na Seção 7.4.3.

7.4.1 Assinaturas digitais com chaves públicas

A criptografia de chave pública é particularmente bem adaptada para a geração de assinaturas digitais, pois ela é relativamente simples e não exige nenhuma comunicação entre o signatário e o destinatário de um documento assinado, ou outra pessoa qualquer.

O método para A assinar uma mensagem M e B verificá-la é o seguinte (e está ilustrado graficamente na Figura 7.10):

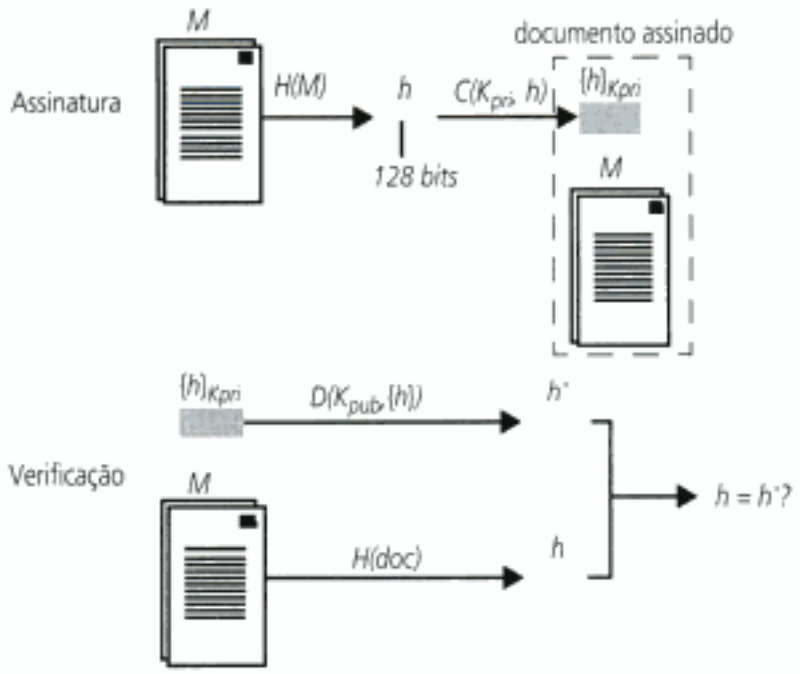


Figura 7.10 Assinatura digital com chaves públicas.

1. *A* gera um par de chaves K_{pub} e K_{priv} e publica a chave pública K_{pub}, colocando-a em um local bem conhecido.
2. *A* calcula o resumo de *M*, *H*(*M*), usando uma função de resumo segura *H* e previamente combinada com *B*, o resumo é cifrado com a chave privada K_{priv} para produzir a assinatura $S = \{H(M)\}k_{priv}$.
3. *A* envia a mensagem assinada $[M]_K = M,S$ para *B*.
4. *B* decifra *S* usando K_{pub} e calcula o resumo de *M*, *H*(*M*). Se eles corresponderem, a assinatura é válida.

O algoritmo RSA é muito conveniente para a construção de assinaturas digitais. Note que a chave *privada* do signatário é usada para cifrar a assinatura, diferentemente de quando o objetivo é transmitir informações em segredo, em que é usado a chave *pública* do destinatário para cifrar a mensagem. A explicação dessa diferença é simples e direta – uma assinatura deve ser criada usando-se um segredo conhecido apenas pelo signatário e ela deve estar acessível para todos para verificação.

7.4.2 Assinaturas digitais com chaves secretas – MACs

Não existe nenhum motivo técnico pelo qual um algoritmo de criptografia de chave secreta não possa ser usado para cifrar uma assinatura digital, mas para verificar tais assinaturas, a chave precisa ser revelada, e isso causa alguns problemas:

- O signatário deve fazer preparativos para o verificador receber com segurança a chave secreta usada para assinatura.
- Talvez seja necessário verificar uma assinatura em vários contextos e em diferentes momentos – na hora da assinatura, o signatário pode não saber as identidades dos verificadores. Para resolver isso, a verificação poderia ser delegada a um terceiro confiável que possua as chaves secretas de todos os signatários, mas isso aumenta a complexidade do modelo de segurança e exige uma comunicação segura com o terceiro confiável.
- A exposição de uma chave secreta usada para assinar é indesejável, pois isso enfraquece a segurança das assinaturas feitas com essa chave – uma assinatura poderia ser falsificada por um possuidor da chave que não seja o proprietário dela.

Por todos esses motivos, o método de chave pública para geração e verificação de assinaturas oferece a solução mais conveniente na maioria das situações.

Uma exceção surge quando um canal seguro é usado para transmitir mensagens não cifradas, mas com a necessidade de verificar a autenticidade das mensagens. Como o canal fornece comunicação segura entre dois processos, uma chave secreta compartilhada pode ser estabelecida usando-se o método misto apresentado na Seção 7.3.3, e com ela produzir assinaturas de baixo custo. Essas assinaturas são chamadas de *códigos de autenticação de mensagem* (MAC – *Message Authentication Codes*) para refletir seu propósito mais limitado – elas autenticam a comunicação entre pares de principais com base em um segredo compartilhado.

Uma técnica de assinatura de baixo custo, baseada em chaves secretas compartilhadas, que tem segurança adequada para muitos propósitos, está ilustrada na Figura 7.11 e delineada a seguir. O método depende da existência de um canal seguro pelo qual a chave compartilhada possa ser distribuída:

1. *A* gera uma chave aleatória *K* para assinatura e a distribui usando canais seguros para um ou mais principais que autenticarão as mensagens recebidas de *A*. Os principais são *confiáveis* e não vão revelar a chave compartilhada.
2. Para qualquer documento *M* que *A* queira assinar, *A* concatena *M* com *K*, calcula o resumo do resultado $h = H(M + K)$ e envia o documento assinado $[M]_k = M, h$ para todos que quiserem verificar a assinatura. (O resumo *h* é um MAC). *K* não será comprometida pela revelação de *h*, pois a função de resumo ocultou seu valor totalmente.
3. O receptor, *B*, concatena a chave secreta *K* com o documento *M* recebido e calcula o resumo $h' = H(M + K)$. A assinatura é verificada se $h = h'$.

Embora esse método sofra das desvantagens listadas anteriormente, ele tem uma vantagem de desempenho, pois não envolve nenhuma criptografia. (Normalmente, o resumo seguro é cerca de 3–10 vezes mais rápido do que a criptografia simétrica, veja a Seção 7.5.1.) O protocolo de canal seguro TLS, descrito na Seção 7.6.3, suporta o uso de uma ampla variedade de MACs, incluindo o esquema descrito aqui. O método também é usado no protocolo de caixa eletrônico Millicent, descrito no endereço www.cdk4.net/security, onde é importante manter baixo o custo do processamento para transações de baixo valor.

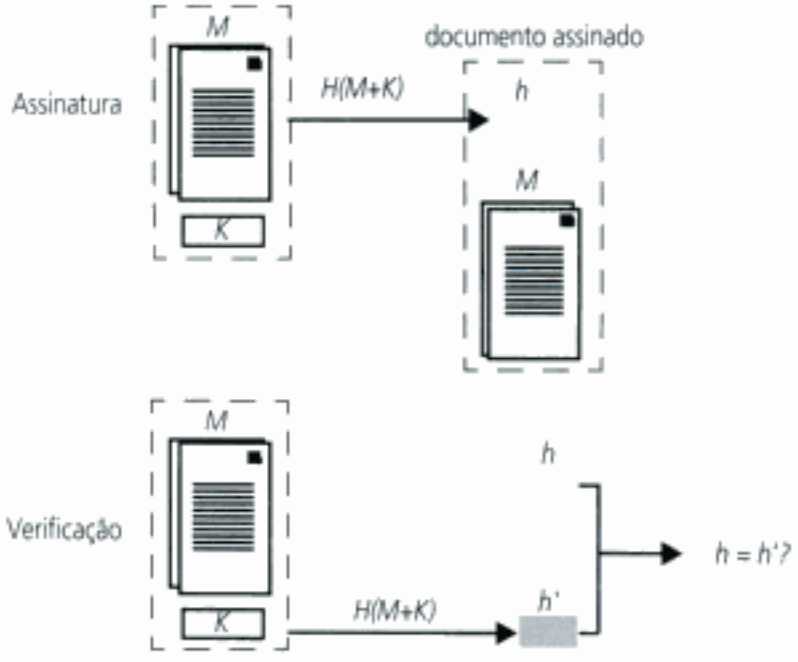


Figura 7.11 Assinaturas de baixo custo com uma chave secreta compartilhada.

7.4.3 Funções de resumo seguras

Existem muitas maneiras de produzir um padrão de bits de comprimento fixo que caracterize uma mensagem ou um documento de comprimento arbitrário. Talvez a mais simples seja usar a operação XOR iterativamente, para combinar partes de comprimento fixo do documento de origem. Tal função é freqüentemente usada em protocolos de comunicação para produzir um resumo de comprimento fixo (curto) para caracterizar uma mensagem com propósitos de detecção de erro, mas ela é inadequada como base para um esquema de assinatura digital. Uma função de resumo segura $h = H(M)$ deve ter as seguintes propriedades:

1. Dado M, é fácil calcular h.
2. Dado h, é difícil calcular M.
3. Dado M, é difícil encontrar outra mensagem M', tal que $H(M) = H(M')$.

Tais funções também são chamadas de *funções de resumo de mão única*. O motivo desse nome é evidente, com base nas duas primeiras propriedades. A propriedade 3 exige uma característica adicional: mesmo sabendo que o resultado de uma função de resumo não pode ser único (porque o resumo é uma transformação de redução da informação), precisamos garantir que um invasor, dada uma mensagem M que produza um resumo h, não possa descobrir outra mensagem M' que também produza h. Se um invasor *pudesse* fazer isso, então ele poderia falsificar um documento assinado M' sem conhecimento da chave de assinatura, copiando a assinatura do documento assinado M e anexando-a em M'.

Admitidamente, o conjunto de mensagens que produz um resumo de mesmo valor é restrito e o invasor teria dificuldade para produzir uma falsificação significativa, mas com paciência, ela poderia ser feita; portanto, é preciso tomar cuidado com isso. A possibilidade de se fazer isso é considerável no caso do *ataque de data de nascimento (birthday attack)*:

1. Alice prepara duas versões, M e M', de um contrato para Bob. M é favorável a Bob e M', não.
2. Alice faz diversas versões sutilmente diferentes de M e de M', que são visualmente indistinguíveis entre si, através de métodos como a adição de espaços nos finais das linhas. Ela compara os resumos de todos os Ms com todos os M's. Se encontrar dois que sejam iguais, ela pode passar para o próximo passo; caso contrário, ela continua produzindo versões visualmente indistinguíveis dos dois documentos, até obter uma correspondência.
3. Quando tiver dois documentos M e M' com resumo e mesmo valor, ela envia o documento M favorável para Bob para que ele produza, usando sua chave privada, uma assinatura digital para esse documento. Quando ele o retorna, ela o substitui pela versão desfavorável M' correspondente, mantendo a assinatura de M.

Se nossos valores de resumo tiverem 64 bits, serão necessárias apenas, em média, de 2^{32} versões de M e M'. Isso é muito pouco para nos tranqüilizar. Precisamos tornar nossos valores de resumo em pelo menos 128 bits para nos protegermos desse ataque.

O ataque conta com um paradoxo estatístico conhecido como *paradoxo da data de nascimento* – a probabilidade de encontrar um par correspondente em determinado conjunto é bem maior do que a de encontrar uma correspondência para determinado indivíduo. Stallings [1999] fornece a derivação estatística da probabilidade de que existam duas pessoas com a mesma data de nascimento em um conjunto de n pessoas. Resumidamente, o resultado é que, para um conjunto de apenas 23 pessoas, as chances de se encontrar duas pessoas que fazem aniversário no mesmo dia são as mesmas para qualquer par de pessoas, enquanto é necessário um conjunto de 253 pessoas para se ter a mesma chance de se encontrar uma pessoa com uma data de nascimento em um determinado dia.

Para satisfazer as propriedades listadas anteriormente, uma função de resumo segura precisa ser cuidadosamente projetada. As operações em nível de bit usadas, e sua disposição em seqüência, são semelhantes àquelas encontradas na criptografia simétrica, mas, neste caso, as operações não precisam preservar as informações, pois a função não é destinada a ser reversível. Portanto, uma função de resumo segura pode usar toda a gama de operações aritméticas e lógicas em nível de bit. O comprimento do texto de origem normalmente é incluído nos dados do resumo.

Duas funções de resumo amplamente usadas para aplicações práticas são o algoritmo MD5 (assim chamado porque ele é o quinto em uma seqüência de algoritmos de resumos de mensagem desenvolvidos por Ron Rivest) e o SHA-1 (*Secure Hash Algorithm*), adotado para padronização pelo *National Institute for Standards and Technology* (NIST) dos EUA. Ambas foram cuidadosamente testadas e analisadas, e podem ser consideradas adequadamente seguras dentro de um futuro próximo, enquanto suas implementações são razoavelmente eficientes. Nós as descreveremos sucintamente aqui. Schneier [1996] e Mitchell *et al.* [1992] fazem um levantamento profundo das técnicas de assinatura digital e das funções de resumo de mensagem.

MD5 ◊ O algoritmo MD5 [Rivest 1992] usa quatro passagens, cada uma aplicando uma de quatro funções não lineares em cada um dos 16 segmentos de 32 bits de um bloco de texto de origem de 512 bits. O resultado é um resumo de 128 bits. O MD5 é um dos algoritmos mais eficientes atualmente disponíveis.

SHA-1 ◊ O SHA-1 [NIST 2002] é um algoritmo que produz um resumo de 160 bits. Ele é baseado no algoritmo MD4 de Rivest (que é semelhante ao MD5), com algumas operações adicionais. Ele é significativamente mais lento do que o MD5, mas o resumo de 160 bits apresenta uma segurança maior contra ataques estilo força bruta e data de nascimento. Algoritmos SHA que geram resumos maiores (224, 256 e 512 bits) também estão incluídos no padrão [NIST 2002]. É claro que seu comprimento adicional acarreta custos a mais para a geração, o armazenamento e a comunicação de assinaturas digitais e MACs, mas após a publicação de ataques nos predecessores do SHA-1, que sugeriram que o SHA-1 é vulnerável [Randall e Szydlo 2004], o NIST anunciou que ele vai ser substituído, em 2010, por versões de resumo do SHA mais longas nos softwares do governo dos EUA [NIST 2004].

Usando um algoritmo de criptografia para fazer um resumo ◊ É possível usar um algoritmo de criptografia simétrico, como aqueles detalhados na Seção 7.3.1, para produzir um resumo seguro. Nesse caso, a chave deve ser publicada para que o algoritmo de resumo possa ser aplicado por qualquer um que queira verificar uma assinatura digital. O algoritmo de criptografia é usado no modo CBC e o resumo é o resultado da combinação do penúltimo valor de CBC com o bloco cifrado final.

7.4.4 Padrões de certificado e autoridades certificadoras

O X.509 é o formato padrão mais usado para certificados [CCITT 1988b]. Embora o formato de certificado X.509 faça parte do padrão X.500 para a construção de diretórios globais de nomes e atributos [CCITT 1988a], ele é comumente usado em trabalhos de criptografia como uma definição de formato para certificados. Descreveremos o padrão de atribuição de nomes X.500 no Capítulo 9.

A estrutura e o conteúdo de um certificado X.509 estão ilustrados na Figura 7.12. Conforme vemos, ele vincula uma chave pública a uma entidade nomeada, chamada de *sujeito* (*subject*). A vinculação se dá na assinatura, que é distribuída por outra entidade nomeada, chamada de *emitente* (*Issuer*). O certificado tem um *período de validade*, que é definido por duas datas. As entradas *Nome Discriminado* (*Distinguished Name*) se destinam a ser o nome de uma pessoa, organização, ou outra entidade, junto com informações contextuais suficientes para torná-las exclusivas. Em uma implementação completa do X.500, essas informações contextuais são extraídas de uma hierarquia global de diretórios na qual a entidade nomeada aparece, mas na ausência de implementações de X.500 globais, elas podem ser apenas um string descritivo.

Sujeito	Nome Discriminado, Chave Pública
Emitente	Nome Discriminado, Assinatura
Período de validade	Não Antes de Data, Não Após Data
Informações administrativas	Versão, Número de Série
Informações Estendidas	

Figura 7.12 Formato do certificado X.509.

Esse formato é incluído no protocolo TLS para comércio eletrônico e, na prática, é amplamente usado para autenticar as chaves públicas de serviços e seus clientes. Certas empresas e organizações bem conhecidas se estabeleceram para atuar como *autoridades certificadoras* (por exemplo, Verisign [www.verisign.com], CREN [www.cren.net]) e, outras empresas e indivíduos, enviando evidência satisfatória de suas identidades, podem obter com elas certificados de chave pública X.509. Isso leva a um procedimento de verificação de duas etapas para qualquer certificado X.509:

1. Obter de uma fonte confiável o certificado de chave pública do emitente (uma autoridade de certificação).
2. Validar a assinatura.

A estratégia SPKI ◊ A estratégia X.509 é baseada na exclusividade global de nomes discriminados. Foi mostrado que esse é um objetivo impraticável que não reflete a realidade da prática jurídica e comercial corrente [Ellison 1996] na qual não se presume que a identidade dos indivíduos seja exclusiva, mas que se torna exclusiva pela referência a outras pessoas e organizações. Isso pode ser visto no uso de uma carteira de motorista, ou de uma carta de um banco, para autenticar o nome e o endereço de um indivíduo (é improvável que apenas um nome seja exclusivo dentre a população mundial). Isso leva a encadeamentos de verificação mais longos, pois existem muitos emitentes possíveis de certificados de chave pública e suas assinaturas devem ser validadas por meio de um encadeamento de verificação que leve de volta a alguém conhecido, e de confiança, do principal que está realizando a verificação. Mas, provavelmente, a verificação resultante será mais convincente, e muitas das etapas desse encadeamento podem ser colocadas em cache para encurtar o processo em ocasiões futuras.

Os argumentos anteriores são a base das propostas SPKI (*Simple Public-key Infrastructure*) recentemente desenvolvidas (veja RFC 2693 [Ellison *et al.* 1999]). Trata-se de um esquema para a criação e gerenciamento de conjuntos de certificados públicos. Ele permite que encadeamentos de certificados sejam processados usando inferência lógica para produzir certificados derivados. Por exemplo, "Bob acredita que a chave pública de Alice é K_{Alice}" e "Carol confia em Bob a respeito das chaves de Alice", implica que "Carol acredita que a chave pública de Alice é K_{Alice}".

7.5 Criptografia na prática

Na Seção 7.5.1, compararemos o desempenho dos algoritmos de criptografia e de resumo seguro descritos ou mencionados anteriormente. Consideraremos os algoritmos de criptografia lado a lado com as funções de resumo seguras, pois a criptografia também pode ser usada como um método de assinatura digital.

Na Seção 7.5.2, discutiremos alguns problemas não técnicos que cercam o uso da criptografia. Não temos espaço para tratar adequadamente o vasto volume de discussão política que tem ocorrido sobre esse assunto desde que os algoritmos de criptografia poderosos apareceram pela primeira vez em domínio público; tampouco os debates chegaram a muitas conclusões definitivas. Nosso objetivo é simplesmente dar ao leitor algum conhecimento desse debate.

7.5.1 Desempenho dos algoritmos de criptografia

A Figura 7.13 mostra a velocidade dos algoritmos de criptografia simétricos e as funções de resumo seguras que discutimos neste capítulo. Onde estão disponíveis, fornecemos duas medidas de velocidade. Na coluna rotulada como *PRB otimizado*, fornecemos valores baseados naqueles publicados por Preneel *et al.* [Preneel *et al.* 1998]. Os valores presentes na coluna rotulada como *Crypto++* foram obtidos muito mais recentemente pelos autores da biblioteca de criptografia Crypto++, de código-fonte aberto [www.cryptopp.com]. Os cabeçalhos de coluna indicam a velocidade do hardware usado para esses comparativos. As implementações de Preneel eram programas em assembly e otimizados manualmente, enquanto as da Crypto++ eram programas em C++ gerados com um compilador com otimização de código.

Os comprimentos de chave dão uma indicação do custo computacional de um ataque de força bruta contra a chave; o poder real dos algoritmos de criptografia é muito mais difícil de avaliar e baseia-se no raciocínio sobre o êxito do algoritmo na ocultação do texto puro. Preneel *et al.* [1998] apresentam uma discussão interessante sobre o poder e o desempenho dos principais algoritmos simétricos.

O que esses valores de desempenho significam para aplicações reais de criptografia, como o seu uso no esquema TLS para interações seguras na web (o protocolo *https*, descrito na Seção 7.6.3)? Raramente as páginas web são maiores do que 100 quilobytes; portanto, o conteúdo de uma página pode ser cifrado em poucos milisegundos usando-se qualquer um dos algoritmos simétricos, mesmo com um processador considerado muito lento pelos padrões atuais. O RSA é usado principalmente para assinaturas digitais e esse passo também pode ser dado em poucos milisegundos. Assim, o impacto do desempenho do algoritmo na velocidade sentida por uma aplicação que usa o protocolo *https* é mínimo.

Os algoritmos assimétricos, como o RSA, raramente são usados para criptografia de dados, mas seu desempenho para assinatura é interessante. As páginas da biblioteca Crypto++ indicam que, com o hardware mencionado na última coluna da Figura 7.13, demora cerca de 4,75 ms, usando RSA com uma chave de 1024 bits, para assinar um resumo seguro (presumivelmente usando SHA-1 de 160 bits) e cerca de 0,18 ms para verificar a assinatura.

7.5.2 Aplicações da criptografia e obstáculos políticos

Todos os algoritmos descritos anteriormente surgiram durante os anos 80 e 90, quando as redes de computadores estavam começando a ser usadas para propósitos comerciais e estava se tornando evidente que sua falta de segurança era um grande problema. Conforme mencionamos na introdução deste capítulo, a revelação do software de criptografia teve forte resistência por parte do governo norte-americano. A resistência tinha duas origens: a Agência de Segurança Nacional dos EUA (NSA), que achava que deveria haver um plano para restringir o poder da criptografia disponível para outros países a um nível que a NSA pudesse decifrar qualquer comunicação secreta para propósitos da inteligência militar, e o FBI (*Federal Bureau of Investigation*), que pretendia garantir que seus agentes pudessem ter acesso privilegiado às chaves de criptografia usadas por todas as organizações privadas e indivíduos dos EUA, para propósitos de cumprimento da lei.

O software de criptografia era classificado como munição nos Estados Unidos e estava sujeito a rigorosas restrições de exportação. Outros países, especialmente aliados dos EUA, aplicaram restrições semelhantes ou, em alguns casos, ainda mais rigorosas. O problema era composto pela ignorância geral dentre os políticos, e o público em geral, com relação ao que era software de criptografia e suas aplicações não militares em potencial. As empresas de software norte-americanas protestaram,

	Tamanho da chave/tamanho do hashing (bits)	*PRB otimizado Pentium 1 de 90 Mhz (Mbytes/s)*	*Crypto++ Pentium 4 de 2,1 GHz (Mbytes/s)*
TEA	128	–	23,801
DES	56	0,775	21,340
Triple-DES	112	0,775	9,848
IDEA	128	1,219	18,963
AES	128	–	61,010
AES	192	–	53,145
AES	256	–	48,229
MD5	128	17,025	216,674
SHA-1	160	–	67,977

Figura 7.13 Desempenho dos algoritmos de criptografia simétrica e de resumo seguros.

dizendo que as restrições inibiriam a exportação de software como os navegadores, e as restrições de exportação foram finalmente reformuladas para permitir a exportação de código usando chaves de não mais do que 40 bits – dificilmente uma criptografia forte!

As restrições de exportação podem ter atrapalhado o crescimento do comércio eletrônico, mas não eram particularmente eficazes para impedir a divulgação da experiência em criptografia, nem em manter o software de criptografia fora das mãos de usuários de outros países, pois muitos programadores, dentro e fora dos EUA, estavam ansiosos e eram capazes de implementar e distribuir código de criptografia. A posição atual é que softwares que implementam a maioria dos principais algoritmos de criptografia estão disponíveis há vários anos para o mundo todo, impressos [Schneier 1996], on-line, em versões comerciais e *freeware* [www.rsasecurity.com, cryptography.org, privacy.nb.ca, www.openssl.org].

Um exemplo é o programa chamado PGP (*Pretty Good Privacy*) [Garfinkel 1994, Zimmermann 1995], originalmente desenvolvido por Philip Zimmermann e levado adiante por ele e outros. Isso é parte da campanha de técnicos e políticos para garantir que a disponibilidade de métodos de criptografia não seja controlada pelo governo norte-americano. O PGP foi desenvolvido e distribuído com o objetivo de permitir que todos os usuários de computador usufruíssem do nível de privacidade e integridade proporcionado pelo uso de criptografia de chave pública em suas comunicações. O PGP gera e gerencia chaves públicas e secretas para um usuário. Ele usa criptografia de chave pública RSA para autenticação e para transmitir chaves secretas a um interlocutor, e usa os algoritmos de criptografia de chave secreta IDEA ou 3DES para cifrar mensagens de correio ou outros documentos. (Quando o PGP foi desenvolvido, o uso do algoritmo DES era controlado pelo governo norte-americano.) O PGP está amplamente disponível em versões gratuitas e comerciais. Ele é distribuído por intermédio de sites de distribuição separados para usuários norte-americanos [www.pgp.com] e para os de outras partes do mundo [International PGP], para burlar (de maneira perfeitamente legal) as normas de exportação dos EUA.

Finalmente, o governo norte-americano reconheceu a futilidade da posição da NSA e o dano que estava causando à indústria da computação dos EUA (que não podia comercializar versões seguras de navegadores web, sistemas operacionais distribuídos e muitos outros produtos no mundo todo). Em janeiro de 2000, o governo norte-americano apresentou um novo plano [www.bxa.doc.gov], destinado a permitir que os fornecedores de software dos EUA exportassem software incorporando criptografia forte. As normas correntes em 2004 permitiam a exportação de produtos de software incorporando chaves de criptografia de até 64 bits e chaves públicas de até 1024 bits, usadas para assinatura e troca de chave. Para software exportado com chaves maiores é necessário uma 'revisão' do governo. É claro que os EUA não detêm o monopólio sobre a produção, ou publicação de software de criptografia; estão disponíveis implementações de código-fonte aberto para todos os algoritmos conhecidos [www.cryptopp.com]. O efeito das normas é simplesmente atrapalhar a comercialização de alguns produtos de software comerciais produzidos nos EUA.

Outras iniciativas políticas tiveram como objetivo manter o controle sobre o uso da criptografia, apresentando uma legislação que insistia na inclusão de brechas ou pontos de entradas disponíveis apenas para os órgãos governamentais jurídicos e de segurança. Tais propostas procedem da percepção de que canais de comunicação secretos podem ser muito úteis para criminosos de todos os tipos. Antes do advento da criptografia digital, os governos sempre tinham um meio de interceptar e analisar as comunicações entre os membros do público. A criptografia digital forte altera radicalmente essa situação. Mas essas propostas de legislação para impedir o uso de criptografia forte têm grande resistência por parte dos cidadãos e organismos de liberdades civis, que estão preocupados com seu impacto sobre os direitos à privacidade das pessoas. Até agora, nenhuma dessas propostas legislativas foi adotada, mas os esforços políticos continuam e a eventual introdução de uma estrutura jurídica para o uso de criptografia pode ser inevitável.

7.6 Estudos de caso: Needham–Schroeder, Kerberos, TLS, 802.11 WiFi

Os protocolos de autenticação originalmente publicados por Needham e Schroeder [1978] estão no centro de muitas técnicas de segurança. Nós os detalharemos na Seção 7.6.1. Uma das aplicações mais importantes do protocolo de autenticação de chave secreta deles é o sistema Kerberos [Neuman e Ts'o

1994], que será o assunto de nosso segundo estudo de caso (Seção 7.6.2). O Kerberos foi projetado para fornecer autenticação entre clientes e servidores em redes que formam um único domínio de gerenciamento (*intranets*).

Nosso terceiro estudo de caso (Seção 7.6.3) trata do protocolo TLS (*Transport Layer Security*). Ele foi projetado especificamente para atender à necessidade de transações eletrônicas seguras. Atualmente, ele é suportado pela maioria dos navegadores e servidores web e é empregado em muitas transações comerciais que ocorrem na web.

Nosso estudo de caso final (Seção 7.6.4) ilustra a dificuldade de construir sistemas seguros. O padrão IEEE 802.11 WiFi foi publicado em 1999 com uma especificação de segurança incluída. Mas a análise e ataques subseqüentes mostraram que a especificação era muito inadequada. Identificaremos as deficiências e as relacionaremos aos princípios de criptografia abordados neste capítulo.

7.6.1 O protocolo de autenticação Needham–Schroeder

Os protocolos descritos aqui foram desenvolvidos em resposta à necessidade de um meio seguro de gerenciar chaves (e senhas) em uma rede. Quando o trabalho foi publicado [Needham e Schroeder 1978], os servidores de arquivos estavam acabando de surgir e havia a necessidade urgente de gerenciar de melhor maneira a segurança em redes locais.

Nas redes gerenciadas de forma integrada essa necessidade pode ser satisfeita por um serviço de chaves seguro, que publica chaves de sessão na forma de desafios (veja a Seção 7.2.2). Esse é o objetivo do protocolo de chave secreta desenvolvido por Needham e Schroeder. No mesmo artigo, Needham e Schroeder também relatam um protocolo baseado no uso de chaves públicas para autenticação e distribuição de chaves que não depende da existência de servidores de chaves seguros e que, portanto, é mais conveniente para uso em redes com muitos domínios de gerenciamento independentes, como a Internet. Não vamos descrever a versão de chave pública aqui, mas o protocolo TLS descrito na Seção 7.6.3 é uma variação dela.

Needham e Schroeder propuseram uma solução para a autenticação e distribuição de chaves baseada em um *servidor de autenticação* que fornece chaves secretas para os clientes. A tarefa do servidor de autenticação é proporcionar uma maneira segura para pares de processos obterem chaves compartilhadas. Para fazer isso, ele precisa se comunicar com seus clientes usando mensagens cifradas.

Needham e Schroeder com chaves secretas ◊ Nesse modelo, um processo atuando em nome de um principal A que queira iniciar uma comunicação segura com outro processo atuando em nome de um principal B, pode obter uma chave para tal. O protocolo é descrito para dois processos arbitrários A e B, mas nos sistemas cliente-servidor, A pode ser um cliente iniciando uma seqüência de requisições para um servidor B. Nesse protocolo, o processo A recebe duas informações para serem usadas na interação com o processo B, uma chave de sessão usada para cifrar mensagens para B, e um *tíquete* que ele pode transmitir com segurança para B. (Este último é cifrado com uma chave conhecida por B, mas não por A, para que B possa decifrá-la sem que a chave de sessão seja comprometida durante a transmissão.)

O servidor de autenticação S mantém uma tabela contendo um nome e uma chave secreta para cada principal conhecido pelo sistema. A chave secreta é usada para autenticar processos clientes no servidor de autenticação e para transmitir mensagens com segurança entre processos clientes e o servidor de autenticação. Ela nunca é revelada para terceiros e é transmitida pela rede no máximo uma vez, ao ser gerada. (De preferência, uma chave sempre deve ser transmitida por algum outro meio, como no papel, ou em uma mensagem verbal, evitando qualquer exposição na rede.) Uma chave secreta é equivalente à senha usada para autenticar usuários em sistemas centralizados. Para principais humanos, o nome mantido pelo serviço de autenticação é seu nome de usuário e a chave secreta é sua senha. Ambos são fornecidos pelo usuário para os processos clientes que atuam em seu nome.

O protocolo é baseado na geração e transmissão de tíquetes pelo servidor de autenticação. Um tíquete é uma mensagem cifrada contendo uma chave secreta (de sessão) para uso na comunicação entre A e B. Na Figura 7.14, tabulamos as mensagens do protocolo de chave secreta de Needham e Schroeder. O servidor de autenticação é S.

Cabeçalho	*Mensagem*	*Notas*
1. $A \rightarrow S$:	A, B, N_A	A pede a S para que forneça uma chave para comunicação com B.
2. $S \rightarrow A$:	$\{N_A, B, K_{AB}, \{K_{AB}, A\}K_B\}K_A$	S retorna uma mensagem cifrada com a chave secreta de A, contendo uma chave recentemente gerada K_{AB} e um tíquete cifrado com chave secreta de B. O número usado uma vez (*nonce*) N_A demonstra que a mensagem foi enviada em resposta à precedente. A acredita que S enviou a mensagem, pois somente S conhece a chave secreta de A.
3. $A \rightarrow B$:	$\{K_{AB}, A\}K_B$	A envia o tíquete para B.
4. $B \rightarrow A$:	$\{N_B\}K_{AB}$	B decifra o tíquete e usa a nova chave K_{AB} para cifrar outro número usado uma vez N_B.
5. $A \rightarrow B$:	$\{N_B - 1\}K_{AB}$	A demonstra para B que foi o remetente da mensagem anterior, retornando uma transformação combinada de N_B.

Figura 7.14 O protocolo de chave secreta de autenticação Needham–Schroeder.

N_A e N_B são *números usados uma vez*. Na terminologia de protocolos baseados em desafio-resposta, os números usados uma vez são denominados de *nonce*. Um número usado uma vez é um valor inteiro incluído em uma mensagem para demonstrar que ela é nova. Os números usados uma vez, como seu próprio nome indica, são utilizados uma única vez e são obtidos sob demanda. Por exemplo, os números usados uma vez podem ser gerados como uma seqüência de valores inteiros ou pela leitura do relógio da máquina remetente.

Se o protocolo for concluído com êxito, A e B podem ter certeza de que qualquer mensagem recebida cifrada com K_{AB} é proveniente um do outro, e que qualquer mensagem enviada cifrada com K_{AB} só pode ser entendida pelo outro, ou pelo servidor S (e pressupõe-se que S é confiável). Isso funciona assim porque as únicas mensagens que foram enviadas contendo K_{AB} foram cifradas com a chave secreta de A ou com a chave secreta de B.

Existe uma deficiência nesse protocolo, pois B não tem motivos para acreditar que a mensagem 3 é nova. Um intruso que consiga obter a chave K_{AB} e fazer uma cópia do tíquete e do autenticador $\{K_{AB}, A\}K_B$ (ambos os quais podem ter sido deixados em um local de armazenamento exposto por um programa cliente descuidado, ou defeituoso, executando sob a autorização de A), pode usá-los para iniciar uma troca subseqüente com B, personificando A. Para que esse ataque ocorra, um valor antigo de K_{AB} deve ser comprometido; usando a terminologia atual, Needham e Schroeder não incluíram essa possibilidade em sua lista de ameaças e a opinião de consenso é que alguém deve fazer isso. A deficiência pode ser remediada pela inserção de um número usado uma vez, ou de uma indicação de tempo na mensagem 3, para que ela se torne: $\{K_{AB}, A, t\}K_{Bmsb}$. B decifra essa mensagem e verifica se t é recente. Essa é a solução adotada no Kerberos.

7.6.2 Kerberos

O Kerberos foi desenvolvido no MIT, nos anos 80 [Steiner *et al.* 1988], com o objetivo de fornecer uma variedade de recursos de autenticação e segurança para uso na rede de computadores do campus daquela instituição e em outras intranets. Ele passou por várias revisões e aprimoramentos, à luz da experiência e dos informes de organizações de usuários. O Kerberos versão 5 [Neuman e Ts'o 1994], que descreveremos aqui, está na seqüência dos padrões de Internet (veja o RFC 1510 [Kohl e Neuman 1993]) e agora é usado por muitas empresas e universidades. O código-fonte para uma implementação do Kerberos está disponível no MIT [web.mit.edu I]; ele está incluído no DCE (*Distributed Computing Environment*) do OSF [OSF 1997] e no sistema operacional Microsoft Windows 2000 como serviço de autenticação padrão [www.microsoft.com II]. Foi proposta uma extensão para incorporar o uso de certificados de chave pública para a autenticação inicial de principais (Passo A, na Figura 7.15) [Neuman *et al.* 1999].

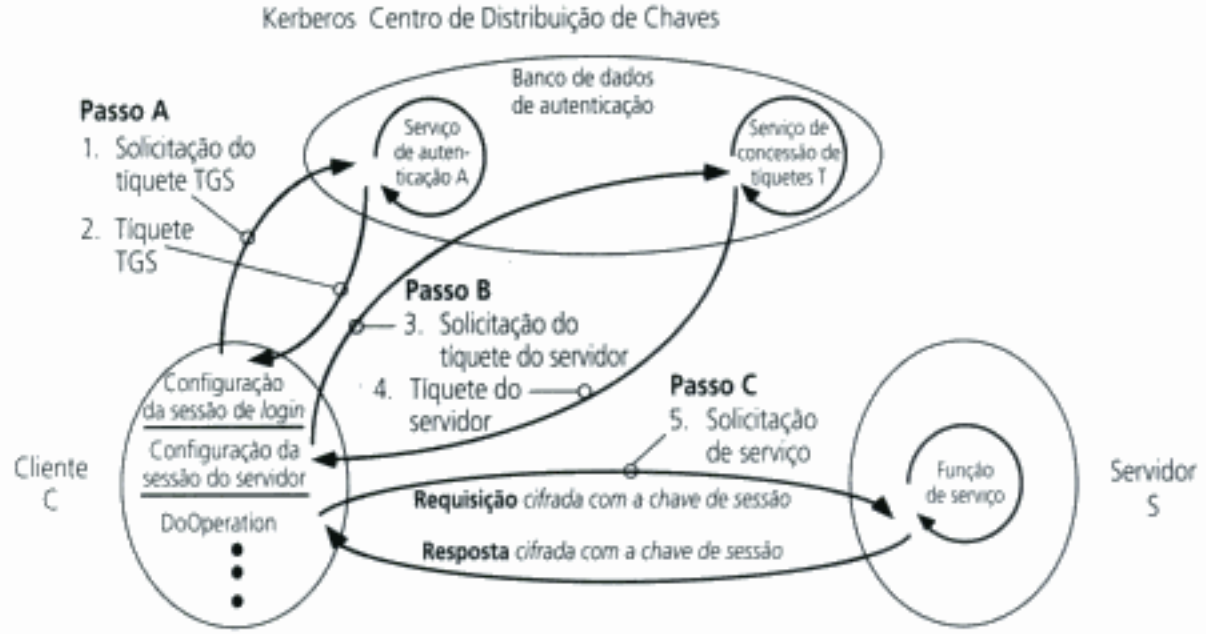


Figura 7.15 Arquitetura do sistema Kerberos.

A Figura 7.15 mostra a arquitetura do processo. O Kerberos trata com três tipos de objeto de segurança:

Tíquete: um *token* emitido para um cliente pelo serviço de concessão de tíquetes do Kerberos para ser apresentado a um servidor específico para verificar se o remetente foi autenticado recentemente pelo Kerberos. Os tíquetes incluem um tempo de expiração e uma chave de sessão gerada para uso do cliente e do servidor.

Autenticação: um *token* construído por um cliente e enviado para um servidor para provar a identidade do usuário e a validade de toda comunicação com esse servidor. Um autenticador só pode ser usado uma vez. Ele contém o nome do cliente e uma indicação de tempo, e é cifrado na chave de sessão apropriada.

Chave de sessão: uma chave secreta gerada aleatoriamente pelo Kerberos e emitida para um cliente para ser usada na comunicação com um servidor em particular. A criptografia não é obrigatória para toda comunicação com servidores; a chave de sessão é usada para cifrar todos os autenticadores e a comunicação com os servidores que exigem isso (veja a seguir).

Os processos clientes devem possuir um tíquete e uma chave de sessão para cada servidor que usarem. Seria impraticável fornecer um novo tíquete e uma chave para cada interação cliente-servidor; portanto a maioria dos tíquetes é concedida aos clientes com um tempo de validade de várias horas, para que eles possam ser usados na interação com um servidor em particular, até que expirem.

Um servidor Kerberos é conhecido como KDC (*Key Distribution Centre* – Centro de Distribuição de Chaves). Cada KDC apresenta um Serviço de Autenticação (*Authentication Service* – AS) e um Serviço de Concessão de Tíquetes (*Ticket-Granting Service* – TGS). No *login*, os usuários são autenticados pelo *AS*, usando uma variação do método de senha segura para a rede, e o processo cliente que está atuando em nome do usuário recebe um *tíquete* e uma chave de sessão para comunicação com o TGS. Subseqüentemente, o processo cliente original e seus descendentes podem usar o tíquete do TGS para obter tíquetes e chaves de sessão para servidores específicos.

O protocolo de Needham e Schroeder é seguido muito de perto no Kerberos, com valores de tempo (inteiros representando uma data e hora) utilizados como números usados uma vez. Isso tem dois objetivos:

- evitar a reprodução de mensagens antigas interceptadas na rede, ou a reutilização de tíquetes antigos encontrados na memória de máquinas das quais o usuário autorizado se desconectou (números usados uma vez foram utilizados para atingir esse objetivo em Needham e Schroeder);
- aplicar um tempo de validade aos tíquetes, permitindo ao sistema revogar direitos dos usuários quando, por exemplo, eles deixam de ser usuários autorizados do sistema.

A seguir, descreveremos os protocolos do Kerberos em detalhes, usando a notação definida abaixo. Primeiramente, descreveremos o protocolo por meio do qual o cliente obtém um tíquete e uma chave de sessão para acesso ao TGS.

Notação:

A	Nome do serviço de autenticação do Kerberos	n	Um número usado uma vez
T	Nome do serviço de concessão de tíquetes Kerberos	t	Uma indicação de tempo
C	Nome do cliente	t_1	Tempo inicial da validade do tíquete
		t_2	Tempo final da validade do tíquete

Um tíquete do Kerberos tem um período de validade fixo, começando no tempo t_1 e terminando no tempo t_2. Um tíquete para um cliente C acessar um servidor S tem a forma:

$\{C, S, t_1, t_2, K_{CS}\}K_S$, que vamos denotar como $\{ticket(C,S)\}K_S$

O nome do cliente é incluído no tíquete para evitar um possível uso por impostores, conforme veremos posteriormente. Os passos e os números de mensagem na Figura 7.15 correspondem àqueles da descrição tabulada A. Note que a mensagem 1 não é cifrada e não inclui a senha de C. Ela contém um número usado uma vez que é utilizado para verificar a validade da resposta.

A mensagem 2 às vezes é chamada de desafio (*challenge*), pois apresenta informações ao solicitante que são úteis apenas se ele conhecer a chave secreta de C, K_C. Um impostor que tente personificar C, enviando a mensagem 1, não poderá ir mais adiante, pois não conseguirá decifrar a mensagem 2. Para principais que são usuários, K_C é uma versão derivada da senha do usuário. O processo cliente pedirá para que o usuário digite sua senha e tentará decifrar a mensagem 2 com ela. Se o usuário fornecer a senha correta, o processo cliente obterá a chave de sessão K_{CT} e um tíquete válido para o serviço de concessão de tíquetes (TGS); caso contrário, ele obterá dados incoerentes. Os servidores têm suas próprias chaves secretas, conhecidas apenas pelo processo servidor relevante e pelo servidor de autenticação.

A. Obtém chave de sessão do Kerberos e tíquete TGS, uma vez por sessão de login

Cabeçalho	*Mensagem*	*Notas*
1. $C \rightarrow A$: Requisição de tíquete TGS	C, T, n	O cliente C pede ao servidor de autenticação Kerberos A para que forneça um tíquete para comunicação com o serviço de concessão de tíquetes T.
2. $A \rightarrow C$: Chave de sessão e tíquete TGS	$\{K_{CT}, n\}K_C$, $\{ticket(C,T)\}K_T$ contendo C, T, t_1, t_2, K_{CT}	A retorna uma mensagem contendo um tíquete cifrado em sua chave secreta e uma chave de sessão para C, para usar com T. A inclusão do número usado uma vez n, cifrado com K_C, mostra que a mensagem vem do destinatário da mensagem 1, que deve conhecer K_C.

Quando um tíquete válido tiver sido obtido do serviço de autenticação, o cliente C poderá usá-lo, até que expire, quantas vezes quiser para se comunicar com o serviço de concessão de tíquetes para obter tíquetes para outros servidores. Assim, para obter um tíquete para um servidor S, C constrói um autenticador cifrado K_{CT}, da forma:

$\{C, t\}K_{CT}$, que vamos denotar como $\{auth(C)\}K_{CT}$, e envia um pedido para T:

B. Obtém tíquete para um servidor S, uma vez por sessão cliente-servidor		
Cabeçalho	*Mensagem*	*Notas*
3. $C \rightarrow T$: Solicita tíquete para o serviço S	$\{auth(C)\}K_{CT}$, $\{ticket(C,T)\}K_T, S, n$	C pede ao servidor de concessão de tíquetes T para que forneça um tíquete para comunicação com outro servidor S.
4. $T \rightarrow C$: Tíquete de serviço	$\{K_{CS}, n\}K_{CT}$, $\{ticket(C,S)\}K_S$	T verifica o tíquete. Se ele for válido, T gerará uma nova chave de sessão aleatória K_{CS} e a retornará com um tíquete para S (cifrado com a chave secreta K_S do servidor).

Então, C está pronto para emitir mensagens de requisição para o servidor S:

C. Emite um pedido de servidor com um tíquete		
Cabeçalho	*Mensagem*	*Notas*
5. $C \rightarrow S$: Requisição de serviço	$\{auth(C)\}K_{CS}$, $\{ticket(C,S)\}K_S$, *request*, n	C envia o tíquete para S, com um autenticador recentemente gerado para C e uma requisição. A requisição pode ser cifrada com K_{CS} se for exigido segredo dos dados.

Para o cliente certificar-se da autenticidade do servidor, S deve retornar para C o número usado uma vez n. (Para reduzir o número de mensagens exigidas, isso poderia ser incluído nas mensagens que contêm a resposta do servidor ao pedido):

D. Autentica o servidor (opcional)		
Cabeçalho	*Mensagem*	*Notas*
6. $S \rightarrow C$: Autenticação do servidor	$\{n\}K_{CS}$	(Opcional): S envia o número usado uma vez para C, cifrado com K_{CS}.

Aplicação do Kerberos ◊ O Kerberos foi desenvolvido para uso no Projeto Athena do MIT – um recurso de computação de rede para os alunos, abrangendo todo o campus, com muitas estações de trabalho e servidores fornecendo serviço para mais de 5.000 usuários. O ambiente é tal que nem a integridade de caráter dos clientes, nem a segurança da rede e das máquinas que oferecem serviços, podem ser presumidas – por exemplo, as estações de trabalho não estão protegidas contra a instalação de um software de sistema desenvolvido pelo usuário e as máquinas servidoras (outras, que não o servidor Kerberos) não são necessariamente seguras contra a interferência física na configuração de seu software.

O Kerberos fornece praticamente toda a segurança do sistema Athena. Ele é usado para autenticar usuários e outros principais. A maioria dos servidores em execução na rede foi estendida para exigir um tíquete de cada cliente no início de cada interação cliente-servidor. Isso inclui sistemas de arquivos (NFS e *Andrew File System*), correio eletrônico, *login* remoto e impressão. As senhas dos usuários são conhecidas apenas por eles próprios e pelo serviço de autenticação do Kerberos. Os serviços têm chaves secretas conhecidas apenas pelo Kerberos e pelos servidores que fornecem o serviço.

Vamos descrever a maneira pela qual o Kerberos é aplicado na autenticação de usuários no momento do *login*. Seu uso para tornar o serviço de arquivos NFS seguro será descrito no Capítulo 8.

***Login* com Kerberos** ◊ Quando um usuário se conecta em uma estação de trabalho, o programa de *login* envia o nome do usuário para o serviço de autenticação do Kerberos. Se o usuário for conhecido

do serviço de autenticação, este responderá com uma chave de sessão e um número usado uma vez, cifrados com base na senha do usuário, e um tíquete para o TGS. Então, o programa de *login* tenta decifrar a chave de sessão e o número usado uma vez, utilizando a senha fornecida pelo usuário em resposta ao *prompt* de senha. Se a senha estiver correta, o programa de *login* obterá a chave de sessão e o número usado uma vez. Ele verificará o número usado uma vez e armazenará a chave de sessão com o tíquete para uso subseqüente, ao se comunicar com o TGS. Nesse ponto, o programa de *login* pode apagar de sua memória a senha do usuário, pois agora o tíquete serve para autenticá-lo. Uma sessão de *login* é então iniciada para o usuário na estação de trabalho. Note que a senha do usuário nunca é exposta à intromissão na rede – ela é mantida na estação de trabalho e é apagada da memória logo depois de ser digitada.

Acessando servidores com o Kerberos ◊ Quando um programa em execução em uma estação de trabalho precisa acessar um serviço, ele solicita um tíquete para o serviço do serviço de concessão de tíquetes (TGS). Por exemplo, quando um usuário do UNIX deseja se conectar em um computador remoto, o programa de comando *rlogin* da estação de trabalho do usuário obtém um tíquete do serviço de concessão de tíquetes do Kerberos para acessar o serviço de rede *rlogind*. O programa de comando *rlogin* envia o tíquete, junto com um novo autenticador, em uma requisição para o processo *rlogind* no computador onde o usuário deseja se conectar. O programa *rlogind* decifra o tíquete com a chave secreta do serviço *rlogin* e verifica a validade do tíquete (isto é, se o tempo de vida do tíquete ainda não expirou). As máquinas servidoras devem tomar o cuidado de armazenar suas chaves secretas de modo que seja inacessível para intrusos.

Então, o programa *rlogind* usa a chave de sessão incluída no tíquete para decifrar o autenticador e verificar se o autenticador é novo (os autenticadores só podem ser usados uma vez). Quando o programa *rlogind* tiver certeza de que o tíquete e o autenticador são válidos, não haverá necessidade de verificar o nome e a senha do usuário, pois a identidade do usuário é conhecida do programa *rlogind* e uma sessão de *login* é estabelecida para esse usuário na máquina remota.

Implementação do Kerberos ◊ O Kerberos é implementado como um servidor executado em uma máquina segura. Um conjunto de bibliotecas é fornecido para ser empregado por aplicativos clientes e serviços. É usado o algoritmo de criptografia DES, mas ele é implementado como um módulo separado que pode ser facilmente substituído.

O serviço Kerberos é flexível – o mundo está dividido em domínios de autoridade de autenticação separados, chamados de *realms*, cada um com seu próprio servidor Kerberos. A maioria dos principais é registrada em apenas um *realm*, mas os servidores de concessão de tíquetes do Kerberos são registrados em todos eles. Os principais podem autenticar a si mesmos em servidores de outros *realms* por meio de seus servidores de concessão de tíquetes locais.

Dentro de um único *realm* podem existir vários servidores de autenticação, todos os quais têm cópias do mesmo banco de dados de autenticação. O banco de dados de autenticação é duplicado por uma técnica simples de mestre–escravo. As atualizações são aplicadas na cópia mestra por um único serviço KDBM (*Kerberos Database Management*), que é executado apenas na máquina mestra. O KDBM trata das solicitações dos usuários para mudar suas senhas e dos pedidos dos administradores de sistema, para adicionar ou excluir principais e alterar suas senhas.

Para tornar esse esquema transparente para os usuários, o tempo de validade dos tíquetes TGS deve ser igual à sessão de *login* mais longa possível, pois o uso de um tíquete expirado resultará na rejeição de requisições de serviço e a única solução é o usuário autenticar a sessão de *login* novamente e depois solicitar novos tíquetes de servidor para todos os serviços em uso. Na prática, são usados tempos de validade na faixa de 12 horas.

Críticas ao Kerberos ◊ O protocolo do Kerberos versão 5, descrito anteriormente, contém vários aprimoramentos projetados para resolver as críticas das versões anteriores [Bellovin e Merritt 1990, Burrows *et al.* 1990]. A crítica mais importante da versão 4 era que os números usados uma vez, utilizados nos autenticadores, eram implementados como indicações de tempo e a proteção contra a reprodução de autenticadores depende pelo menos de uma sincronização aproximada dos relógio dos clientes e dos servidores. Além disso, se um protocolo de sincronização for usado para acertar os relógios dos clientes e dos servidores, ele próprio deverá ser seguro contra ataques. Veja o Capítulo 11 para obter informações sobre os protocolos de sincronização de relógios.

A definição de protocolo da versão 5 permite que os números usados uma vez nos autenticadores sejam implementados como indicações de tempo ou como números de seqüência. Nos dois casos, isso exige que eles sejam exclusivos e, para verificar se eles não foram reproduzidos, que os servidores contenham uma lista dos números usados uma vez recentemente recebidos de cada cliente. Esse é um requisito de implementação inconveniente e, em caso de falhas, difícil de ser garantido pelos servidores. Kehne *et al.* [1992] publicaram um aprimoramento proposto para o protocolo Kerberos, que não conta com relógios sincronizados.

A segurança do Kerberos depende dos tempos de duração das sessões de trabalho – o período de validade dos tíquetes TGS geralmente é limitado a poucas horas; o período deve ser escolhido de forma a ser longo o bastante para evitar interrupções de serviço inconvenientes, mas curto o suficiente para garantir que os usuários que deixaram de ser autenticados, ou que foram recadastrados, não continuem a usar os recursos por mais do que um curto período. Isso pode causar dificuldades em alguns ambientes comerciais, pois pode impor o requisito do usuário fornecer uma nova autenticação em um ponto arbitrário da interação.

7.6.3 Tornando seguras transações eletrônicas com soquetes

O protocolo SSL (*Secure Sockets Layer*) foi desenvolvido originalmente pela Netscape Corporation [Netscape 1996] e, posteriormente, proposto como padrão para atender as necessidade descritas a seguir. Uma versão estendida do SSL foi adotada como padrão da Internet com o nome de protocolo TLS (*Transport Layer Security*), descrito no RFC 2246 [Dierks e Allen 1999]. O TLS é suportado pela maioria dos navegadores e é amplamente usado no comércio da Internet. Suas principais características são:

Criptografia negociável e algoritmos de autenticação ◊ Em uma rede aberta, não devemos supor que todas as partes envolvidas utilizam o mesmo software cliente, ou que todo software cliente e servidor incluam um algoritmo de criptografia em particular. Na verdade, as leis de alguns países tentam restringir o uso de certos algoritmos de criptografia apenas para esses países. O TLS foi projetado de modo que os algoritmos usados para criptografia e autenticação sejam negociados entre os processos nas duas extremidades da conexão, durante um contato inicial (*handshake*). Pode ser que elas não tenham algoritmos suficientes em comum e, nesse caso, a tentativa de conexão falhará.

Comunicação de partida segura ◊ Para atender a necessidade de comunicação segura, sem negociação prévia, nem a ajuda de terceiros, o canal seguro é estabelecido com um protocolo semelhante ao esquema misto descrito anteriormente. É usada uma comunicação não cifrada para as trocas iniciais, depois é usada criptografia de chave pública e, finalmente, troca para criptografia de chave secreta, após uma chave secreta compartilhada tiver sido estabelecida. Cada troca é opcional e precedida de uma negociação.

Assim, o canal seguro pode ser totalmente configurado, permitindo que a comunicação em cada direção seja cifrada e autenticada, mas não impondo isso, para que recursos de computação não precisem ser consumidos em execuções desnecessárias de criptografia.

Os detalhes do protocolo TLS estão publicados e padronizados, e várias bibliotecas de software e *toolkits* estão disponíveis para suportá-lo [Hirsch 1997, www.openssl.org], alguns deles no domínio público. Ele foi incorporado em uma ampla variedade de software aplicativo e sua segurança foi verificada por auditorias independentes.

O TLS consiste em duas camadas (Figura 7.16): uma camada Protocolo de Registro TLS (*TLS Record Protocol*), que implementa um canal seguro, cifrando e autenticando as mensagens transmitidas por qualquer protocolo orientado a conexão; e uma camada de *handshake*, contendo o protocolo de *handshake* TLS e dois outros protocolos relacionados, que estabelecem e mantêm uma sessão TLS (isto é, um canal seguro) entre um cliente e um servidor. Ambas são normalmente implementadas por bibliotecas de software em nível de aplicativo, no cliente e no servidor. O protocolo de registro TLS é uma camada em nível de sessão; e ele pode ser usado para transportar dados em nível de aplicativo de forma transparente entre dois processos, enquanto garante sua privacidade, integridade e autenticidade. Essas são exatamente as propriedades que especificamos para canais seguros em nosso modelo de segurança (Seção 2.3.3), mas no TLS existem opções para os parceiros da comunicação escolherem

Figura 7.16 Pilha de protocolos TLS.
(As Figuras 7.16 a 7.19 são baseadas nos diagramas presentes em Hirsch [1977] e foram usadas com a permissão de Frederick Hirsch.)

ou não se vão implantar decifração e autenticação de mensagens em cada direção. Cada sessão segura recebe um identificador, e cada parceiro pode armazenar identificadores de sessão em uma cache para subseqüente reutilização, evitando a sobrecarga do estabelecimento de uma nova sessão, quando for exigida outra sessão segura com o mesmo parceiro.

O protocolo TLS é amplamente usado para adicionar uma camada de comunicação segura abaixo dos protocolos em nível de aplicativo existentes. A maior utilização do TLS é em interações HTTP seguras como as de comércio eletrônico e em outras aplicações web sensíveis a segurança. Ele é implementado por praticamente todos os navegadores e servidores web – o uso do prefixo de protocolo *https:* nos URLs inicia o estabelecimento de um canal seguro TLS entre um navegador e um servidor web. Ele também tem sido amplamente implantado para fornecer implementações seguras de Telnet, FTP e muitos outros protocolos de aplicativo. O TLS é o padrão *de facto* para uso em aplicações que exigem canais seguros e há uma ampla escolha de implementações disponíveis, tanto comerciais como de domínio público, com APIs para CORBA e Java.

O protocolo de *handshake* TLS está ilustrado na Figura 7.17. O *handshake* é realizado por meio de uma conexão existente. Para estabelecer uma sessão TLS, ele inicia fazendo "em claro" uma troca de mensagens que contêm opções e parâmetros necessários para realizar a criptografia e a autenticação. A seqüência de *handshake* varia, dependendo se é exigida a autenticação de cliente e/ou do servidor. O protocolo de *handshake* também pode ser ativado posteriormente para alterar a especificação de um canal seguro; por exemplo, a comunicação pode começar com autenticação de mensagem usando apenas códigos de autenticação de mensagem e, na seqüência, se desejar que seja adicionado criptografia. Isso é obtido executando-se novamente o protocolo de *handshake* para negociar uma nova especificação de cifras usando o canal existente.

O *handshake* inicial do TLS é potencialmente vulnerável a ataques de homem no meio, conforme descrito na Seção 7.2.2, Cenário 3. Para proteger-se contra eles, a chave pública usada para verificar o primeiro certificado recebido pode ser distribuída por um canal separado – por exemplo, navegadores e outros softwares de Internet distribuídos em CD-ROM podem incluir um conjunto de chaves públicas de algumas autoridades certificadoras reconhecidas. Outra defesa para os clientes de serviços conhecidos é baseada na inclusão do nome de domínio do serviço em seus certificados de chave pública – os clientes só devem contactar o serviço no endereço IP correspondente a esse nome de domínio.

O TLS suporta uma variedade de opções para as funções de criptografia a serem usadas. Coletivamente, elas são conhecidas como *conjunto de cifras*. Um conjunto de cifras inclui uma escolha única para cada um dos recursos mostrados na Figura 7.18.

Uma variedade de conjuntos de cifras populares são previamente armazenadas com identificadores padrão no cliente e no servidor. Durante o *handshake*, o servidor apresenta ao cliente uma lista dos identificadores de conjunto de cifras que tem disponível e o cliente responde selecionando um deles (ou fornecendo uma indicação de erro, caso não tenha nenhum que corresponda). Nesse estágio, eles

Figura 7.17 Protocolo de *handshake* do TLS.

também concordam com um método de compactação (opcional) e com um valor inicial aleatório para as funções de criptografia de bloco CBC (veja a Seção 7.3).

Em seguida, opcionalmente, os parceiros autenticam-se um ao outro, trocando certificados de chave pública assinados no formato X.509. Esses certificados podem ser obtidos com uma autoridade de chave pública, ou podem simplesmente ser gerados temporariamente para isso. De qualquer modo, pelo menos uma chave pública deve estar disponível para uso no próximo estágio do *handshake*.

Um parceiro gera então um *segredo pré-mestre* e o envia para o outro parceiro, cifrado com a chave pública. Um segredo pré-mestre é um valor aleatório (grande) usado pelos dois parceiros para gerar duas chaves de sessão (chamadas de chaves de *escrita*) empregadas para cifrar os dados em cada direção, e os segredos de autenticação de mensagem a serem usados na autenticação da mensagem. Quando tudo isso estiver pronto, uma sessão segura começará. Isso é iniciado pelas mensagens de *Troca de Especificação de Cifra* entre os parceiros. Elas são seguidas de mensagens *Terminada*. Quando as mensagens *Terminada* tiverem sido trocadas, toda comunicação restante será cifrada e assinada de acordo com o conjunto de cifras escolhido, com as chaves combinadas.

A Figura 7.19 mostra o funcionamento do protocolo de registro. Uma mensagem de transmissão é primeiramente fragmentada em blocos de tamanho gerenciável e, em seguida, os blocos são opcionalmente compactados. A compactação não é rigorosamente uma característica da comunicação segura,

Componente	*Descrição*	*Exemplo*
Método de troca de chave	método a ser usado para a troca de uma chave de sessão	RSA com certificados de chave pública
Cifra para transferência de dados	bloco ou cifra de fluxo a ser usado para dados	IDEA
Função de resumo (*digest*) de mensagem	para criar códigos de autenticação de mensagem (MACs)	SHA-1

Figura 7.18 Opções de configuração de *handshake* do protocolo TLS.

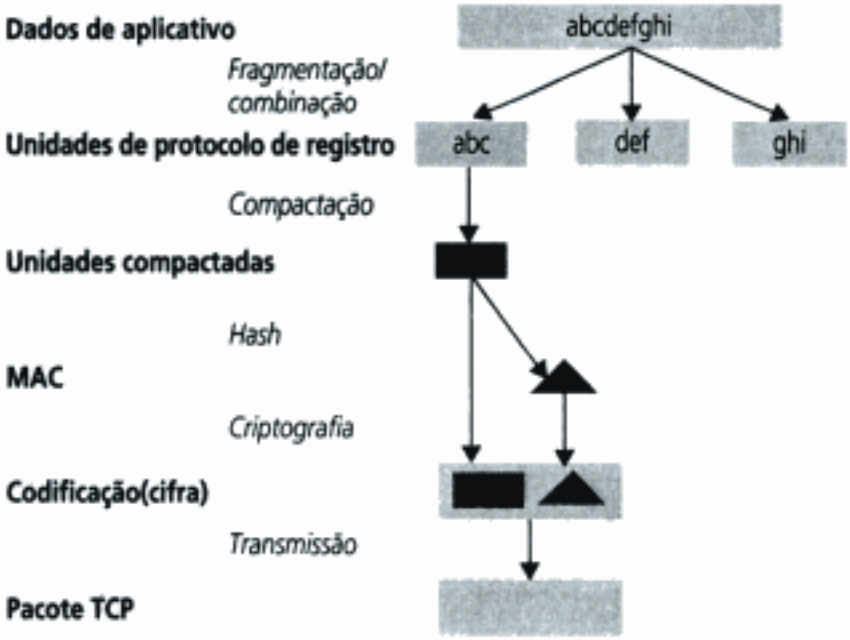


Figura 7.19 Protocolo de registro TLS.

mas ela é fornecida aqui porque um algoritmo de compactação pode auxiliar o trabalho de processamento do grande volume de dados feito pela criptografia e pelos algoritmos de assinatura digital. Em outras palavras, uma seqüência de transformações pode ser feita nos dados para tornar mais eficiente as operações realizadas posteriormente.

A criptografia e as transformações de autenticação de mensagem (MAC) implantam os algoritmos especificados no conjunto de cifras combinado, exatamente como descrito nas Seções 7.3.1 e 7.4.2. Finalmente, o bloco assinado e cifrado é transmitido para o seu interlocutor, por meio da conexão TCP associada, onde as transformações são revertidas para produzir o bloco de dados original.

Resumo ◊ O protocolo TLS fornece uma implementação prática de um esquema de criptografia misto com autenticação e troca de chave baseadas em chaves públicas. Como as cifras são negociadas no *handshake*, elas não dependem da disponibilidade de nenhum algoritmo em particular e também não dependem de quaisquer serviços seguros no momento do estabelecimento da sessão. O único requisito é que os certificados de chave pública sejam emitidos por uma autoridade reconhecida pelas duas partes.

Como a base SSL do TLS e sua implementação de referência foram publicadas [Netscape 1996], elas foram assunto de muita análise e debate. Algumas emendas foram feitas nos primeiros projetos e ele foi aprovado como um padrão válido. Atualmente, o TLS está integrado na maioria dos navegadores e servidores web, e é usado em outras aplicações, como Telnet e FTP seguros. Implementações comerciais e de domínio público [www.rsasecurity.com, Hirsch 1997, www.openssl.org] estão disponíveis, na forma de bibliotecas e *plug-ins* de navegador.

7.6.4 Deficiências no projeto de segurança do IEEE 802.11 WiFi

O padrão IEEE 802.11 para redes locais sem fio, descrito na Seção 3.5.2, foi lançado pela primeira vez em 1999 [IEEE 1999]. Ele foi implementado em estações de base, *laptops* e equipamentos portáteis e é largamente usado em comunicação móvel. Infelizmente, descobriu-se que o projeto de segurança do padrão era inadequado sob vários aspectos. Vamos esboçar esse projeto inicial e as suas deficiências como um estudo de caso das dificuldades do projeto de segurança já referido na Seção 7.1.3.

Foi reconhecido que as redes sem fio são, por sua natureza, mais expostas a ataques do que as redes cabeadas, pois a rede e os dados transmitidos são sensíveis a intromissão e ao mascaramento por parte de qualquer dispositivo equipado com um transmissor/receptor dentro de um raio de cobertura. Portanto, o projeto 802.11 inicial tinha como objetivo fornecer controle de acesso para redes WiFi, privacidade e integridade dos dados nelas transmitidos, através de uma especificação de segurança

intitulada WEP (*Wired Equivalent Privacy*), que incorporava as seguintes medidas, todas configuradas opcionalmente por um administrador de rede:

Controle de acesso por um protocolo de desafio-resposta (cf. Kerberos, Seção 7.6.2), no qual um nó que esteja se juntando a rede é questionado pela estação de base para demonstrar se tem a chave compartilhada correta. Uma única chave K, designada por um administrador de rede, é compartilhada entre a estação de base e todos os dispositivos autorizados.

Privacidade e integridade usando um mecanismo de criptografia opcional baseado na cifra de fluxo RC4. A mesma chave K, empregada no controle de acesso, também é usada na criptografia. Existem opções de comprimento de chave de 40, 64 ou 128 bits. Uma soma de verificação cifrada é incluída em cada pacote para proteger sua integridade.

As seguintes deficiências e fraquezas de projeto foram descobertas logo depois que o padrão foi distribuído:

O compartilhamento de uma única chave por todos os usuários de uma rede torna o projeto fraco na prática, pois:

- a chave provavelmente será transmitida para novos usuários em canais desprotegidos.
- um único usuário descuidado, ou mal-intencionado (como um antigo funcionário descontente), que tenha obtido acesso à chave, pode comprometer a segurança da rede inteira e isso pode passar despercebido.

Solução: um protocolo baseado em chave pública para negociar chaves individuais, como é feito no TLS/SSL (veja a Seção 7.6.3).

As estações de base nunca são autenticadas; portanto, um invasor que conheça a chave compartilhada corrente poderia introduzir uma estação de base de *spoofing* e invadir, inserir, ou falsificar qualquer tráfego.

Solução: as estações de base devem fornecer um certificado que possa ser autenticado pelo uso de uma chave pública obtida de terceiros.

Uso inadequado de uma cifra de fluxo, em vez de uma cifra de bloco (veja descrições das cifras de bloco e de fluxo na Seção 7.3). A Figura 7.20 mostra o processo de criptografia e decifração na segurança do 802.11 WEP. Cada pacote é cifrado pela operação XOR de seu conteúdo com um fluxo de chave produzido pelo algoritmo RC4. A estação receptora usa RC4 para gerar o mesmo fluxo de chave e decifrar o pacote com outra operação XOR. Para evitar erros de sincronização quando pacotes são perdidos ou corrompidos, o RC4 é reiniciado com um valor de partida composto por um *valor inicial* de 24 bits concatenado com a chave compartilhada globalmente. O valor inicial é atualizado e incluído (abertamente) em cada pacote transmitido. A chave compartilhada não pode ser alterada facilmente no uso normal; portanto, o valor de partida tem apenas $s = 2^{24}$ (ou cerca de 10^7) estados diferentes, resultando na repetição do valor inicial após o envio de 10^7 pacotes. Na prática isso pode ocorrer dentro de poucas horas e até ciclos de repetição mais curtos podem surgir, caso pacotes sejam perdidos. Um invasor que receba os pacotes cifrados sempre pode detectar repetições, pois o valor inicial é enviado abertamente.

A especificação RC4 alerta explicitamente contra a repetição de fluxo de chaves. Isso porque um invasor que receba um pacote cifrado C_i e conheça o texto puro P_i (por exemplo, supondo que se trata de uma pergunta padrão para um servidor) pode calcular o fluxo de chaves K_i usado para cifrar o pacote. Mas o mesmo valor de K_i tornará a acontecer depois que s pacotes forem transmitidos, e o invasor poderá usá-lo para decifrar o pacote recentemente transmitido. Dessa maneira, o invasor poderá eventualmente ter êxito na decifração de uma grande proporção dos pacotes, adivinhando os pacotes de texto puro corretamente. Essa deficiência foi apontada pela primeira vez por Borisov *et al.* [2001] e levou a uma reavaliação importante da segurança do WEP e sua substituição nas versões posteriores do padrão 802.11.

Solução: negociar uma nova chave após um tempo menor do que o pior caso de repetição. Seria necessário um código de finalização explícito, como acontece no TLS.

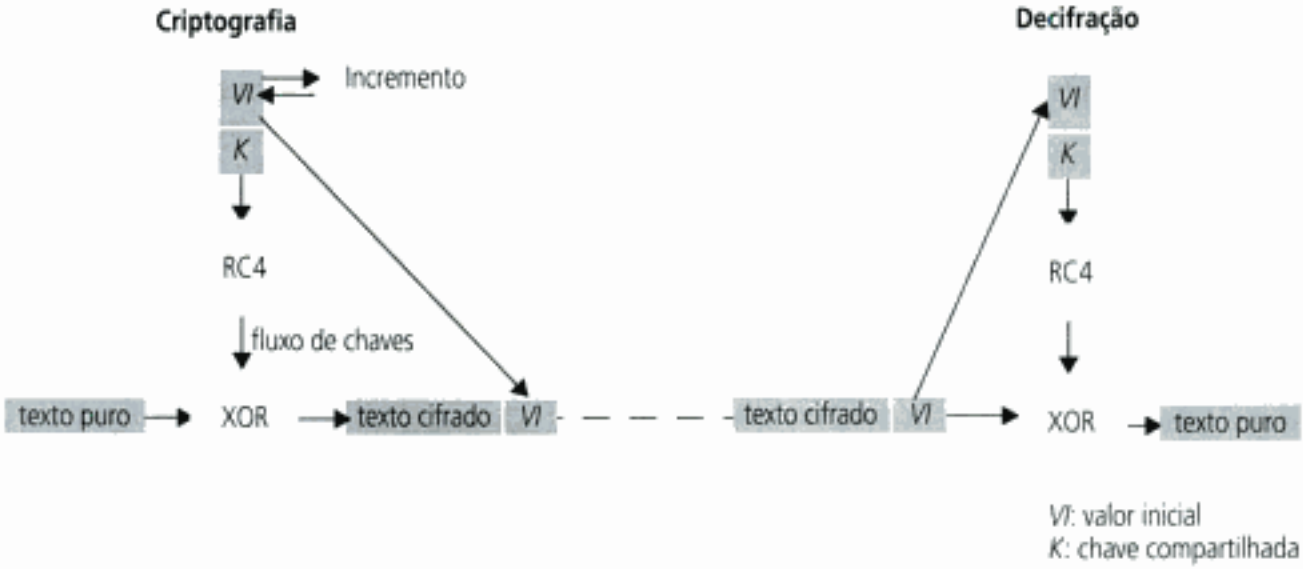


Figura 7.20 Uso de cifra de fluxo RC4 no IEEE 802.11 WEP.

Comprimentos de chave de 40 e 64 bits foram incluídos no padrão para permitir que os produtos fossem distribuídos no exterior por fornecedores norte-americanos, em uma época em que as normas do governo dos EUA, referidas na Seção 7.5.2, restringiam a 40 bits (e, subseqüentemente, 64 bits) os comprimentos de chave para equipamentos exportados. Mas as chaves de 40 bits são tão facilmente violadas pela força bruta que oferecem muito pouca segurança, e mesmo chaves de 64 bits são potencialmente violáveis com um ataque determinado.

Solução: usar somente chaves de 128 bits. Isso foi adotado em muitos produtos WiFi recentes.

A cifra de fluxo RC4 revelou, após a publicação do padrão 802.11, ter deficiências que permitiam o descobrimento da chave após observação de um volume substancial do tráfego, mesmo sem repetição do fluxo de chaves [Fluhrer *et al.* 2001]. Essa deficiência foi demonstrada na prática. Ela tornou o esquema WEP inseguro, mesmo com chaves de 128 bits, e levou algumas empresas a proibir o uso de redes WiFi por seus funcionários.

Solução: preparativos para a negociação de especificações de cifra, como foi feito no TLS, fornecendo uma escolha de algoritmos de criptografia. O RC4 é incorporado no padrão WEP, sem nenhum preparativo para a negociação de algoritmos de criptografia.

Freqüentemente, os usuários não implantavam a proteção oferecida pelo esquema WEP, provavelmente porque não percebiam claramente como seus dados estavam expostos. Essa não era uma deficiência no projeto do padrão, mas na comercialização dos produtos nele baseados. A maioria era projetada para começar com a segurança desativada e, freqüentemente, a documentação sobre os riscos para a segurança era deficiente.

Solução: melhores configurações padrão e documentação. Mas os usuários estavam preocupados em obter desempenho ótimo e, com o hardware disponível na época, a comunicação era perceptivelmente mais lenta com a criptografia ativada. Tentativas de evitar o uso de criptografia WEP levaram ao acréscimo de recursos nas estações de base para a supressão dos pacotes de identificação normalmente divulgados por elas e à rejeição de pacotes não enviados a partir de um endereço MAC autorizado (veja a Seção 3.5.1). Nenhum deles oferecia muita segurança, pois uma rede pode ser descoberta pela interceptação de pacotes (*sniffing*) na transmissão e os endereços MAC estão sujeitos a *spoofing* por meio de modificações no sistema operacional.

O IEEE 802.11i é um padrão de segurança 802.11 lançado em 2004, que trata de todas as deficiências acima, com opções para autenticação mútua, chaves com reconhecimento de pares negociadas dinamicamente e uso de criptografia AES [IEEE 2004b, Edney e Arbaugh 2003].

7.7 Resumo

As ameaças à segurança dos sistemas distribuídos são generalizadas. É fundamental proteger os canais de comunicação e as interfaces de qualquer sistema que manipule informações que possam ser objeto de ataques. Correspondência pessoal, comércio eletrônico e outras transações financeiras, todos são exemplos de tais informações. Os protocolos de segurança são cuidadosamente projetados para evitar brechas. O projeto de sistemas seguros começa a partir de uma lista de ameaças e um conjunto de suposições de "pior caso".

Os mecanismos de segurança são baseados em criptografia de chave pública e de chave secreta. Os algoritmos de criptografia misturam as mensagens de uma maneira que não possam ser revertidas sem o conhecimento da chave de decifração. A criptografia de chave secreta é simétrica – a mesma chave serve tanto para cifrar como para decifrar. Se duas pessoas compartilham uma chave secreta, elas podem trocar informações cifradas sem risco de intromissão, ou falsificação, e com garantia de autenticidade.

A criptografia de chave pública é assimétrica – chaves distintas são usadas para cifrar e decifrar, e o conhecimento de uma não revela a outra. Uma chave é publicada, permitindo que qualquer um envie mensagens seguras para o portador da chave privada correspondente. Também permite que o portador da chave privada assine mensagens e certificados. Os certificados podem atuar como credenciais para a utilização de recursos protegidos.

Os recursos são protegidos por mecanismos de controle de acesso. Os esquemas de controle de acesso atribuem direitos aos principais (isto é, os portadores de credenciais) para efetuarem operações nos objetos distribuídos e em coleções de objetos. Os direitos podem ser mantidos em listas de controle de acesso (ACLs) associadas a conjuntos de objetos ou podem ser mantidos pelos principais, na forma de capacidades – chaves impossíveis de falsificar, para acessar conjuntos de recursos. As capacidades são convenientes para a delegação de direitos de acesso, mas são difíceis de revogar. As alterações nas ACLs surtem efeito imediatamente, revogando os direitos de acesso anteriores, mas elas são mais complexas e dispendiosas de gerenciar do que as capacidades.

Até recentemente, o algoritmo de criptografia DES era o esquema de criptografia simétrica mais difundido, mas suas chaves de 56 bits não são mais seguras contra ataques de força bruta. A versão *triple* DES fornece o poder de chaves de 112 bits, que são seguras, mas outros algoritmos modernos, como IDEA e AES, são muito mais rápidos e poderosos.

O RSA é o esquema de criptografia assimétrica mais usado. Para obter segurança contra ataques de decomposição em fatores, ele deve ser usado com chaves de 768 bits, ou mais. A execução dos algoritmos de chave pública (assimétricos) é superada pela dos algoritmos de chave secreta (simétricos) em várias ordens de grandeza; portanto, geralmente eles são usados apenas em protocolos mistos, como o TLS, para o estabelecimento de canais seguros que utilizam chaves compartilhadas para trocas subseqüentes.

O protocolo de autenticação Needham–Schroeder foi o primeiro protocolo de segurança prático de propósito geral e ainda serve de base para muitos sistemas. O Kerberos é um esquema bem projetado para a autenticação de usuários e proteção de serviços dentro de uma única organização. O Kerberos é baseado no protocolo Needham–Schroeder e em criptografia simétrica. O TLS é o protocolo de segurança projetado para comércio eletrônico e é amplamente usado para tal. Trata-se de um protocolo flexível para o estabelecimento e uso de canais seguros, baseado em criptografia simétrica e assimétrica. As deficiências da segurança do IEEE 802.11 WiFi proporcionam uma demonstração prática das dificuldades do projeto de segurança.

Exercícios

7.1 Descreva algumas das políticas de segurança física de sua organização. Expresse-as em termos que poderiam ser implementados em um sistema de travamento de porta computadorizado. *página 237*

7.2 Descreva algumas das maneiras pelas quais o e-mail convencional é vulnerável à intromissão, mascaramento, falsificação, reprodução e negação de serviço. Sugira métodos pelos quais o e-mail poderia ser protegido contra cada uma dessas formas de ataque. *página 238–239*

7.3 As trocas iniciais de chaves públicas são vulneráveis ao ataque do homem no meio. Descreva quantas defesas contra ele você puder imaginar. *páginas 244, 277*

7.4 O PGP é freqüentemente usado para comunicação segura por e-mail. Descreva os passos que dois usuários usando PGP devem efetuar, antes que possam trocar mensagens por e-mail com garantias de privacidade e autenticidade. Qual escopo existe para tornar a negociação de chaves preliminar invisível para os usuários? (A negociação PGP é um caso do esquema misto.) *páginas 260, 269*

7.5 Como o e-mail poderia ser enviado para uma lista grande de destinatários, usando PGP ou um esquema similar? Sugira um esquema que seja mais simples e mais rápido, quando a lista for usada freqüentemente. *página 268–269, Seção 4.5*

7.6 A implementação do algoritmo de criptografia simétrica TEA, apresentado nas Figuras 7.7– 7.9, não pode ser portado entre todas as arquiteturas de máquina. Explique por quê. Como uma mensagem cifrada usando a implementação TEA poderia ser transmitida de forma a ser decifrada corretamente em todas as outras arquiteturas? *página 256–257*

7.7 Modifique o programa aplicativo TEA da Figura 7.10 para usar encadeamento de blocos de cifra (CBC). *páginas 254, 257*

7.8 Construa um aplicativo de cifra de fluxo baseado no programa da Figura 7.9. *páginas 255, 256–257*

7.9 Faça uma estimativa do tempo exigido para violar uma chave DES de 56 bits, por um ataque de força bruta usando um computador de 2000 MIPS (milhões de instruções por segundo), supondo que o laço interno de um programa de ataque de força bruta envolva cerca de 10 instruções por valor de chave, mais o tempo para cifrar um texto puro de 8 bytes (veja a Figura 7.13). Efetue o mesmo cálculo para uma chave IDEA de 128 bits. Extrapole seus cálculos para obter o tempo de violação para um processador paralelo de 200.000 MIPS (ou um consórcio na Internet com poder de processamento similar). *página 257–258*

7.10 No protocolo de autenticação de Needham e Shroeder com chaves secretas, explique por que a seguinte versão da mensagem 5 não é segura:

$$A \rightarrow B: \quad [N_B]K_{AB}$$

página 270–271

7.11 Examine as soluções propostas na discussão sobre o projeto do protocolo 802.11 *Wireless Equivalent Privacy*, esboçando maneiras pelas quais cada solução poderia ser implementada e mencionando as desvantagens ou os inconvenientes. (5 respostas) *página 279–280*

8 Sistemas de Arquivos Distribuídos

Um sistema de arquivos distribuído permite aos programas armazenarem e acessarem arquivos remotos exatamente como se fossem locais, possibilitando que os usuários acessem arquivos a partir de qualquer computador em uma rede. O desempenho e a segurança no acesso aos arquivos armazenados em um servidor devem ser comparáveis aos arquivos armazenados em discos locais.

Neste capítulo, definiremos uma arquitetura simples para sistemas de arquivos e descreveremos duas implementações básicas de serviços de arquivos distribuídos com projetos contrastantes que estão em pleno uso há duas décadas:

- o Network File System, NFS, da Suna
- o Andrew File System, AFS

Cada um deles simula a interface de sistema de arquivos do UNIX com diferentes graus de escalabilidade, tolerância a falhas e variações da restrita semântica de cópia única (*one-copy*) do UNIX.

Também serão examinados outros sistemas de arquivos que exploram novos modos de organização de dados no disco, ou entre vários servidores, para obter sistemas de arquivos de alto desempenho, tolerantes a falhas e com capacidade de mudar de escala. Diferentes tipos de sistemas de armazenamento serão descritos em outras partes do livro. Isso inclui os sistemas de armazenamento *peer-to-peer* (Capítulo 10), os sistemas de arquivos replicados (Capítulo 15) e os servidores de dados multimídia (Capítulo 17).

8.1 Introdução

Nos Capítulos 1 e 2, identificamos o compartilhamento de recursos como um objetivo chave dos sistemas distribuídos. O compartilhamento de informações armazenadas talvez seja o aspecto mais importante dos recursos distribuídos. Os mecanismos para compartilhamento de dados assumem muitas formas e serão descritos em várias partes deste livro. Os servidores web fornecem uma forma restrita de compartilhamento de dados, na qual os arquivos armazenados de forma local no servidor estão disponíveis para clientes em toda a Internet, mas os dados acessados por meio de servidores web são gerenciados e atualizados em sistemas de arquivos no servidor, ou distribuídos em uma rede local. O projeto de sistemas de armazenamento de arquivos remotos para leitura e escrita em larga escala apresenta problemas de balanceamento de carga, confiabilidade, disponibilidade e segurança, cuja solução é o objetivo dos sistemas de armazenamento de arquivos *peer-to-peer*, descritos no Capítulo 10. O Capítulo 15 focaliza os sistemas de armazenamento replicados, que são convenientes para aplicações que exigem acesso confiável a dados armazenados em sistemas onde a disponibilidade de *hosts* individuais não pode ser garantida. No Capítulo 17, descreveremos um servidor de mídia projetado para servir fluxos de dados de vídeo em tempo real para grandes números de usuários.

Os requisitos de compartilhamento dentro de redes locais e intranets levam à necessidade de um tipo de serviço diferente – que suporte o armazenamento persistente dos dados e programas de todos os tipos e a distribuição consistente de dados atualizados. O objetivo deste capítulo é descrever a arquitetura e a implementação desses sistemas de arquivos distribuídos *básicos*. Usamos a palavra "básico" aqui para denotarmos os sistemas de arquivos distribuídos cujo principal objetivo é simular a funcionalidade de um sistema de arquivos não-distribuído para programas clientes executados em computadores remotos. Eles não mantêm várias réplicas persistentes dos arquivos, nem suportam as garantias de largura de banda e temporização exigidas para fluxos de dados multimídia – esses requisitos serão tratados em capítulos posteriores. Os sistemas de arquivos distribuídos básicos fornecem um apoio fundamental para computação organizacional baseada em intranets.

Os sistemas de arquivos foram originalmente desenvolvidos para sistemas de computadores centralizados e computadores *desktop*, como um recurso do sistema operacional que fornece uma interface de programação conveniente para armazenamento em disco. Subseqüentemente, eles adquiriram características como controle de acesso e mecanismos de proteção de arquivos, que os tornaram úteis para o compartilhamento de dados e programas. Os sistemas de arquivos distribuídos suportam o compartilhamento de informações através de arquivos e recursos de hardware com armazenamento persistente em toda uma intranet. Um serviço de arquivo bem projetado dá acesso a arquivos armazenados em um servidor com desempenho e confiabilidade semelhantes (e, em alguns casos, melhores) aos arquivos armazenados em discos locais. Seu projeto é adaptado para as características de desempenho e confiabilidade das redes locais e, portanto, eles são mais eficazes no fornecimento de armazenamento persistente compartilhado para uso em intranets. Os primeiros servidores de arquivos foram desenvolvidos por pesquisadores nos anos 70 [Birrell e Needham 1980, Mitchell e Dion 1982, Leach *et al.* 1983] e o Sun Network File System se tornou disponível no início dos anos 80 [Sandberg *et al.* 1985, Callaghan 1999].

Um serviço de arquivos permite que os programas armazenem e acessem arquivos remotos exatamente como se fossem locais, possibilitando que os usuários acessem seus arquivos a partir de qualquer computador em uma intranet. A concentração do armazenamento persistente em alguns poucos servidores reduz a necessidade de armazenamento em disco local e (o mais importante) permite que sejam feitas economias no gerenciamento e no arquivamento (*backups*) dos dados persistentes pertencentes a uma organização. Outros serviços, como o serviço de nomes, o serviço de autenticação de usuário e o serviço de impressão, podem ser mais facilmente implementados quando são capazes de chamar o serviço de arquivos para atender suas necessidades de armazenamento persistente. Os servidores web confiam no sistema de arquivos para o armazenamento das páginas que servem. Nas organizações que operam servidores web para acesso externo e interno por meio de uma intranet, esses servidores freqüentemente armazenam e acessam o material a partir de um sistema de arquivos distribuído local.

Com o advento da programação orientada a objetos distribuída surgiu a necessidade do armazenamento persistente e da distribuição de objetos compartilhados. Uma maneira de obter isso é dispor de objetos serializados (da maneira descrita na Seção 4.3.2) e armazenar e recuperar esses objetos

usando arquivos. Mas esse método de obtenção de persistência e distribuição se torna impraticável para objetos que mudam rapidamente e, portanto, várias estratégias mais diretas foram desenvolvidas. A invocação a objeto remoto da linguagem Java e os ORBs do CORBA fornecem acesso a objetos remotos compartilhados, mas nenhum deles garante a persistência dos objetos, nem os objetos distribuídos são replicados.

A Figura 8.1 apresenta um panorama dos tipos de sistema de armazenamento. Além daqueles já mencionados, a tabela inclui sistemas de memória compartilhada distribuída (DSM – *Distributed Shared Memory*) e armazenamento de objetos persistentes. A DSM será descrita em detalhes no Capítulo 18. Ela fornece uma simulação de uma memória compartilhada por meio da replicação de páginas ou de segmentos de memória em cada *host*. Ela não fornece necessariamente persistência automática. O armazenamento de objetos persistentes foi apresentado no Capítulo 5. Seu objetivo é fornecer persistência para objetos compartilhados distribuídos. Exemplos incluem o Persistent State Service CORBA (veja o Capítulo 20) e extensões persistentes para Java [Jordan 1996, java.sun.com VIII]. Alguns projetos de pesquisa foram desenvolvidos em plataformas que suportam a replicação automática e o armazenamento persistente de objetos (por exemplo, PerDiS [Ferreira *et al.* 2000] e Khazana [Carter *et al.* 1998]). Os sistemas de armazenamento *peer-to-peer* oferecem escalabilidade para suportar cargas de cliente muito maiores do que os sistemas descritos neste capítulo, mas eles acarretam altos custos de desempenho para fornecer controle de acesso seguro e consistência entre as réplicas que podem ser atualizadas.

A coluna *consistência* indica se existem mecanismos para a manutenção da consistência entre várias cópias dos dados quando ocorrem atualizações. Praticamente todos os sistemas de armazenamento contam com o uso de cache para otimizar o desempenho dos programas. O uso de cache foi aplicado pela primeira vez na memória principal e em sistemas de arquivos não-distribuídos, e para eles, a consistência é estrita (denotado por 1, significando consistência de cópia única, na Figura 8.1) – após uma atualização, os programas não conseguem observar quaisquer discrepâncias entre as cópias na cache e os dados armazenados. Quando são usadas réplicas distribuídas, é mais difícil de obter a consistência estrita. Os sistemas de arquivos distribuídos, como o Sun NFS e o Andrew File System, colocam em cache, nos computadores clientes, cópias parciais de arquivos e adotam mecanismos de

Figura 8.1 Sistemas de armazenamento e suas propriedades

	Compartilhamento	*Persistência*	*Cache/ réplicas distribuídas*	*Manutenção da consistência*	*Exemplo*
Memória principal	×	×	×	1	RAM
Sistema de arquivos	×	✔	×	1	Sistema de arquivos UNIX
Sistema de arquivos distribuído	✔	✔	✔	✔	Sun NFS
Web	✔	✔	✔	×	Servidor web
Memória compartilhada distribuída	✔	×	✔	✔	Ivy (DSM, Cap. 18)
Objetos remotos (RMI/ORB)	✔	×	×	1	CORBA
Armazenamento de objetos persistentes	✔	✔	×	1	Persistent State Service do CORBA
Sistema de armazenamento *peer-to-peer*	✔	✔	✔	2	OceanStore (Cap. 10)

Tipos de consistência:
1: Uma cópia restrita. ✔: Garantias ligeiramente mais fracas. 2: garantias consideravelmente mais fracas.

consistência específicos para manter uma aproximação da consistência estrita. Isso está indicado por um tique (✔) na coluna da consistência na Figura 8.1 – discutiremos esses mecanismos e até que ponto eles se desviam da consistência estrita, nas Seções 8.3 e 8.4.

A web usa cache extensivamente, tanto em computadores clientes como em servidores *proxies* mantidos por organizações de usuários. A consistência entre o original e as cópias armazenadas em servidores *proxies* web e em caches de clientes é mantida apenas por ações explícitas do usuário. Os clientes não são notificados quando uma página original, armazenada em um servidor, é atualizada; eles precisam fazer verificações explícitas para manter suas cópias locais atualizadas. Isso tem como objetivo a navegação adequada na web, mas não suporta o desenvolvimento de aplicativos cooperativos, como o do quadro branco compartilhado distribuído. Os mecanismos de consistência usados nos sistemas DSM serão discutidos em detalhes no Capítulo 18. Os sistemas de objetos persistentes variam consideravelmente em sua estratégia para uso de cache e consistência. Os esquemas CORBA e Persistent Java mantêm cópias únicas de objetos persistentes e é necessário usar invocação remota para acessá-los; portanto, o único problema de consistência se dá entre a cópia persistente de um objeto no disco e a cópia ativa na memória, que não é visível para clientes remotos. Os projetos PerDiS e Khazana, mencionados anteriormente, mantêm réplicas dos objetos em cache e empregam mecanismos de consistência bastante elaborados para reproduzir àquelas encontradas nos sistemas DSM.

Apresentamos aqui alguns problemas mais amplos relacionados ao armazenamento e à distribuição de dados persistentes e não persistentes, agora vamos voltar ao assunto principal deste capítulo – o projeto de sistemas de arquivos distribuídos básicos. Descreveremos algumas características relevantes dos sistemas de arquivos (não-distribuídos), na Seção 8.1.1, e os requisitos dos sistemas de arquivos distribuídos, na Seção 8.1.2. A Seção 8.1.3 apresentará os estudos de caso que serão utilizados por todo o capítulo. Na Seção 8.2, definiremos um modelo abstrato de um sistema de arquivos distribuído básico, incluindo um conjunto de interfaces de programação. O sistema Sun NFS será descrito na Seção 8.3; ele compartilha muitas das características do modelo abstrato. Na Seção 8.4, descreveremos o Andrew File System – um sistema amplamente usado, que emprega mecanismos de cache e de consistência substancialmente diferentes. A Seção 8.5 examina alguns desenvolvimentos recentes no projeto de serviços de arquivo.

Os sistemas descritos neste capítulo não cobrem todo o espectro dos sistemas de gerenciamento de arquivos e dados distribuídos. Vários sistemas, com características mais avançadas, serão descritos posteriormente neste livro. O Capítulo 15 contém uma descrição do Coda, um sistema de arquivos distribuído que mantém réplicas persistentes dos arquivos para obter confiabilidade, disponibilidade e trabalho em modo não-conectado. O Bayou, um sistema de gerenciamento de dados distribuído que fornece uma forma de replicação com consistência fraca para obter alta disponibilidade, também será abordado no Capítulo 15. O Capítulo 17 abordará o servidor de arquivos de vídeo Tiger, projetado para fornecer a distribuição de fluxos de dados para grandes números de clientes.

8.1.1 Características dos sistemas de arquivos

Os sistemas de arquivos são responsáveis pela organização, armazenamento, recuperação, atribuição de nomes, compartilhamento e proteção de arquivos. Eles fornecem uma interface de programação que caracteriza a abstração de arquivo, liberando os programadores da preocupação com os detalhes da alocação e do leiaute do armazenamento físico no disco. Os arquivos são armazenados em discos ou em outra mídia de armazenamento não-volátil.

Os arquivos contêm *dados* e *atributos*. Os dados consistem em uma seqüência de elementos (normalmente, bytes de 8 bits), acessíveis pelas operações de leitura e escrita de qualquer parte dessa seqüência. Os atributos são mantidos como um único registro, contendo informações como o tamanho do arquivo, indicações de tempo, tipo de arquivo, identidade do proprietário e listas de controle de acesso. Uma estrutura de registro de atributo típica está ilustrada na Figura 8.3. Os atributos escondidos (*shadow attributes*) são gerenciados pelo sistema de arquivos e normalmente não podem ser atualizados por programas de usuário.

Os sistemas de arquivos são projetados para armazenar e gerenciar um grande número de arquivos, com recursos para criação, atribuição de nomes e exclusão de arquivos. A atribuição de nomes de arquivos é suportada pelo uso de diretórios. Um *diretório* é um arquivo, freqüentemente de um tipo especial, que fornece um mapeamento dos nomes textuais para identificadores internos de arquivo.

Módulo de diretório:	relaciona nomes de arquivo com IDs de arquivo
Módulo de arquivo:	relaciona IDs de arquivo com arquivos em particular
Módulo de controle de acesso:	verifica a permissão para a operação solicitada
Módulo de acesso a arquivo:	lê ou escreve dados, ou atributos
Módulo de bloco:	acessa e aloca blocos de disco
Módulo de dispositivo:	E/S de disco e uso de *buffers*

Figura 8.2 Módulos do sistema de arquivos.

Os diretórios podem incluir nomes de outros diretórios, levando ao conhecido esquema de atribuição hierárquica de nomes e aos *nomes de caminho* (*pathname*), usados nos sistemas de arquivos do UNIX e de outros sistemas operacionais. Os sistemas de arquivos também assumem a responsabilidade pelo controle do acesso aos arquivos, restringindo o acesso de acordo com as autorizações dos usuários e com o tipo de acesso solicitado (leitura, atualização, execução, etc.).

O termo *metadados* é usado freqüentemente para se referir a todas as informações extras armazenadas por um sistema de arquivos que são necessárias para o gerenciamento dos arquivos. Isso inclui atributos de arquivo, diretórios e todas as outras informações persistentes utilizadas pelo sistema de arquivos.

A Figura 8.2 mostra uma estrutura modular em camadas, típica, para a implementação de um sistema de arquivos não-distribuído em um sistema operacional convencional. Cada camada depende apenas das camadas que estão abaixo dela. A implementação de um serviço de arquivo distribuído precisa de todos os componentes lá mostrados, com componentes adicionais para tratar da comunicação cliente-servidor e da atribuição de nomes e da localização de arquivos distribuídos.

Operações do sistema de arquivos ◊ A Figura 8.4 resume as principais operações sobre arquivos que estão disponíveis nos sistemas UNIX para aplicativos. Essas são as chamadas de sistema implementadas pelo núcleo; normalmente, os programadores de aplicativos as acessam por meio de funções de biblioteca, como a biblioteca de entrada e saída padrão da linguagem C, ou as classes de arquivo Java. Fornecemos as primitivas como uma indicação das operações que os serviços de arquivo devem suportar e para comparação com as interfaces do serviço de arquivos que vamos apresentar a seguir.

As operações UNIX são baseadas em um modelo de programação no qual algumas informações sobre o estado do arquivo são armazenadas pelo sistema de arquivos para cada programa em execu-

Tamanho do arquivo
Horário de criação
Horário de acesso (leitura)
Horário de modificação (escrita)
Horário de alteração de atributo
Contagem de referência
Proprietário
Tipo de arquivo
Lista de controle de acesso

Figura 8.3 Estrutura básica de um registro de atributo de arquivo.

filedes = open(nome, modo)	Abre um arquivo existente com o *nome* fornecido.
filedes = creat(nome, modo)	Cria um novo arquivo com o *nome* fornecido.
	As duas operações produzem um descritor de arquivo referenciando o arquivo aberto. O *modo* é *read*, *write* ou ambos.
status = close(filedes)	Fecha o arquivo aberto referenciado por *filedes*.
count = read(filedes, buffer, n)	Transfere para *buffer n* bytes do arquivo referenciado por *filedes*
count = write(filedes, buffer, n)	Transfere *n* bytes de *buffer* para o arquivo referenciado por *filedes*
	As duas operações produzem o número de bytes realmente transferidos e avançam o ponteiro de leitura e escrita
pos = lseek(filedes, offset, whence)	Desloca o ponteiro de leitura e escrita para *offset* (relativo ou absoluto, dependendo de *whence*).
status = unlink(nome)	Remove o arquivo *nome* da estrutura de diretório. Se o arquivo não tiver outras referências, ele será excluído.
status = link(nome1, nome2)	Cria uma nova referência, ou nome (*nome2*), para um arquivo (*nome1*).
status = stat(nome, buffer)	Escreve os atributos do arquivo *nome* em *buffer*.

Figura 8.4 Operações do sistema de arquivos UNIX.

ção. Elas consistem, entre outros, em uma lista de arquivos correntemente abertos e um ponteiro de leitura e escrita que fornece a posição dentro do arquivo na qual será aplicada à próxima operação de leitura ou de escrita.

O sistema de arquivos é responsável por realizar o controle de acesso para os arquivos. Em sistemas de arquivos locais, como o UNIX, ele faz isso quando cada arquivo é aberto, verificando os direitos permitidos à identidade do usuário na lista de controle de acesso em relação ao *modo* de acesso solicitado na chamada de sistema *open*. Se os direitos corresponderem ao modo, o arquivo será aberto e o *modo* é armazenado nas informações de estado de arquivo aberto.

8.1.2 Requisitos do sistema de arquivos distribuído

Muitos dos requisitos e das armadilhas em potencial no projeto de serviços distribuídos foram detectados pela primeira vez no desenvolvimento inicial dos sistemas de arquivos distribuídos. Inicialmente, eles ofereciam transparência de acesso e de localização; surgiram requisitos de desempenho, escalabilidade, controle de concorrência, tolerância a falhas e segurança, e eles foram atendidos em fases subseqüentes de desenvolvimento. Vamos discutir esses e outros requisitos, nas subseções a seguir.

Transparência ◊ O serviço de arquivo normalmente é o mais usado em uma intranet; portanto, sua funcionalidade e seu desempenho são críticos. O projeto do serviço de arquivos deve suportar muitos dos requisitos de transparência dos sistemas distribuídos, identificados na Seção 1.4.7. O projeto deve contrabalançar a flexibilidade e a escalabilidade derivadas da transparência, com a complexidade e o desempenho do software. As seguintes formas de transparência são parcialmente, ou totalmente, tratadas pelos serviços de arquivos atuais:

Transparência do acesso: os programas clientes não devem conhecer a distribuição de arquivos. Um único conjunto de operações é fornecido para acesso a arquivos locais e remotos. Os programas escritos para operar sobre arquivos locais são capazes de acessar arquivos remotos sem modificação.

Transparência de localização: os programas clientes devem ver um espaço de nomes de arquivos uniforme. Os arquivos, ou grupos de arquivos, podem ser deslocados de um servidor a outro sem

alteração de seus nomes de caminho, e os programas de usuário devem ver o mesmo espaço de nomes onde quer que sejam executados.

Transparência de mobilidade: nem os programas clientes, nem as tabelas de administração de sistema nos nós clientes precisam ser alterados quando os arquivos são movidos. Isso permite a mobilidade do arquivo – arquivos ou, mais comumente, conjuntos ou volumes de arquivos podem ser movidos, ou pelos administradores de sistema ou automaticamente.

Transparência de desempenho: os programas clientes devem continuar a funcionar satisfatoriamente, enquanto a carga sobre o serviço varia dentro de um intervalo especificado.

Transparência de mudança de escala: o serviço pode ser expandido de forma paulatina, para tratar com uma ampla variedade de cargas e tamanhos de rede.

Atualizações concorrentes de arquivos ◊ As alterações feitas em um arquivo por um único cliente não devem interferir na operação de outros clientes que estejam acessando, ou alterando, o mesmo arquivo simultaneamente. Esse é o conhecido problema do controle de concorrência, discutido em detalhes no Capítulo 13. Em muitos aplicativos, a necessidade de controle de concorrência para acesso a dados compartilhados é amplamente aceita, e são conhecidas técnicas para sua implementação, mas elas são dispendiosas. A maior parte dos serviços de arquivo atuais segue os padrões UNIX modernos, fornecendo travamento (*locking*) em nível de arquivo ou em nível de registro.

Replicação de arquivos ◊ Em um serviço de arquivos que suporta replicação, um arquivo pode ser representado por várias cópias de seu conteúdo em diferentes locais. Isso tem duas vantagens – permite que vários servidores compartilhem a carga do fornecimento de um serviço para clientes que acessam o mesmo conjunto de arquivos, melhorando a escalabilidade do serviço, e melhora a tolerância a falhas, permitindo que, em caso de falhas, os clientes localizem outro servidor que contenha uma cópia do arquivo. Poucos serviços de arquivo suportam replicação completa, mas a maioria suporta o armazenamento de arquivos, ou de porções de arquivos, em caches locais, que é uma forma limitada de replicação. A replicação de dados será discutida no Capítulo 15, que inclui uma descrição do serviço de arquivos replicado Coda.

Heterogeneidade do hardware e do sistema operacional ◊ As interfaces de serviço devem ser definidas de modo que o software cliente e servidor possa ser implementado para diferentes sistemas operacionais e computadores. Esse requisito é um aspecto importante de sistemas abertos.

Tolerância a falhas ◊ Por ser parte essencial nos sistemas distribuídos, é essencial que o serviço de arquivo distribuídos continue a funcionar diante de falhas de clientes e servidores. Felizmente, um projeto moderadamente tolerante a falhas é fácil para servidores simples. Para suportar falhas de comunicação transientes, o projeto pode ser baseado na semântica de invocação *no máximo uma vez* (veja a Seção 5.2.4). Ou então, ele pode usar uma semântica mais simples, como *pelo menos uma vez*, com um protocolo de servidor projetado em termos de operações *idempotentes*, garantindo que pedidos duplicados não resultem em atualizações inválidas nos arquivos. Os servidores podem ser *sem estado* (*stateless*), para que após uma falha o serviço possa ser reiniciado e restaurado sem necessidade de recuperar o estado anterior. A tolerância às falhas de desconexão, ou de servidor, exige replicação de arquivo, a qual é mais difícil de obter (e será discutida no Capítulo 15).

Consistência ◊ Os sistemas de arquivos convencionais, como o fornecido no UNIX, oferecem *semântica de atualização de cópia única* (*one-copy*). Isso se refere a um modelo de acesso concorrente a arquivos, no qual o conteúdo do arquivo visto por todos os processos que estão acessando, ou atualizando determinado arquivo, é aquele que eles veriam se existisse apenas uma cópia do conteúdo do arquivo. Quando os arquivos são replicados, ou armazenados em cache, em diferentes sites, há um atraso inevitável na propagação das modificações feitas em um site para os outros sites que contêm cópias, e isso pode resultar em certo desvio da semântica de cópia única.

Segurança ◊ Praticamente todos os sistemas de arquivos fornecem mecanismos de controle de acesso baseados no uso de listas de controle de acesso. Nos sistemas de arquivos distribuídos, há necessidade de autenticar as requisições dos clientes para que o controle de acesso no servidor seja baseado nas identidades corretas de usuário e para proteger o conteúdo das mensagens de requisição e resposta

com assinaturas digitais e (opcionalmente) criptografia de dados secretos. Vamos discutir o impacto desses requisitos em nossas descrições de estudo de caso.

Eficiência ◊ Um serviço de arquivo distribuído deve oferecer recursos que tenham pelo menos o mesmo poder e generalidade daqueles encontrados nos sistemas de arquivos convencionais, e deve obter um nível de desempenho comparável. Birrell e Needham [1980] expressaram seus objetivos de projeto para o CFS (Cambridge File Server) nos seguintes termos:

> Desejaríamos ter um servidor de arquivos simples, de baixo nível, para compartilhar um recurso dispendioso, a saber, o disco, que nos deixasse livres para projetar um sistema de arquivos mais apropriado para um cliente em particular, ao mesmo tempo em que disponibilizasse um sistema de alto nível compartilhado entre os clientes.

A queda no custo do armazenamento em disco reduziu a importância do primeiro objetivo, mas a percepção da necessidade de uma gama de serviços tratando dos requisitos dos clientes com diferentes objetivos permanece e pode ser melhor resolvida por meio de uma arquitetura modular, do tipo esboçado anteriormente.

As técnicas usadas para a implementação de serviços de arquivos representam uma parte importante do projeto de sistemas distribuídos. Um sistema de arquivos distribuído deve fornecer um serviço que seja comparável (ou melhor) aos sistemas de arquivos locais, em termos de desempenho e confiabilidade. Ele deve ser conveniente para administrar, com operações e ferramentas que permitam aos administradores de sistema instalá-lo e operá-lo adequadamente.

8.1.3 Estudos de caso

Construiremos um modelo abstrato para um serviço de arquivo, para atuar como um exemplo introdutório, separando as preocupações com a implementação e fornecendo um modelo simplificado. Descreveremos o Sun Network File System com alguns detalhes, recorrendo ao nosso modelo abstrato mais simples para esclarecer sua arquitetura. O Andrew File System será descrito em seguida, fornecendo uma visão de um sistema de arquivos distribuído que adota uma estratégia diferente para escalabilidade e manutenção da consistência.

Arquitetura do serviço de arquivos ◊ Trata-se de um modelo abstrato de arquitetura que serve tanto para o NFS como para o AFS. Ele é baseado em uma divisão de responsabilidades entre três módulos – um módulo cliente, que simula uma interface de sistema de arquivos convencional para programas aplicativos, e dois módulos servidores que efetuam operações em diretórios e em arquivos. A arquitetura é projetada para permitir uma implementação *sem estado* (*stateless*) dos módulos servidores.

Sun NFS ◊ O NFS *(Network File System)* da Sun Microsystem foi amplamente adotado na indústria e nos ambientes acadêmicos, desde sua introdução, em 1985. O projeto e desenvolvimento do NFS foram feitos pelo pessoal da Sun Microsystems, em 1984 [Sandberg *et al.* 1985; Sandberg 1987, Callaghan 1999]. Embora vários serviços de arquivo distribuídos já tivessem sido desenvolvidos, e usados em universidades e laboratórios de pesquisa, o NFS foi o primeiro serviço de arquivo projetado como um produto. O projeto e a implementação do NFS obtiveram sucesso tanto técnico como comercial.

Para estimular sua adoção como padrão, as definições das principais interfaces foram colocadas em domínio público [Sun 1989], permitindo que outros fornecedores produzissem implementações, e o código-fonte de uma implementação de referência foi disponibilizado, sob licença, para outros fabricantes de computadores. Atualmente, ele é suportado por muitos fornecedores e o protocolo NFS (versão 3) é um padrão da Internet, definido no RFC 1813 [Callaghan *et al.* 1995]. O livro de Callaghan sobre o NFS [Callaghan 1999] é uma excelente fonte sobre o projeto e desenvolvimento do NFS e a respeito de tópicos relacionados.

O NFS fornece acesso transparente a arquivos remotos para programas clientes executando em UNIX e outros sistemas operacionais. O relacionamento cliente-servidor é simétrico: cada computador em uma rede NFS pode atuar como cliente e como servidor, e os arquivos de cada máquina podem se tornar disponíveis para acesso remoto por outras máquinas. Qualquer computador pode ser um servidor, exportando alguns de seus arquivos, e cliente, acessando arquivos de outras máquinas.

Mas é uma prática comum, em instalações maiores, configurar algumas máquinas como servidores dedicados e outras como estações de trabalho.

Um objetivo importante do NFS é obter um alto nível de suporte para heterogeneidade de hardware e sistema operacional. O projeto é independente do sistema operacional: existem implementações de cliente e servidor para quase todos os sistemas operacionais e plataformas, incluindo todas as versões do Windows, Mac OS, Linux e qualquer outra versão de UNIX. Implementações do NFS em *hosts* multiprocessadores de alto desempenho foram desenvolvidas por diversos fornecedores e são usadas para satisfazer os requisitos de armazenamento em intranets com muitos usuários concomitantes.

Andrew File System ◊ O Andrew é um ambiente de computação distribuída, desenvolvido na Carnegie Mellon University (CMU) para uso como sistema de computação e informação do campus [Morris *et al.* 1986]. O projeto do Andrew File System (daqui em diante abreviado como AFS) reflete a intenção de suportar o compartilhamento de informações em larga escala, minimizando a comunicação cliente-servidor. Isso foi conseguido pela transferência de arquivos inteiros entre computadores servidores e clientes, armazenando-os em cache nos clientes até que o servidor receba uma versão mais atualizada. Vamos descrever o AFS-2, a primeira implementação de "produção", seguindo as descrições de Satyanarayanan [1989a; 1989b]. Descrições mais recentes podem ser encontradas em Campbell [1997] e [Linux AFS].

Inicialmente, o AFS foi implementado na CMU em uma rede de estações de trabalho e servidores executando UNIX BSD e o sistema operacional Mach e, subseqüentemente, se tornou disponível em versões comerciais e de domínio público. Uma implementação de domínio público do AFS está disponível no sistema operacional Linux [Linux AFS]. O AFS foi adotado como a base do sistema de arquivos DCE/DFS no DCE (*Distributed Computing Environment*) da Open Software Foundation [www.opengroup.org]. O projeto do DCE/DFS foi além do AFS sob vários aspectos importantes, os quais destacaremos na Seção 8.5.

8.2 Arquitetura do serviço de arquivos

Uma arquitetura que apresenta uma separação clara das principais preocupações no fornecimento de acesso aos arquivos é obtida por meio da estruturação do serviço de arquivo em três componentes – um *serviço de arquivos plano*, um *serviço de diretório* e um *módulo cliente*. Os módulos relevantes e seus relacionamentos aparecem na Figura 8.5. O serviço de arquivos plano e o serviço de diretório exportam uma interface para uso dos programas clientes, e suas interfaces RPC, consideradas em grupo, fornecem um amplo conjunto de operações para acesso aos arquivos. O módulo cliente fornece uma única interface de programação, com operações sobre arquivos semelhantes àquelas encontradas

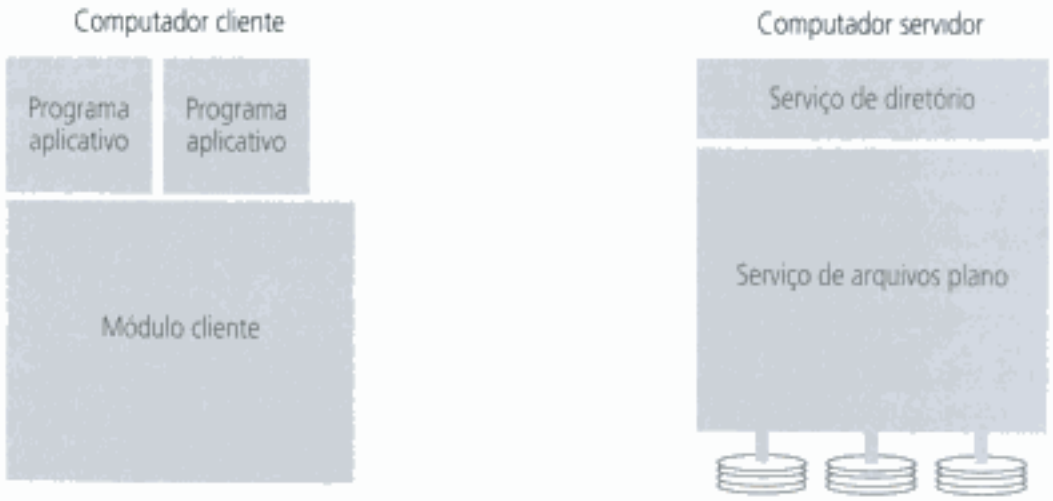


Figura 8.5 Arquitetura do serviço de arquivos.

nos sistemas de arquivos convencionais. O projeto é *aberto*, no sentido de que vários módulos clientes podem ser usados para implementar distintas interfaces de programação, simulando as operações de arquivo de vários sistemas operacionais e otimizando o desempenho para diferentes configurações de hardware de cliente e servidor.

A divisão de responsabilidades entre os módulos pode ser definida como segue:

Serviço de arquivos plano (*flat file service*) ◊ O serviço de arquivos plano se preocupa com a implementação de operações sobre o conteúdo dos arquivos. São usados *identificadores exclusivos de arquivo* (UFIDs – *Unique File IDentifiers*) para fazer referência aos arquivos em todas as requisições de operações ao serviço de arquivos plano. A divisão de responsabilidades entre o serviço de arquivos e o serviço de diretório é baseada no uso de UFIDs. As UFIDs são longas seqüências de bits, escolhidas de modo que cada arquivo tenha um UFID único dentre todos os arquivos de um sistema distribuído. Quando o serviço de arquivos plano recebe um pedido para criar um arquivo, ele gera um novo UFID e retorna esse UFID para o solicitante.

Serviço de diretório ◊ O serviço de diretório fornece um mapeamento entre *nomes textuais* de arquivos e seus UFIDs. Os clientes podem obter o UFID de um arquivo citando seu nome textual para o serviço de diretório. O serviço de diretório fornece as funções necessárias para gerar diretórios, adicionar novos nomes de arquivo a eles e para obter suas UFIDs. Ele é um cliente do serviço de arquivos plano; os arquivos de seu diretório são armazenados em arquivos do serviço de arquivos plano. Quando um esquema de atribuição de nomes hierárquico é adotado, como no UNIX, os diretórios contêm referências para outros diretórios.

Módulo cliente ◊ Um módulo cliente é executado em cada computador cliente, integrando e estendendo as operações do serviço de arquivos plano e do serviço de diretório sob uma interface de programação de aplicativo única, disponível para programas em nível de usuário nos computadores clientes. Por exemplo, em *hosts* UNIX seria fornecido um módulo cliente que simularia o conjunto completo de operações de arquivos UNIX, interpretando nomes de arquivos por meio de pedidos iterativos ao serviço de diretório. O módulo cliente também contém informações sobre os locais de rede dos processos servidor de arquivos plano e servidor de diretório. Finalmente, o módulo cliente pode assumir um papel importante na obtenção de um desempenho satisfatório, por meio da implementação de uma cache de blocos de arquivo recentemente usados no cliente.

Interface do serviço de arquivos plano ◊ A Figura 8.6 contém uma definição da interface para um serviço de arquivos planos. Essa é a interface RPC usada pelos módulos clientes. Normalmente, ela não é usada diretamente por programas em nível de usuário. Um argumento *FileId* é inválido se o arquivo a que se refere não está presente no servidor que processa o pedido, ou se suas permissões de acesso são inadequadas para a operação solicitada. Todas as funções da interface, exceto *Create*, geram exceções se o argumento *FileId* contém um UFID inválido, ou se o usuário não tem direitos de acesso suficientes. Essas exceções foram omitidas da definição por motivos de clareza.

Read(FileId, i, n) → Data — gera *BadPosition*	Se $1 \le i \le Length(File)$: lê uma seqüência de até n elementos de um arquivo, começando no elemento i, e a retorna em *Data*. Gera uma exceção se o valor i é inválido.
Write(FileId, i, Data) — gera *BadPosition*	Se $1 \le i \le Length(File)+1$: grava uma seqüência de *Data* em um arquivo, começando no elemento i, ampliando o arquivo, se necessário. Gera uma exceção se o valor i é inválido.
Create() → FileId	Cria um novo arquivo de tamanho 0 e gera um UFID para ele.
Delete(FileId)	Remove o arquivo.
GetAttributes(FileId) → Attr	Retorna os atributos do arquivo.
SetAttributes(FileId, Attr)	Configura os atributos do arquivo (somente os atributos que não estão sombreados na Figura 8.3).

Figura 8.6 Operações do serviço de arquivos plano.

As operações mais importantes são as de leitura e escrita. Tanto a operação *Read* (leitura) como a operação *Write* (escrita) necessitam de um parâmetro *i* para especificar uma posição no arquivo. A operação *Read* copia para *Data* a seqüência de *n* elementos de dados a partir do elemento *i* do arquivo especificado. A operação *Write* escreve, a partir do elemento *i*, a seqüência de elementos de dados em *Data* para o arquivo especificado, substituindo o conteúdo anterior do arquivo na posição correspondente e, se necessário, aumentando o arquivo.

Create cria um novo arquivo vazio e retorna o UFID gerado. *Delete* remove o arquivo especificado.

GetAttributes e *SetAttributes* permitem que os clientes acessem o registro de atributos. Normalmente, *GetAttributes* está disponível para qualquer cliente que tenha permissão para ler o arquivo. Normalmente, o acesso à operação *SetAttributes* é restrito ao serviço de diretório que dá acesso ao arquivo. Os valores das partes relativas ao tamanho e à indicação de tempo do registro de atributos não são afetados por *SetAttributes*; eles são mantidos separadamente pelo próprio serviço de arquivos plano.

Comparação com o UNIX: nossa interface e as primitivas do sistema de arquivos UNIX são funcionalmente equivalentes. Basta construir um módulo cliente que simule as chamadas de sistema UNIX em termos de nosso serviço de arquivos plano e das operações do serviço de diretório descritas na próxima seção.

Em comparação com a interface UNIX, nosso serviço de arquivos plano não tem as operações *open* e *close* – os arquivos podem ser acessados imediatamente citando-se o UFID apropriado. Em nossa interface, as funções *Read* e *Write* incluem um parâmetro especificando um ponto de partida dentro do arquivo para cada transferência, enquanto as operações equivalentes do UNIX não incluem. No UNIX, cada operação *read* ou *write* começa na posição corrente do ponteiro de leitura e escrita, e esse ponteiro avança de acordo com o número de bytes transferidos após cada operação *read* ou *write*. É fornecida uma operação *seek* para permitir que o ponteiro de leitura e escrita seja posicionado explicitamente.

A interface para nosso serviço de arquivos plano difere da interface de sistema de arquivos do UNIX principalmente por motivos de tolerância a falhas:

Operações que podem ser repetidas: com exceção de *Create*, as operações são *idempotentes*, permitindo o uso da semântica RPC *pelo menos uma vez* – os clientes podem repetir chamadas para as quais não receberam resposta. A execução repetida de *Create* produz um novo arquivo diferente para cada chamada.

Servidores sem estado (*stateless*): a interface é conveniente para a implementação por servidores sem estado. Um servidor sem estado pode ser reiniciado após uma falha e retomar a operação sem necessidade dos clientes, ou do servidor, para restaurar qualquer estado.

As operações de arquivo do UNIX não são idempotentes, nem consistentes, com o requisito de uma implementação sem estado. Quando um arquivo é aberto, um ponteiro de leitura e escrita é gerado pelo sistema de arquivos do UNIX, e ele é mantido, junto com os resultados das verificações de controle de acesso, até que o arquivo seja fechado. As operações *read* e *write* do UNIX não são idempotentes; se uma operação for repetida acidentalmente, o avanço automático do ponteiro de leitura e escrita resultará no acesso a uma parte diferente do arquivo na operação repetida. O ponteiro de leitura e escrita é uma variável (oculta) de estado relacionado ao cliente. Para imitá-la em um servidor de arquivos, seriam necessárias as operações *open* e *close*, e o valor do ponteiro de leitura e escrita teria de ser mantido pelo servidor, enquanto o arquivo relevante estivesse aberto. Eliminando o ponteiro de leitura e escrita, eliminamos a parte da necessidade do servidor de arquivos manter informações de estado em nome de clientes específicos.

Controle de acesso ◊ No sistema de arquivos do UNIX, os direitos de acesso do usuário são verificados com relação ao *modo* de acesso (leitura ou escrita) solicitado na chamada *open* (a Figura 8.4 mostra a API de sistema de arquivos do UNIX), e o arquivo é aberto somente se o usuário tiver os direitos necessários. A identidade de usuário (UID) utilizada na verificação dos direitos de acesso é o resultado do *login* anteriormente autenticado do usuário e não pode ser falsificada em implementações não-distribuídas. Os direitos de acesso resultantes são mantidos até que o arquivo seja fechado, e mais nenhuma verificação é exigida quando são solicitadas operações subseqüentes no mesmo arquivo.

Nas implementações distribuídas, as verificações de direitos de acesso não precisam ser realizadas no servidor, pois, sob outros aspectos, a interface RPC do servidor é um ponto desprotegido de acesso aos arquivos. Uma identidade de usuário precisa ser passada com as requisições e o servidor é vulnerável a identidades falsificadas. Além disso, se os resultados de uma verificação de direitos de acesso fossem mantidos no servidor e usados para acessos futuros, o servidor não seria mais sem estado. Podem ser adotadas duas estratégias alternativas para este último problema:

- É feita uma verificação de acesso quando um nome de arquivo é convertido em um UFID e os resultados são codificados na forma de uma capacidade (veja a Seção 7.2.4), a qual é retornada para o cliente, para envio com as requisições subseqüentes.
- Uma identidade de usuário é enviada com cada requisição de cliente e as verificações de acesso são realizadas pelo servidor para cada operação de arquivo.

Os dois métodos permitem implementação de servidor sem estado e ambos têm sido usados em sistemas de arquivos distribuídos. O segundo é mais comum; ele é usado no NFS e no AFS. Nenhuma dessas estratégias resolve o problema de segurança relativo às identidades de usuário falsificadas. Vimos, no Capítulo 7, que isso pode ser tratado com o uso de assinaturas digitais. O Kerberos é um esquema de autenticação eficaz que tem sido aplicado no NFS e no AFS.

Em nosso modelo abstrato, não fazemos nenhuma suposição sobre o método pelo qual o controle de acesso é implementado. A identidade do usuário é passada como um parâmetro implícito e pode ser usada quando for necessário.

Interface do serviço de diretório ◊ A Figura 8.7 contém uma definição da interface RPC para um serviço de diretório. O principal objetivo do serviço de diretório é fornecer um serviço para a transformação de nomes textuais em UFIDs. Para isso, ele mantém arquivos de diretório contendo os mapeamentos entre nomes textuais de arquivos e UFIDs. Cada diretório é armazenado como um arquivo convencional com um UFID, de modo que o serviço de diretório é um cliente do serviço de arquivos.

Definimos apenas operações sobre diretórios individuais. Para cada operação, é exigido o UFID do arquivo que contém o diretório (no parâmetro *Dir*). A operação *Lookup* no serviço de diretório realiza a transformação *Nome* → *UFID*. Ela é um bloco básico para uso em outros serviços, ou no módulo cliente, para realizar transformações mais complexas, como a interpretação de nome hierárquica encontrada no UNIX. Como antes, as exceções causadas por direitos de acesso inadequados foram omitidas das definições.

Existem duas operações para alterar diretórios: *AddName* e *UnName*. *AddName* adiciona uma entrada em um diretório e incrementa o campo de contagem de referência no registro de atributos do arquivo.

UnName remove uma entrada de um diretório e decrementa a contagem de referência. Se a contagem de referência chegar à zero, o arquivo é removido. *GetNames* é fornecida para permitir que os clientes examinem o conteúdo dos diretórios e para implementar operações de correspondência de padrão em nomes de arquivo, como aquelas encontradas no *shell* do UNIX. Ela retorna todos os nomes

Lookup(Dir, Name) → *FileId* — gera *NotFound*	Localiza o nome textual no diretório e retorna o UFID relevante. Se *Name* não estiver no diretório, gera uma exceção.
AddName(Dir, Name, FileId) — gera *NameDuplicate*	Se *Name* não estiver no diretório, adiciona *(Name, File)* no diretório e atualiza o registro de atributos do arquivo. Se *Name* já estiver no diretório, gera uma exceção.
UnName(Dir, Name) — gera *NotFound*	Se *Name* estiver no diretório, a entrada contendo *Name* é removida do diretório. Se *Name* não estiver no diretório, gera uma exceção.
GetNames(Dir, Pattern) → *NameSeq*	Retorna todos os nomes textuais presentes no diretório que correspondam à expressão regular *Pattern*.

Figura 8.7 Operações do serviço de diretório.

(ou um subconjunto deles) armazenados em determinado diretório. Os nomes são selecionados pela correspondência de padrão contra uma expressão regular fornecida pelo cliente.

A existência da correspondência de padrão na operação *GetNames* permite que os usuários determinem os nomes de um ou mais arquivos fornecendo uma especificação incompleta dos caracteres presentes nos nomes. Uma expressão regular é uma especificação para uma classe de *strings*, na forma de uma expressão contendo uma combinação de *substrings* literais e símbolos denotando caracteres variáveis ou ocorrências repetidas de caracteres ou *substrings*.

Sistema de arquivos hierárquico ◊ Um sistema de arquivos hierárquico, como aquele fornecido pelo UNIX, consiste em vários diretórios organizados em uma estrutura em árvore. Cada diretório contém os nomes dos arquivos e de outros diretórios que podem ser acessados a partir dele. Qualquer arquivo ou diretório pode ser referenciado usando-se um *nome de caminho* (*pathname*) – um nome composto de várias partes que representa um caminho através da árvore. A raiz tem um nome distinto e cada arquivo, ou (sub)diretório, possui um nome definido pelo usuário e armazenado em um diretório. O esquema de atribuição de nomes de arquivo do UNIX não é rigorosamente hierárquico – os arquivos podem ter vários nomes (*alias*) que podem estar no mesmo diretório ou em diretórios diferentes. Isso é implementado por meio de uma operação *link*, a qual adiciona um novo nome para um arquivo em um diretório especificado.

Um sistema de atribuição de nomes de arquivos como o do UNIX pode ser implementado pelo módulo cliente através dos serviços de arquivos plano e de diretório já definidos. Uma hierarquia de diretórios estruturada em árvore é construída com arquivos nas folhas e diretórios nos outros nós da árvore. A raiz da árvore é um diretório com um UFID conhecido. É possível atribuir vários nomes a um arquivo através da operação *AddName* e com o campo de contagem de referência no registro de atributos.

No módulo cliente, pode ser fornecida uma função que obtenha o UFID de um arquivo a partir de seu nome de caminho. A função interpreta o nome de caminho a partir da raiz, usando *Lookup* para obter o UFID de cada parte presente no caminho.

Em um serviço de diretório hierárquico, os atributos associados aos arquivos devem incluir um campo de tipo que faça a distinção entre arquivos normais e de diretórios. Isso é usado ao se seguir um caminho para garantir que cada parte do nome, exceto a última, se refira a um diretório.

Grupos de arquivos ◊ Um *grupo de arquivos* é um conjunto de arquivos localizado em determinado servidor. Um servidor pode conter vários grupos de arquivos e os grupos podem ser movidos entre os servidores, mas um arquivo não pode mudar do grupo a que pertence. Uma construção semelhante (chamada de *sistema de arquivos*) é usada no UNIX e na maioria dos outros sistemas operacionais. Os grupos de arquivos foram introduzidos originalmente para suportar recursos para mover conjuntos de arquivos armazenados em mídia removível entre computadores. Em um serviço de arquivos distribuído, os grupos de arquivos suportam a alocação de arquivos em unidades lógicas maiores e permitem que o serviço seja implementado com arquivos armazenados em vários servidores. Em um sistema de arquivos distribuído que suporta grupos de arquivos, a representação de UFIDs inclui um componente identificador de grupo de arquivos permitindo que o módulo cliente assuma a responsabilidade por enviar as requisições para o servidor que contém o grupo de arquivos em questão.

Os identificadores de grupo de arquivos devem ser únicos em todo um sistema distribuído. Como os grupos de arquivos podem ser movidos, os sistemas de arquivos que são inicialmente separados podem ser combinados para formar um só sistema de arquivo, e a única maneira de garantir que os identificadores de grupo de arquivos sejam sempre diferentes em determinado sistema de arquivo é gerá-los com um algoritmo que garanta a exclusividade global. Por exemplo, quando um novo grupo de arquivos é criado, um identificador exclusivo pode ser gerado pela concatenação do endereço IP de 32 bits do *host* que está criando o novo grupo, com um inteiro de 16 bits derivado da data, produzindo um inteiro único de 48 bits:

	32 bits	16 bits
identificador de grupo de arquivos:	**endereço IP**	**data**

Note que o endereço IP não pode ser usado para localizar o grupo de arquivos, pois ele pode ser movido para outro servidor. Em vez disso, o serviço de arquivo deve manter um mapeamento entre os identificadores de grupo e os servidores.

8.3 Estudo de caso: Sun Network File System

A Figura 8.8 mostra a arquitetura do NFS. Ela segue o modelo abstrato definido na seção anterior. Todas as implementações de NFS suportam o protocolo NFS – um conjunto de chamadas de procedimentos remotos que fornece o meio para os clientes efetuarem operações em um meio de armazenamento de arquivos remoto. O protocolo NFS é independente do sistema operacional, mas foi originalmente desenvolvido para uso em redes UNIX, e vamos descrever a implementação UNIX do protocolo NFS (versão 3).

O módulo *servidor NFS* reside no núcleo de cada computador que atua como servidor NFS. As requisições que se referem a arquivos de um sistema de arquivos remoto são transformados em operações do protocolo NFS pelo módulo cliente e depois passados para o módulo servidor NFS no computador que contém o sistema de arquivos em questão.

Os módulos cliente e servidor NFS se comunicam usando chamadas de procedimentos remotos. O sistema RPC da Sun, descrito na Seção 5.3.1, foi desenvolvido para ser usado no NFS. Ele pode ser configurado para usar UDP ou TCP, e o protocolo NFS é compatível com ambos. É incluído um serviço mapeador de porta (*portmapper*) para permitir que os clientes se associem, por meio de um nome, aos serviços de determinado *host*. A interface RPC do servidor NFS é aberta: qualquer processo pode enviar requisições para um servidor NFS; caso as requisições sejam válidas e incluam credenciais de usuários autorizados, elas serão atendidas. Como um recurso de segurança opcional, pode-se exigir o envio de credenciais de usuário assinadas, assim como a criptografia dos dados para se obter privacidade e integridade.

Sistema de arquivos virtual ◊ A Figura 8.8 torna claro que o NFS fornece transparência de acesso: os programas de usuário podem executar operações de arquivo, em arquivos locais ou remotos, sem distinção. Podem estar presentes outros sistemas de arquivos distribuídos que suportem chamadas de sistema UNIX e, se assim for, eles seriam integrados da mesma maneira.

A integração é obtida por meio de um módulo de sistema de arquivos virtual (VFS – *Virtual File System*), o qual foi adicionado no núcleo do UNIX para fazer a distinção entre arquivos locais e

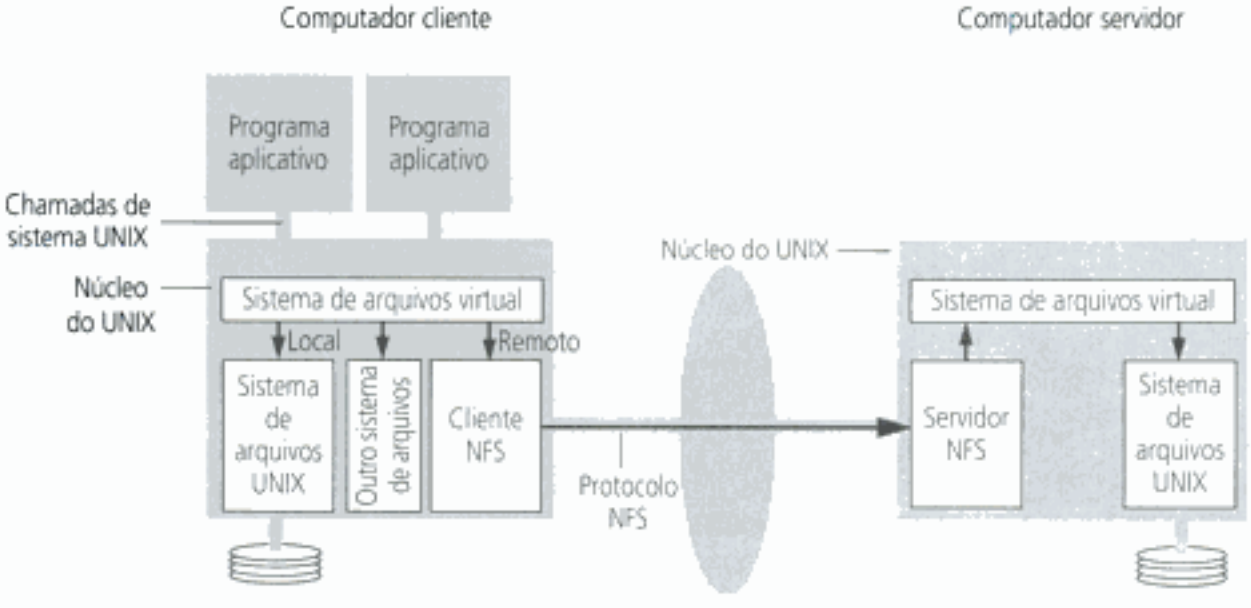


Figura 8.8 Arquitetura do NFS.

remotos, e para transformar os identificadores de arquivo usados pelo NFS em identificadores internos normalmente usados no UNIX e em outros sistemas de arquivos. Além disso, o VFS controla os sistemas de arquivos que estão correntemente disponíveis tanto de forma local como remota e passa cada requisição para o módulo apropriado (o sistema de arquivos UNIX, o módulo de cliente NFS ou o módulo de serviço de outro sistema de arquivos).

Os identificadores de arquivo usados no NFS são chamados de *manipuladores de arquivo*. Um manipulador de arquivo não é visível para os clientes e contém as informações de que o servidor precisa para distinguir um arquivo individual. Nas implementações UNIX do NFS, o manipulador de arquivo é derivado do *i-node* do arquivo, adicionando-se dois campos extras, como dado a seguir (o *i-node* de um arquivo UNIX é um número que serve para identificar e localizar o arquivo dentro do sistema de arquivos no qual ele está armazenado):

O NFS adota a montagem de sistemas de arquivos* (*filesystem*) do UNIX como unidade de agrupamento de arquivos, definida na seção anterior. O campo *identificador de filesystem* é um número exclusivo alocado para cada sistema de arquivos (*filesystem*) quando ele é criado (e, na implementação UNIX, é armazenado no superbloco do sistema de arquivos). O número de geração do *i-node* é necessário porque no sistema de arquivos UNIX convencional os *i-nodes* são reutilizados após um arquivo ser removido. Nas extensões VFS do sistema de arquivos UNIX, um número de geração é armazenado com cada arquivo e é incrementado sempre que um *i-node* é reutilizado (por exemplo, em uma chamada de sistema *creat*). O cliente obtém o primeiro manipulador de arquivo para um sistema de arquivo remoto ao montá-lo. Os manipuladores de arquivo são passados do servidor para o cliente nos resultados das operações *lookup*, *create* e *mkdir* (veja a Figura 8.9) e do cliente para o servidor como um dos argumentos que as requisições contêm.

A camada de sistema de arquivos virtual tem uma estrutura VFS para cada sistema de arquivos (*filesystem*) montado e um *v-node* por arquivo aberto. Uma estrutura VFS relaciona um sistema de arquivo remoto com o diretório local em que ele é montado. O *v-node* contém uma indicação se um arquivo é local ou remoto. Se o arquivo é local, o *v-node* contém uma referência para o índice do arquivo local (um *i-node* em uma implementação UNIX). Se o arquivo é remoto, ele contém o manipulador de arquivo do arquivo remoto.

Integração com o cliente ◊ O módulo cliente NFS desempenha o papel descrito para o módulo cliente em nosso modelo básico, fornecendo uma interface conveniente para os programas aplicativos. Mas, ao contrário de nosso modelo de módulo cliente, ele simula precisamente a semântica das primitivas padrão do sistema de arquivos UNIX e é integrado com o núcleo UNIX. As principais razões para que ele seja integrado, e não fornecido como uma biblioteca a ser ligada nos processos clientes, são para permitir que:

- os programas de usuário possam acessar arquivos por meio de chamadas de sistema UNIX sem serem ligados e compilados novamente;
- um único módulo cliente atenda todos os processos em nível de usuário, com uma cache compartilhada dos blocos usados recentemente (descrito a seguir);
- a chave de criptografia usada para autenticar as IDs de usuário passadas para o servidor (veja a seguir) possa ser mantida no núcleo, impedindo a personificação por parte de clientes em nível de usuário.

* N. de R.T.: Em inglês, há uma distinção entre os termos *filesystem* (uma única palavra) e *file system* (duas palavras). Por *filesystem*, entende-se o conjunto de arquivos e diretórios mantidos em uma partição de um dispositivo de armazenamento; *file system* refere-se ao módulo de software do sistema operacional que provê acesso aos arquivos. Corriqueiramente, em português, o termo *sistema de arquivos* engloba as duas funcionalidades e é empregado para referenciar ambos. Quando essa distinção for necessária, usaremos os termos em inglês entre parênteses.

O módulo cliente NFS coopera com o sistema de arquivos virtual em cada máquina cliente. Ele funciona de maneira semelhante ao sistema de arquivos convencional do UNIX, transferindo blocos de arquivos para o servidor (e dele) e, quando possível, colocando os blocos em cache na memória local. Ele compartilha a mesma cache usada pelo sistema de entrada e saída local. Porém, como vários clientes, em diferentes *hosts*, podem acessar simultaneamente o mesmo arquivo remoto, surge um problema de consistência de cache.

Controle de acesso e autenticação ◊ Ao contrário do sistema de arquivos UNIX convencional, o servidor NFS é sem estado e não mantém arquivos abertos no nome de seus clientes. Portanto, a cada requisição, o servidor deve verificar novamente a identidade do usuário nos atributos de permissão de acesso do arquivo, para ver se o usuário pode acessar o arquivo da maneira solicitada. O protocolo RPC Sun exige que os clientes enviem informações de autenticação de usuário (por exemplo, a ID de usuário e a ID de convencionais – em 16 bits – do UNIX) com cada requisição e isso é verificado em relação à permissão de acesso presente nos atributos do arquivo. Esses parâmetros adicionais não aparecem em nossa visão geral do protocolo NFS da Figura 8.9; eles são fornecidos automaticamente pelo sistema RPC.

Em sua forma mais simples, existe uma brecha de segurança nesse mecanismo de controle de acesso. Um servidor NFS fornece uma interface RPC convencional em uma porta bem conhecida em cada *host* e qualquer processo pode se comportar como cliente, enviando requisições para o servidor para acessar ou atualizar um arquivo. O cliente pode modificar as chamadas de RPC para incluir a ID de qualquer usuário, personificando-o sem seu conhecimento ou permissão. Essa brecha de segurança foi fechada pelo uso de uma opção do protocolo RPC para a criptografia DES das informações de autenticação do usuário. Mais recentemente, o Kerberos foi integrado com o Sun NFS para proporcionar uma solução mais forte e abrangente para os problemas da autenticação de usuário e da segurança, e descreveremos isso a seguir.

Interface do servidor NFS ◊ Uma representação simplificada da interface RPC fornecida pelos servidores NFS versão 3 (definidos no RFC 1813 [Callaghan *et al.* 1995]) aparece na Figura 8.9. As operações de acesso a arquivo do NFS, *read, write, getattr* e *setattr*, são quase idênticas às operações *Read, Write, GetAttributes* e *SetAttributes* definidas para nosso modelo de serviço de arquivo simples (Figura 8.6). A operação *lookup* e a maior parte das outras operações de diretório, definidas na Figura 8.9, são semelhantes àquelas de nosso modelo de serviço de diretório (Figura 8.7).

As operações de arquivo e diretório são integradas em um único serviço; a criação e a inserção de nomes de arquivo em diretórios são realizadas por uma única operação *create*, que recebe como argumentos o nome textual do novo arquivo e o manipulador do arquivo do diretório de destino. As outras operações NFS sobre diretórios são *create, remove, rename, link, symlink, readlink, mkdir, rmdir, readdir* e *statfs*. Elas são semelhantes às suas correlatas do UNIX, com exceção de *readdir*, que fornece um método independente de representação para ler o conteúdo de diretórios, e *statfs*, que fornece as informações de status em sistemas de arquivos remotos.

Serviço de montagem ◊ A montagem de subárvores de sistemas de arquivos (*filesystems*) remotos feita por clientes é suportada por um processo separado de *serviço de montagem*, executado em nível de usuário em cada computador servidor NFS. Em cada servidor existe um arquivo com um nome conhecido (*/etc/exports*), contendo os nomes dos sistemas de arquivos locais (*local filesystems*) que estão disponíveis para montagem remota. Uma lista de acesso é associada a cada nome de sistema de arquivos (*filesystem*), indicando quais *hosts* podem montá-los.

Os clientes usam uma versão modificada do comando *mount* do UNIX para solicitar a montagem de um sistema de arquivos remoto, especificando o nome de *host* remoto, o nome de caminho de um diretório no sistema de arquivos remoto e o nome local com o qual ele será montado. O diretório remoto pode ser qualquer subárvore do sistema de arquivo remoto solicitado, permitindo aos clientes montarem qualquer parte sua. O comando *mount* modificado se comunica com o processo do serviço de montagem no *host* remoto, usando um *protocolo de montagem*. Trata-se de um protocolo RPC que inclui uma operação que recebe um nome de caminho de diretório e retorna o manipulador do arquivo do diretório especificado, caso o cliente tenha permissão de acesso para o sistema de arquivos relevante. A localização (endereço IP e número de porta) do servidor e do manipulador de arquivo do diretório remoto são passados na camada VFS e no cliente NFS.

lookup(dirfh, name) → *fh, attr*	Retorna o manipulador de arquivo e os atributos do arquivo *name* no diretório *dirfh*.
create(dirfh, name, attr) → *newfh, attr*	Cria um arquivo *name* no diretório *dirfh* com atributos *attr* e retorna o novo manipulador de arquivo e os seus atributos.
remove(dirfh, name) → *status*	Remove o arquivo *name* do diretório *dirfh*.
getattr(fh) → *attr*	Retorna os atributos do arquivo *fh*. (Semelhante à chamada de sistema *stat* do UNIX.)
setattr(fh, attr) → *attr*	Configura os atributos (modo, ID de usuário, ID de grupo, tamanho, horário de acesso e de modificação de um arquivo). Configurar o tamanho como 0 trunca o arquivo.
read(fh, offset, count) → *attr, data*	Retorna até *count* bytes de dados de um arquivo, a partir de *offset*. Além disso, retorna os últimos atributos do arquivo.
write(fh, offset, count, data) → *attr*	Grava *count* bytes de dados em um arquivo, a partir de *offset*. Retorna os atributos do arquivo após a gravação terminar.
rename(dirfh, name, todirfh, toname) → *status*	Muda o nome do arquivo *name* no diretório *dirfh* para *toname* no diretório para *dirfh*.
link(newdirfh, newname, fh) → *status*	Cria uma entrada *newname* no diretório *newdirfh* referindo-se ao arquivo *ou* diretório *fh*.
symlink(newdirfh, newname, string) → *status*	Cria uma entrada *newname* no diretório *newdirfh* de tipo *vínculo simbólico* (*link* simbólico) com o valor *string*. O servidor não interpreta o *string*, mas produz um arquivo de vínculo simbólico para contê-la.
readlink(fh) → *string*	Retorna o string associado ao arquivo de vínculo simbólico identificado por *fh*.
mkdir(dirfh, name, attr) → *newfh, attr*	Cria um novo diretório *name* com atributos *attr* e retorna seu manipulador de arquivo e atributos.
rmdir(dirfh, name) → *status*	Remove o diretório vazio *name* do diretório pai *dirfh*. Falha se o diretório não estiver vazio.
readdir(dirfh, cookie, count) → *entries*	Retorna até *count* bytes de entradas do diretório *dirfh*. Cada entrada contém um nome de arquivo, um manipulador de arquivo e um ponteiro para a próxima entrada de diretório, chamado *cookie*. O *cookie* é usado nas chamadas de *readdir* subseqüentes para começar a ler a partir da entrada seguinte. Se o valor de *cookie* for 0, lê a partir da primeira entrada no diretório.
statfs(fh) → *fsstats*	Retorna informações do sistema de arquivos (como o tamanho do bloco, número de blocos livres, etc.) que contém um arquivo *fh*.

Figura 8.9 Operações do servidor NFS (protocolo NFS versão 3, simplificado).

A Figura 8.10 ilustra um cliente *Cliente* com dois sistemas de arquivos montados de forma remota. Os nós *people* e *users* nos sistemas de arquivos de *Servidor 1* e *Servidor 2* são montados sobre os nós *students* e *staff* no sistema de arquivos local de *Cliente*. O significado disso é que os programas executados em *Cliente* podem acessar arquivos em *Servidor 1* e em *Servidor 2*, usando nomes de caminho como /usr/students/jon e /usr/staff/ann.

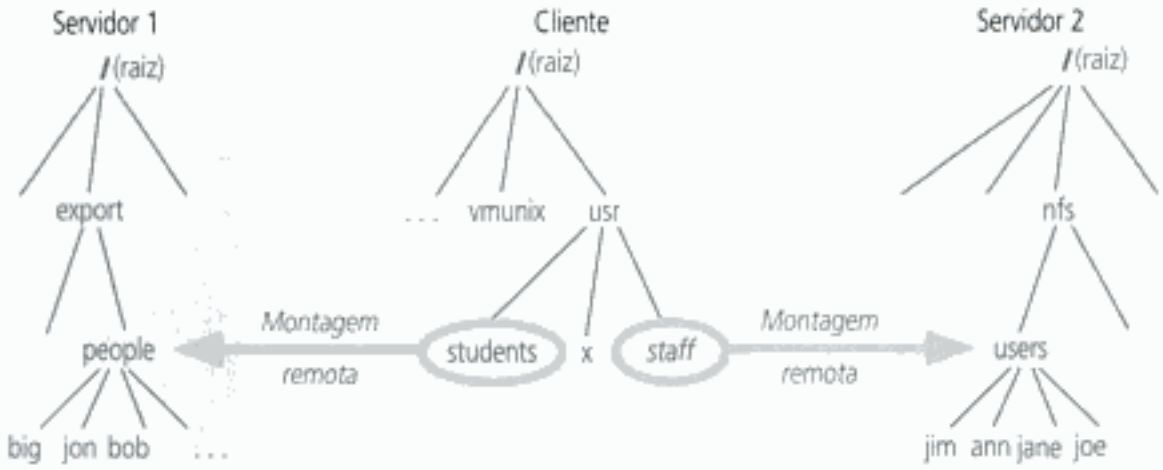


Figura 8.10 **Sistemas de arquivos local e remoto acessíveis em um cliente NFS.**
Nota: O sistema de arquivos montado em */usr/students* no cliente é, na verdade, a subárvore localizada em */export/people* no Servidor 1; o sistema de arquivos montado em */usr/staff* no cliente é, na verdade, a subárvore localizada em */nfs/users* no Servidor 2.

Os sistemas de arquivos remotos podem ser *montados incondicionalmente* ou *condicionalmente* em um computador cliente. Quando um processo em nível de usuário acessa um arquivo em um sistema de arquivos montado incondicionalmente, o processo é suspenso até que a requisição possa ser concluída e, se o *host* remoto estiver indisponível por qualquer razão, o módulo cliente NFS continuará a fazer novas tentativas até ser atendido. Assim, no caso de uma falha do servidor, os processos em nível de usuário são suspensos até que o servidor seja reiniciado e depois continuam normalmente. Mas se o sistema de arquivos relevante for montado condicionalmente, o módulo cliente NFS retornará uma indicação de erro para os processos em nível de usuário, após um pequeno número de novas tentativas. Então, os programas construídos corretamente detectarão a falha e executarão ações apropriadas de recuperação ou de geração de relatórios. Entretanto, muitos utilitários e aplicativos UNIX não testam a existência de falhas em operações de acesso a arquivo e se comportam de maneiras imprevisíveis no caso de falha de um sistema de arquivos montado condicionalmente. Por isso, muitas instalações utilizam exclusivamente a montagem incondicional, com a conseqüência de que os programas são incapazes de se recuperar normalmente quando um servidor NFS fica indisponível por um período de tempo significativo.

Tradução de nomes de caminho ◊ Os sistemas de arquivos UNIX traduzem *nomes de caminho* de arquivos composto por muitas partes em referências de *i-nodes*, em um processo passo a passo, quando são usadas as chamadas de sistema *open*, *creat* ou *stat*. No NFS, os nomes de caminho não podem ser traduzidos em um servidor, pois um nome pode cruzar um ponto de montagem no cliente – diretórios contendo várias partes em nome podem residir em sistemas de arquivos de diferentes servidores. Portanto, os nomes de caminho são analisados e sua tradução é realizada de maneira iterativa pelo cliente. Cada parte de um nome que se refere a um diretório montado de forma remota é transformado em um manipulador de arquivo usando uma requisição de *lookup* separada para o servidor remoto.

A operação *lookup* procura uma única parte de um nome de caminho em determinado diretório e retorna o manipulador de arquivo e os atributos correspondentes. O manipulador de arquivo retornado em um passo anterior é usado como parâmetro no próximo passo da operação *lookup*. Como os manipuladores de arquivo são opacos (não interpretados) para o código do cliente NFS, o sistema de arquivos virtual é responsável por traduzir o manipulador de arquivo em um diretório local, ou remoto, e por realizar o procedimento indireto necessário quando fizer referência a um ponto de montagem local. O armazenamento em cache dos resultados de cada passo diminui a aparente ineficiência desse processo tirando proveito do caráter local da referência a arquivos e diretórios; isto é, normalmente, os usuários e programas acessam arquivos em apenas um diretório ou em um pequeno número de diretórios.

Montagem automática (*automounter*) ◊ A montagem automática foi adicionada na implementação UNIX do NFS para montar dinamicamente um diretório remoto quando um ponto de montagem "vazio" fosse referenciado por um cliente. A implementação original da montagem automática era executada como um processo UNIX em nível de usuário em cada computador cliente. Versões posteriores (chamadas de *autofs*) foram implementadas no núcleo do Solaris e do Linux. Descreveremos aqui a versão original.

O *automounter* mantém uma tabela de pontos de montagem (nomes de caminho), com uma referência, para cada um deles, de um ou mais servidores NFS. Ele se comporta como um servidor NFS local na máquina cliente. Quando o módulo cliente NFS tenta solucionar um nome de caminho que inclui um desses pontos de montagem, ele faz uma requisição de *lookup*() para o *automounter* local, que localiza em sua tabela o sistema de arquivos exigido e envia um pedido de *sondagem* (*probe*) para cada servidor listado. O sistema de arquivos do primeiro servidor a responder é então montado no cliente, usando o serviço de montagem normal. O sistema de arquivos montado é associado ao ponto de montagem usando um vínculo simbólico, de modo que os acessos a ele não resultarão em mais requisições para o *automounter*. Então, o acesso ao arquivo prossegue normalmente, sem mais referências ao *automounter*, a não ser que não existam referências a esse vínculo simbólico por vários minutos. Neste caso, o *automounter* desmonta o sistema de arquivos remoto.

As implementações posteriores, no núcleo, substituíram os vínculos simbólicos por montagens reais, evitando alguns problemas que surgiam em aplicativos que armazenavam em cache os nomes de caminho temporários usados pelo *automounter* em nível de usuário [Callaghan 1999].

Uma forma simples de replicação, somente em leitura, pode ser obtida por um conjunto de vários servidores contendo cópias idênticas de um sistema de arquivos, ou de uma subárvore de arquivos, com uma única entrada na tabela do *automounter*. Isso é útil para sistemas de arquivos muito utilizados que mudam pouco, como os binários de sistema do UNIX. Por exemplo, cópias do diretório */usr/lib* e de sua subárvore poderiam ser mantidas em mais de um servidor. Na primeira ocasião em que um arquivo de */usr/lib* fosse aberto em um cliente, todos os servidores receberiam as mensagens de *probe* e o primeiro a responder seria montado no cliente. Isso proporciona um grau limitado de tolerância a falhas e balanceamento de carga, pois o primeiro servidor a responder será um que não falhou e é provavelmente um que não esteja muito ocupado com o atendimento de outras requisições.

Uso de cache no servidor ◊ O uso de cache, tanto no cliente como no servidor, é um recurso indispensável para obter um desempenho adequado nas implementações do NFS.

Nos sistemas UNIX convencionais, blocos de arquivo, diretórios e atributos de arquivo que foram lidos do disco são mantidos em *cache* na memória principal até que o espaço da cache seja exigido para outros blocos. Se um processo emite então uma requisição de leitura, ou escrita, de um bloco que já está na cache, ele pode ser atendido sem outro acesso ao disco. A *leitura antecipada* (*read-ahead*) adianta os acessos de leitura e busca os blocos seguintes àquelas lidas mais recentemente, e a *escrita postergada* (*delayed write*) otimiza as operações de escrita: quando um bloco tiver sido alterado (por uma requisição de escrita), seu novo conteúdo será gravado no disco somente quando o espaço ocupado por esse bloco na cache for exigido por outro. Para evitar a perda de dados no caso de falha do sistema, a operação *sync* do UNIX passa para o disco, a cada 30 segundos, os blocos alterados. Essas técnicas de uso de cache funcionam em um ambiente UNIX convencional porque todas as requisições de leitura e escritas feitas pelos processos em nível de usuário passam por uma única cache, que é implementada no espaço do núcleo do UNIX. A cache é sempre mantida atualizada e os acessos a arquivo devem sempre passar por ela.

Os servidores NFS usam cache exatamente como ela é empregada para outros acessos a arquivo. O uso da cache do servidor para conter blocos de disco lidos recentemente não acarreta nenhum problema de consistência, mas quando um servidor realiza operações de escritas, medidas extras são necessárias para garantir que os clientes possam ter confiança de que os resultados das operações de escrita sejam persistentes, mesmo quando ocorre uma falha do servidor. Na versão 3 do protocolo NFS, a operação *write* oferece duas opções (não mostradas na Figura 8.9):

1. Nas operações *write*, os dados recebidos dos clientes são armazenados na cache no servidor *e* gravados no disco, antes que uma resposta seja enviada para o cliente. Isso é chamado de uso de cache com *escrita direta* (*write-through*). O cliente pode ter certeza de que os dados estão armazenados de modo persistente, assim que a resposta for recebida.

2. Nas operações *write*, os dados são armazenados apenas na cache de memória. Eles serão gravados no disco quando uma operação de efetivação (*commit*) for recebida para o arquivo relevante. O cliente tem certeza de que os dados estão armazenados de modo persistente somente quando a resposta da operação *commit* for recebida. Os clientes NFS padrão usam esse modo de operação, executando uma operação *commit* quando um arquivo aberto para gravação for fechado.

Commit é uma operação adicional fornecida na versão 3 do protocolo NFS; ela foi incluída para superar o gargalo de desempenho causado pelo modo de operação *write-through* nos servidores que recebem grandes quantidades de operações *write*.

O requisito de *write-through* em sistemas de arquivos distribuídos é um caso dos modos de falha independentes, discutidos no Capítulo 1 – os clientes continuam a operar quando um servidor falha, e os programas aplicativos podem executar ações supondo que os resultados das escritas anteriores foram enviados para armazenamento em disco. É improvável que isso ocorra no caso de atualizações de arquivos locais, pois é quase certo que a falha de um sistema de arquivos local resulte na falha de todos os processos aplicativos em execução no mesmo computador.

Uso de cache no cliente ◊ O módulo cliente NFS armazena em cache os resultados das operações *read*, *write*, *getattr*, *lookup* e *readdir*, para reduzir o número de requisições feitas aos servidores. O uso de cache no cliente implica na possibilidade de existirem diversas versões de arquivos, ou porções deles, em diferentes nós clientes, pois as gravações feitas por um cliente não resultam na atualização imediata em outros clientes das cópias do mesmo arquivo armazenados em cache. Em vez disso, os clientes são responsáveis por fazerem uma consulta seqüencial no servidor para verificar se os dados armazenados em cache que eles contêm são atuais.

Um método baseado em *timestamps* é usado para validar blocos armazenados em cache antes de serem usados. Cada elemento de dados, ou de metadados, presente na cache é rotulado com duas indicações de tempo (*timestamps*):

Tc é a hora em que a entrada da cache foi validada pela última vez.
Tm é a hora em que o bloco foi modificado pela última vez no servidor.

Uma entrada da cache é válida no tempo *T*, se $T - Tc$ for menor que um intervalo de atualização *t*, ou se o valor de *Tm* gravado no cliente corresponder ao valor *Tm* presente no servidor (isto é, os dados não foram modificados no servidor desde que a entrada na cache foi feita). Formalmente, a condição de validade é:

$$(T - Tc < t) \vee (Tm_{cliente} = Tm_{servidor})$$

A escolha de um valor para *t* é um compromisso entre consistência e eficiência. Um intervalo de atualização muito curto resultará em uma forte aproximação da consistência de cópia única, ao custo de uma carga relativamente pesada de chamadas para o servidor para verificar o valor de $Tm_{servidor}$. Nos clientes Sun Solaris, *t* é configurado de forma adaptativa para arquivos individuais, com um valor no intervalo de 3 a 30 segundos, dependendo da freqüência das atualizações no arquivo. Para diretórios, o intervalo é de 30 a 60 segundos, refletindo o menor risco de atualizações concorrentes.

Existe um valor de $Tm_{servidor}$ para todos os blocos de dados de um arquivo e outro para os atributos de arquivo. Como os clientes NFS não podem determinar se um arquivo está sendo compartilhado ou não, o procedimento de validação deve ser usado para todos os acessos ao arquivo. Uma verificação de validade é realizada quando uma entrada da cache é usada. A primeira metade da condição de validade pode ser avaliada sem nenhum acesso ao servidor. Se ela for verdadeira, então a segunda metade não precisará ser avaliada; se for falsa, o valor corrente de $Tm_{servidor}$ será obtido (por meio de uma chamada de *getattr* no servidor) e comparado com o valor local $Tm_{cliente}$. Se eles forem iguais, então a entrada da cache será considerada válida, e o valor de *Tc* dessa entrada da cache será atualizado com o tempo corrente. Se eles diferirem, então os dados armazenados em cache foram atualizados no servidor e a entrada da cache será invalidada, resultando em uma requisição para o servidor fornecer os dados relevantes.

Várias medidas são usadas para reduzir o tráfego das chamadas de *getattr* para o servidor:

- Quando um novo valor de $Tm_{servidor}$ é recebido em um cliente, ele é aplicado em todas as entradas de cache derivadas do arquivo relevante.

- Os valores de atributo correntes são enviados "a tiracolo" (*piggyback*) com os resultados de cada operação em um arquivo e, se o valor de $Tm_{servidor}$ tiver mudado, o cliente o utilizará para atualizar as entradas de cache relativas ao arquivo.
- O algoritmo adaptativo para configurar o intervalo de atualização t, mencionado anteriormente, reduz consideravelmente o tráfego para a maioria dos arquivos.

O procedimento de validação não garante o mesmo nível de consistência de arquivos que é fornecido nos sistemas UNIX convencionais, pois as atualizações recentes nem sempre são visíveis para os clientes que estão compartilhando um arquivo. Existem duas fontes de atraso temporais: o atraso após uma escrita e antes que os dados atualizados sejam removidos da cache do cliente, e a janela de 3 segundos para a validação da cache. Felizmente, a maioria dos aplicativos UNIX não depende da sincronização das atualizações de arquivo e poucas dificuldades foram relatadas por causa dessa fonte.

As escritas são manipuladas de forma diferente. Quando um bloco armazenado em cache é modificado, ele é marcado como alterado (*dirty*) e programado para ser transferido para o servidor de forma assíncrona. Os blocos modificados são transferidos quando o arquivo é fechado, quando ocorre uma operação *sync* no cliente, ou ainda, mais freqüentemente, se processos *bio-daemon* estiverem em uso (veja a seguir). Isso não proporciona a mesma garantia de persistência que a cache do servidor, mas simula o comportamento das escritas locais.

Para implementar leitura antecipada (*read-ahead*) e escrita postergada (*write-delayed*), o cliente NFS precisa realizar algumas leituras e escritas de forma assíncrona. Isso é obtido nas implementações UNIX do NFS por meio da inclusão de um ou mais processos *bio-daemon* em cada cliente. (*Bio* significa entrada e saída de bloco [*block input-output*]; o termo *daemon* é freqüentemente usado para referenciar processos em nível de usuário que executam tarefas de sistema.) A função dos processos *bio-daemon* é realizar operações de leitura antecipada e escrita postergada. Após cada requisição de leitura, o bio-deamon é notificado e então ele inicia a transferência, do servidor para a cache cliente, do bloco de arquivo seguinte ao que foi lido. No caso de escrita, o *bio-daemon* envia um bloco para o servidor sempre que o bloco tiver sido completamente escrito pelo cliente. Os blocos de diretório são enviados quando ocorre uma modificação.

Os processos *bio-daemon* melhoram o desempenho do sistema garantindo que o módulo cliente não fique bloqueado esperando pelo retorno de operações *read*, nem que operações *write* sejam efetivadas no servidor. Eles não são um requisito lógico obrigatório, pois na ausência da leitura antecipada uma operação *read*, em um processo de usuário, provocará uma requisição síncrona para o servidor, e os resultados das operações *write* serão transferidos para o servidor quando o arquivo for fechado, ou quando o sistema de arquivos virtual no cliente executar uma operação *sync*.

Outras otimizações ◊ O sistema de arquivos da Sun é baseado no Fast File System do BSD UNIX, que usa blocos de disco de 8 Kbytes, resultando em menos chamadas de sistema para acessar arquivos seqüenciais do que os sistemas UNIX anteriores. Os pacotes UDP usados para a implementação da RPC Sun são ampliados para 9 Kbytes, permitindo que uma chamada de RPC, contendo um bloco inteiro como argumento, seja transferida em um único pacote, minimizando o efeito da latência da rede na leitura seqüencial de arquivos. No NFS versão 3, não há limite para o tamanho máximo de blocos de arquivo que podem ser manipulados em operações *read* e *write*; clientes e servidores podem negociar tamanhos maiores do que 8 Kbytes, se ambos puderem manipulá-los.

Conforme mencionado anteriormente, as informações de status do arquivo colocadas na cache dos clientes devem ser atualizadas pelo menos a cada três segundos para arquivos ativos. Para reduzir a conseqüente carga no servidor, resultante das requisições *getattr*, todas as operações que se referem a arquivos, ou diretórios, são consideradas como requisições *getattr* implícitos, e os valores de atributo correntes são levados "a tiracolo", junto com os outros resultados da operação.

Tornando o NFS seguro com o Kerberos ◊ Na Seção 7.6.2, descrevemos o sistema de autenticação Kerberos, desenvolvido no MIT, que se transformou em um padrão para tornar servidores de intranet seguros contra acessos não autorizados e a ataques de impostores. A segurança das implementações do NFS foi melhorada com o uso do esquema Kerberos para autenticar clientes. Nesta subseção, descreveremos a "kerberização" do NFS, conforme especificado pelos projetistas do Kerberos.

Na implementação padrão original do NFS, a identidade do usuário é incluída em cada requisição, na forma de um identificador numérico "em claro". (Nas versões posteriores do NFS, o identifi-

cador podia ser cifrado.) O NFS não faz nada para verificar a autenticidade do identificador fornecido. Isso implica, por parte do NFS, em um alto grau de confiança na integridade do computador cliente e em seu software; enquanto o objetivo do Kerberos, e de outros sistemas de segurança baseados em autenticação, é reduzir a um mínimo a gama de componentes nos quais a confiança é pressuposta. Basicamente, quando o NFS é usado em um ambiente "kerberizado", ele só deve aceitar requisições de clientes que possam mostrar que sua identidade foi autenticada pelo Kerberos.

Uma solução óbvia, considerada pelos desenvolvedores do Kerberos, foi alterar a natureza das credenciais exigidas pelo NFS para que fossem um tíquete e um autenticador completos do Kerberos. Mas como NFS é implementado como um servidor sem estado, cada requisição de acesso a arquivo é tratada isoladamente e, por conseqüência, os dados de autenticação devem ser incluídos em cada uma delas. Isso foi considerado inaceitavelmente dispendioso quanto ao tempo exigido para realizar os procedimentos de criptografia necessários e, além disso, acarretaria na adição da biblioteca de cliente Kerberos no núcleo de cada uma das estações de trabalho.

Em vez disso, foi adotada uma estratégia mista, na qual cada servidor NFS ao montar os sistemas de arquivos de usuários e raiz recebe os dados de autenticação completos do Kerberos. Os resultados dessa autenticação, incluindo o identificador numérico do usuário (*uid*) e o endereço do computador cliente, são mantidos pelo servidor com as informações de montagem para cada sistema de arquivos. (Embora o servidor NFS não mantenha o estado relacionado aos processos clientes individuais, ele mantém as montagens correntes em cada computador cliente).

Em cada requisição de acesso a arquivo, o servidor NFS verifica o identificador do usuário e o endereço do remetente, e só garante o acesso se eles corresponderem àqueles armazenados no servidor para o cliente relevante, no momento da montagem. Essa estratégia mista envolve apenas um custo adicional mínimo e é segura contra a maioria das formas de ataque, desde que apenas um usuário por vez possa se conectar em cada computador cliente. No MIT, o sistema é configurado de modo que esse seja o caso. As implementações de NFS recentes incluem autenticação do Kerberos como uma de várias opções de autenticação, e sites que também executam servidores Kerberos são aconselhados a utilizar essa opção.

Desempenho ◊ Os primeiros resultados relatados por Sandberg [1987] mostraram que o uso de NFS, normalmente, não impunha uma penalidade sobre o desempenho em comparação com o acesso aos arquivos armazenados em discos locais. Ele identificou duas áreas problemáticas remanescentes:

- uso freqüente da chamada de *getattr* para buscar indicações de tempo nos servidores para validação da cache;
- desempenho relativamente deficiente da operação *write*, porque era usada escrita direta (*write-through*) no servidor.

Ele observou que as escritas são relativamente raras na carga de trabalho normal do UNIX (cerca de 5% de todas as chamadas para o servidor) e, portanto, o custo da escrita direta é tolerável, exceto quando arquivos grandes são gravados no servidor. A versão do NFS testada não incluía o mecanismo *commit* mencionado anteriormente, e isso resultou em uma melhoria substancial no desempenho da escrita nas versões atuais. Seus resultados também mostram que a operação *lookup* é responsável por quase 50% das chamadas ao servidor. Isso é uma conseqüência do método de tradução de nomes de caminho, passo a passo, necessário à semântica de atribuição de nomes de arquivos do UNIX.

Regularmente, medidas são realizadas pela Sun, e por outros implementadores de NFS, usando uma versão atualizada de um conjunto exaustivo de programas de *benchmark*, conhecido como LADDIS [Keith e Wittle 1993]. Resultados atuais e antigos estão disponíveis no endereço [www.spec.org]. Lá, está resumido o desempenho para implementações do servidor NFS de muitos fornecedores e de diferentes configurações de hardware. Implementações com uma só CPU, baseadas em hardware de PC, mas com sistemas operacionais dedicados, atingem desempenho de mais de 12.000 operações no servidor por segundo, e configurações grandes, com multiprocessadores, vários discos e controladoras, atingirem cerca de até 300.000 operações no servidor por segundo. Esses valores indicam que o NFS oferece uma solução muito eficiente para as necessidades de armazenamento distribuído em intranets em diferentes tipos de uso, variando, por exemplo, desde uma carga de atividades em desenvolvimento, em um UNIX tradicional, executada por várias centenas de engenheiros de software, até um conjunto de servidores web recuperando páginas de um único servidor NFS.

Resumo do NFS ◊ O NFS segue de perto nosso modelo abstrato. O projeto resultante proporciona boa transparência de localização e de acesso, caso o serviço de montagem NFS seja usado corretamente para produzir espaços de nomes semelhantes em todos os clientes. O NFS suporta hardware e sistemas operacionais heterogêneos. A implementação do servidor NFS é sem estado (*stateless*), permitindo que clientes e servidores retomem a execução após uma falha, sem a necessidade de quaisquer procedimentos de recuperação. A migração de arquivos, ou de sistemas de arquivos, não é suportada, a não ser que seja feita no nível de intervenção manual para reconfigurar diretivas de montagem, após a mudança de um sistema de arquivos para um novo local.

O desempenho do NFS melhora com o uso de cache para blocos de arquivo em cada computador cliente. Isso é importante para a obtenção de um desempenho satisfatório, mas resulta em certo desvio da semântica estrita de atualização de arquivo com cópia única do UNIX.

Os outros objetivos de projeto do NFS, e até que ponto eles foram atingidos, serão discutidos a seguir.

Transparência de acesso: o módulo cliente NFS fornece uma interface de programação de aplicativos para processos locais idêntica à interface do sistema operacional local. Assim, em um cliente UNIX, os acessos a arquivos remotos são realizados usando-se as chamadas de sistema UNIX normais. Nenhuma modificação é exigida nos programas existentes, para permitir que eles funcionem corretamente com arquivos remotos.

Transparência de localização: cada cliente estabelece um espaço de nomes de arquivo, adicionando diretórios montados em sistemas de arquivos remotos em seu espaço de nomes local. Os sistemas de arquivo precisam ser *exportados* pelo nó que os contém, e *montados de forma remota* por um cliente, antes que possam ser acessados pelos processos em execução no cliente (veja a Figura 8.10). O ponto na hierarquia de nomes de um cliente em que um sistema de arquivos montado aparece é determinado pelo cliente; assim, o NFS não impõe um único espaço de nomes de arquivo em nível de rede – cada cliente vê um conjunto de sistemas de arquivos remotos de acordo com o ponto de montagem usado. Como conseqüência, sistemas de arquivos remotos podem ter diferentes nomes de caminho em diferentes clientes, mas um espaço de nomes uniforme pode ser estabelecido com tabelas de configuração apropriadas em cada cliente, atingindo o objetivo da transparência de localização.

Transparência de mobilidade: os sistemas de arquivos (no sentido do UNIX, isto é, subárvores de arquivos) podem ser movidos entre servidores, mas as tabelas de montagem em cada cliente devem ser atualizadas separadamente para permitir que os clientes acessem o sistema de arquivos em sua nova localização; portanto, a transparência da migração não é totalmente obtida pelo NFS.

Escalabilidade: os valores de desempenho publicados mostram que podem ser construídos servidores NFS para manipular cargas reais muito grandes, de maneira eficiente e econômica. O desempenho de um único servidor pode ser aumentado com a adição de processadores, discos e controladores. Quando os limites desse processo são atingidos, servidores adicionais devem ser instalados e os sistemas de arquivos devem ser realocados entre eles. A eficácia dessa estratégia é limitada pela existência de arquivos "concorridos" – aqueles que são acessados com tanta freqüência que o servidor atinge um limite de desempenho. Quando a carga ultrapassa o desempenho máximo disponível com essa estratégia, uma solução melhor pode ser oferecida por um sistema de arquivos distribuído que suporte a replicação de arquivos (como o Coda, descrito no Capítulo 15), ou por um sistema de arquivos como o AFS, que reduz o tráfego do protocolo colocando arquivos inteiros na cache. Vamos discutir outras estratégias para atingir escalabilidade, na Seção 8.5.

Replicação de arquivos: Meios de armazenamentos de arquivos somente para leitura (*read-only*) podem ser replicados em vários servidores NFS, mas o NFS não suporta replicação de arquivos com atualizações. O NIS (*Network Information Service*) da Sun é um serviço separado, disponível para uso com o NFS, que suporta a replicação de bancos de dados simples organizados como pares chave-valor (por exemplo, os arquivos de sistema do UNIX */etc/passwd* e */etc/hosts*). Ele gerencia a distribuição de atualizações e acessos a arquivos replicados com base em um modelo de replicação mestre–escravo simples (também conhecido como modelo de *cópia primária*, melhor discutido no Capítulo 15), com possibilidade de replicação parcial ou total do banco de dados em cada site. O NIS fornece um repositório compartilhado de infor-

mações de sistema que muda com pouca freqüência e não exige que as atualizações ocorram simultaneamente em todos os sites.

Heterogeneidade de hardware e de sistema operacional: o NFS foi implementado por quase todos os sistemas operacionais e plataformas de hardware disponíveis e é suportado por uma variedade de sistemas de arquivos.

Tolerância a falhas: as características sem estado e idempotente do protocolo de acesso NFS garantem que os modos de falha observados pelos clientes ao acessar arquivos remotos sejam semelhantes àqueles encontrados no acesso de arquivos locais. Quando um servidor falha, o serviço que ele fornece é suspenso até que o servidor seja reiniciado, mas uma vez que ele tenha sido reiniciado, os processos clientes em nível de usuário prosseguem a partir do ponto em que o serviço foi interrompido, sem conhecimento da falha (exceto no caso de acesso a sistemas de arquivos remotos *montados condicionalmente*). Na prática, a montagem incondicional é usada na maioria dos casos, e isso tende a impedir que os programas aplicativos tratem das falhas do servidor.
A falha de um computador cliente, ou de um processo em nível de usuário, não tem nenhum efeito sobre qualquer servidor que ele possa usar, pois os servidores não contêm nenhum estado para seus clientes.

Consistência: descrevemos o comportamento da atualização com alguns detalhes. Ele apresenta uma forte aproximação com a semântica de cópia única e atende as necessidades da maioria dos aplicativos; porém o uso de compartilhamento de arquivos por meio do NFS para comunicação, ou forte coordenação, entre processos em diferentes computadores não é recomendado.

Segurança: a necessidade de segurança no NFS surgiu com a conexão de muitas intranets com a Internet. A integração do Kerberos com o NFS foi um passo importante nesse sentido. Outros desenvolvimentos recentes incluem a opção de usar uma implementação de RPC segura (RPCSEC_GSS, documentada no RFC 2203 [Eisler *et al.* 1997]) para autenticação, privacidade e segurança dos dados transmitidos com operações de leitura e escrita. Existem muitas instalações que não implantam esses mecanismos e, portanto, são inseguras.

Eficiência: o desempenho medido de várias implementações de NFS, e sua adoção para uso em situações que geram cargas de trabalho muito pesadas, são indicações claras da eficiência com que o protocolo NFS pode ser implementado.

8.4 Estudo de caso: Andrew File System

Assim como o NFS, o AFS (*Andrew File System*) fornece acesso transparente a arquivos remotos compartilhados para programas UNIX executando em estações de trabalho. O acesso a arquivos AFS se dá por intermédio das primitivas de arquivo UNIX normais, permitindo que os programas UNIX existentes acessem arquivos AFS sem modificação, nem recompilação. O AFS é compatível com o NFS. Os servidores AFS contêm arquivos UNIX "locais", mas o sistema de arquivos nos servidores é baseado no NFS; portanto, os arquivos são referenciados, em vez de por *i-nodes*, por manipuladores de arquivo do estilo NFS permitindo serem acessados de forma remota por meio do NFS.

O AFS difere do NFS em seu projeto e implementação. As diferenças são atribuídas principalmente à identificação da escalabilidade como o objetivo de projeto mais importante. O AFS é feito para funcionar bem com números maiores de usuários ativos do que outros sistemas de arquivos distribuídos. A principal estratégia para se obter a escalabilidade é o uso de cache capazes de armazenar arquivos inteiros nos nós clientes. O AFS tem duas características de projeto incomuns:

Servir arquivos inteiros: o conteúdo inteiro de diretórios e de arquivos é transmitido para os computadores clientes pelos servidores AFS (no AFS-3, arquivos maiores do que 64 Kbytes são transferidos em porções de 64 Kbytes).

Cache de arquivos inteiros: quando uma cópia de um arquivo, ou uma porção, tiver sido transferida para um computador cliente, ela é armazenada em uma cache no disco local. A cache contém centenas dos arquivos usados mais recentemente nesse computador. Por ser em disco, a cache é

permanente, sobrevivendo às reinicializações do computador cliente. Quando possível, são usadas cópias locais dos arquivos para atender às requisições *open* dos clientes, em detrimento das cópias remotas.

Cenário ◊ Aqui está um cenário simples ilustrando o funcionamento do AFS:

- Quando um processo de usuário em um computador cliente executa uma chamada de sistema *open* para um arquivo no espaço de arquivo compartilhado e não existe uma cópia corrente do arquivo na cache local, o servidor que contém o arquivo é localizado e é enviada uma requisição de cópia do arquivo.
- A cópia é armazenada no sistema de arquivos local do UNIX, no computador cliente; então, a cópia é aberta (com *open*) e o descritor de arquivo UNIX resultante é retornado para o cliente.
- As operações *read*, *write* e outras, subseqüentes no arquivo, executadas por processos no computador cliente, são realizadas na cópia local.
- Quando o processo no cliente executa uma chamada de sistema *close*, se a cópia local tiver sido atualizada, seu conteúdo é enviado de volta para o servidor. O servidor atualiza o conteúdo do arquivo e os *timestamps* associados. A cópia que está no disco local do cliente é mantida, para o caso de ser novamente necessária, para um processo em nível de usuário na mesma estação de trabalho.

A seguir, vamos discutir o desempenho observado do AFS, mas podemos fazer aqui algumas observações e previsões gerais com base nas características de projeto descritas anteriormente:

- Para arquivos compartilhados que são raramente modificados (como aqueles que contêm o código de comandos e bibliotecas do UNIX) e para arquivos que normalmente são acessados por apenas um usuário (como a maioria dos arquivos do diretório de base de um usuário e sua subárvore), é provável que as cópias armazenadas localmente na cache permaneçam válidas por longos períodos de tempo – no primeiro caso, porque eles não são atualizados, e no segundo, porque, se eles forem atualizados, a cópia atualizada estará na cache da estação de trabalho do proprietário. Essas classes de arquivo são responsáveis pela maioria dos acessos a arquivo.
- A cache local pode usar uma proporção substancial do espaço em disco de cada estação de trabalho, digamos, 100 megabytes. Normalmente, isso é suficiente para o estabelecimento de um conjunto de trabalho dos arquivos usados por um único usuário. A provisão de armazenamento em cache suficiente para o estabelecimento de um conjunto de trabalho garante que os arquivos regularmente usados em uma determinada estação de trabalho sejam normalmente mantidos na cache até que sejam novamente necessários.
- A estratégia de projeto é baseada em algumas suposições sobre o tamanho de arquivo médio e máximo, e sobre o caráter local da referência a arquivos nos sistemas UNIX. Essas suposições são derivadas de observações de cargas de trabalho típicas do UNIX em ambientes acadêmicos e outros [Satyanarayanan 1981; Ousterhout *et al.* 1985; Floyd 1986]. As observações mais importantes são:
  - Os arquivos são pequenos; a maioria tem menos de 10 Kbytes de tamanho.
  - As operações de leitura nos arquivos são muito mais comuns do que as de escrita (cerca de seis vezes mais comuns).
  - Acesso seqüencial é comum, o acesso aleatório é raro.
  - A maioria dos arquivos é lida e escrita por apenas um usuário. Quando um arquivo é compartilhado, normalmente é apenas um usuário que o modifica.
  - Os arquivos são referenciados em momentos específicos. Se um arquivo tiver sido referenciado recentemente, existe uma grande probabilidade de que ele seja referenciado novamente em um futuro próximo.

  Essas observações foram usadas para delinear o projeto e a otimização do AFS e *não* para restringir a funcionalidade vista pelos usuários.

- O AFS funciona melhor com as classes de arquivo identificadas no primeiro ponto anterior. Há um tipo de arquivo importante que não se encaixa em nenhuma dessas classes – os bancos de dados são normalmente compartilhados por muitos usuários e, muitas vezes, são atualizados com bastante freqüência. Os projetistas do AFS excluíram explicitamente de seus objetivos de projeto o suporte para bancos de dados, dizendo que as restrições impostas por diferentes estruturas de atribuição de nomes (isto é, acesso baseado no conteúdo), a necessidade de granularidade fina para acesso aos dados, o controle de concorrência e a atomicidade das atualizações tornam difícil projetar um sistema de banco de dados distribuído que também seja um sistema de arquivos distribuído. Eles argumentam que os requisitos necessários para bancos de dados distribuídos devem ser tratados separadamente [Satyanarayanan 1989a].

8.4.1 Implementação

O cenário anterior ilustra o funcionamento do AFS, mas deixa sem resposta muitas perguntas sobre sua implementação. Dentre as mais importante estão:

- Como o AFS obtém o controle quando uma chamada de sistema *open* (ou *close*) se referindo a um arquivo no espaço de arquivos compartilhado é feito por um cliente?
- Como é localizado o servidor que contém o arquivo solicitado?
- Que espaço é alocado para arquivos em cache nas estações de trabalho?
- Como o AFS garante que as cópias dos arquivos colocadas na cache estão atualizadas quando os arquivos podem ser modificados por vários clientes?

Respondemos essas perguntas a seguir.

O AFS é implementado com base em dois componentes de software que existem como processos UNIX, em nível de usuário, chamados *Vice* e *Venus*. A Figura 8.11 mostra a distribuição dos processos Vice e Venus. Vice é o nome dado ao software servidor executado em cada computador servidor, e Venus é um processo executado em cada computador cliente e corresponde ao módulo cliente em nosso modelo abstrato.

Os arquivos disponíveis para processos de usuário em execução nas estações de trabalho são *locais* ou *compartilhados*. Os arquivos locais são tratados como arquivos normais do UNIX. Eles são armazenados em um disco da estação de trabalho e estão disponíveis somente para processos de usuário locais.

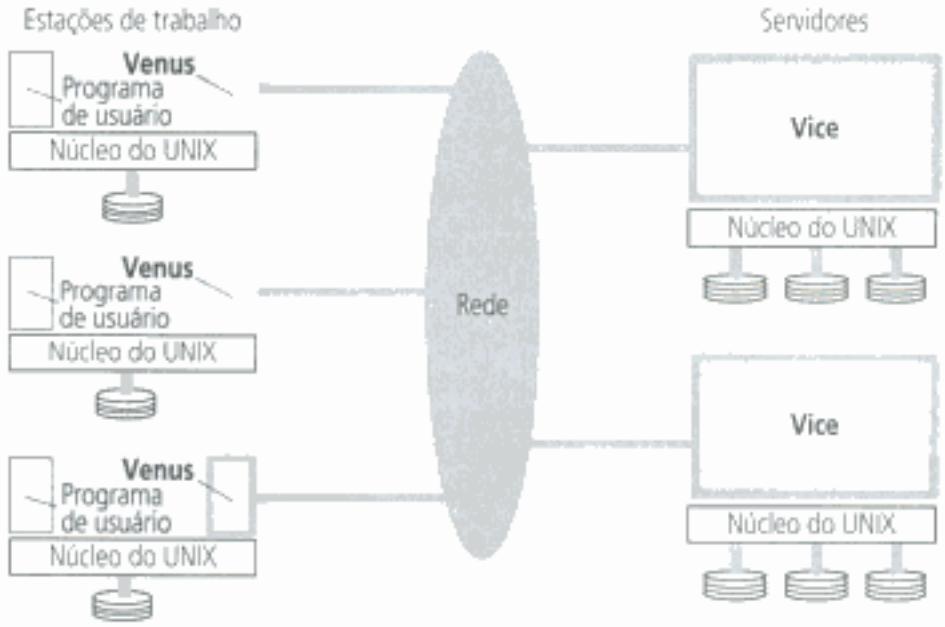


Figura 8.11 Distribuição de processos no Andrew File System.

Os arquivos compartilhados são armazenados em servidores, e cópias deles são armazenadas na cache nos discos locais das estações de trabalho. O espaço de nomes visto pelos processos de usuário está ilustrado na Figura 8.12. Trata-se de uma hierarquia de diretórios UNIX convencional, com uma subárvore específica (chamada *cmu*), contendo todos os arquivos compartilhados. Essa divisão do espaço de nomes de arquivo em arquivos locais e compartilhados leva a certa perda de transparência de localização, mas isso dificilmente é perceptível para usuários que não sejam os administradores de sistema. Os arquivos locais são usados apenas por arquivos temporários (*/tmp*) e por processos que são fundamentais para a inicialização da estação de trabalho. Outros arquivos padrão do UNIX (como aqueles normalmente encontrados em */bin*, */lib*, etc.) são implementados como vínculos simbólicos de diretórios locais para arquivos mantidos no espaço compartilhado. Os diretórios dos usuários ficam no espaço compartilhado, permitindo que os usuários acessem seus arquivos a partir de qualquer estação de trabalho.

O núcleo UNIX, em cada estação de trabalho e servidor, é uma versão modificada do UNIX BSD. As modificações são feitas para interceptar chamadas de sistema de arquivos *open*, *close* e algumas outras, quando elas se referem a arquivos do espaço de nomes compartilhado e as passam para o processo Venus no computador cliente (ilustrado na Figura 8.13). Uma outra modificação do núcleo é incluída por motivos de desempenho e ela será descrita posteriormente.

Uma das partições do disco local de cada estação de trabalho é usada como a cache que conterá as cópias dos arquivos do espaço compartilhado. Venus gerencia a cache. Se, ao obter um novo arquivo a partir do servidor a partição de cache estiver cheia, Venus remove os arquivos usados menos recentemente para abrir espaço. Normalmente, a cache da estação de trabalho é grande o suficiente para acomodar várias centenas de arquivos de tamanho médio, tornando a estação de trabalho amplamente independente dos servidores Vice, uma vez que um conjunto dos arquivos de trabalho do usuário corrente e dos arquivos de sistema usados freqüentemente foram colocados na cache.

O AFS é semelhante ao modelo de serviço de arquivos abstrato, descrito na Seção 8.2, sob os seguintes aspectos:

- Um serviço de arquivos simples é implementado pelos servidores Vice, e a estrutura de diretório hierárquica exigida pelos programas de usuário do UNIX é implementada pelo conjunto de processos Venus nas estações de trabalho.
- Cada arquivo e diretório no espaço de arquivo compartilhado é identificado por um identificador de arquivo (*fid – file identifier*) único de 96 bits, semelhante a um UFID. Os processos Venus traduzem os nomes de caminho, fornecidos pelos clientes, em *fids*.

Os arquivos são agrupados em *volumes* para facilitar a localização e a movimentação. Geralmente, os volumes são menores do que os sistemas de arquivos UNIX, que são a unidade de agrupamento

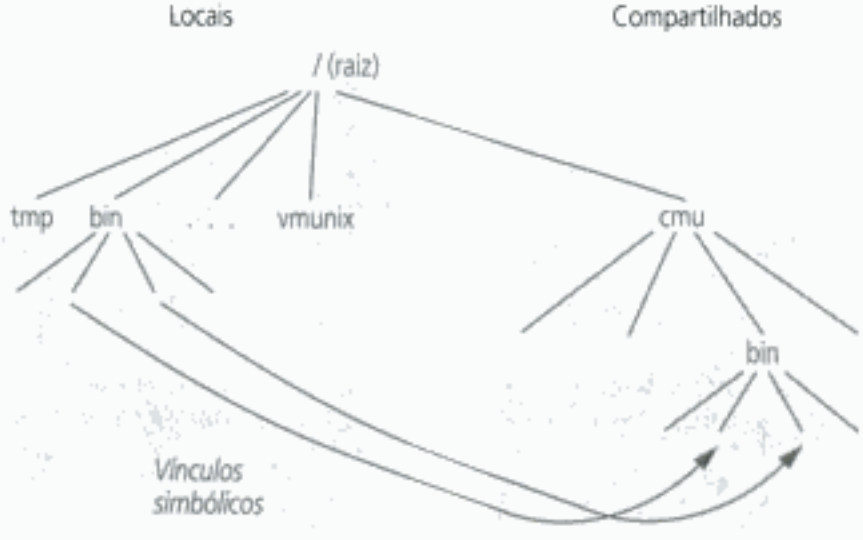


Figura 8.12 Espaço de nomes de arquivo visto pelos clientes do AFS.

Figura 8.13 Interceptação de chamada de sistema no AFS.

de arquivos do NFS. Por exemplo, os arquivos pessoais de cada usuário geralmente estão localizados em um volume separado. Outros volumes são alocados para os binários do sistema, documentação e códigos de bibliotecas.

A representação dos *fid*s inclui o número do volume que contém o arquivo (compare com o *identificador de grupo de arquivos* nos UFIDs), um manipulador de arquivo NFS identificando o arquivo dentro do volume (conforme o *número de arquivo* nos UFIDs) e um *elemento de exclusividade* (*uniquifier*) para garantir que os identificadores de arquivo não sejam reutilizados:

32 bits	*32 bits*	*32 bits*
Número do volume	**Manipulador de arquivo**	**Elemento de exclusividade**

Os programas de usuário utilizam nomes de caminho convencionais do UNIX para referenciar arquivos, mas o AFS usa *fid*s na comunicação entre os processos Venus e Vice. Os servidores Vice aceitam requisições apenas em termos de *fid*s. O Venus traduz os nomes de caminho fornecidos pelos clientes em *fid*s, usando uma pesquisa passo a passo para obter as informações dos diretórios de arquivo mantidos nos servidores Vice.

A Figura 8.14 descreve as ações executadas pelo Vice, pelo Venus e pelo núcleo do UNIX, quando um processo de usuário executa cada uma das chamadas de sistema mencionadas em nosso cenário delineado anteriormente. A *promessa de callback* mencionada aqui é um mecanismo para garantir que as cópias dos arquivos armazenadas em cache sejam atualizadas quando outro cliente fechar o mesmo arquivo após modificá-lo. Esse mecanismo será discutido na próxima seção.

8.4.2 Consistência da cache

Quando o Vice fornece uma cópia de um arquivo para um processo Venus, ele também fornece uma *promessa de callback* – um *token* emitido pelo servidor Vice que é o depositário do arquivo, garantindo que notificará o processo Venus quando qualquer outro cliente modificar o arquivo. As promessas de *callback* são armazenadas junto com os arquivos na cache em disco da estação de trabalho e têm dois estados: *válida* ou *cancelada*. Quando um servidor realiza uma requisição de atualização de um arquivo, ele notifica todos os processos Venus para os quais tem promessas de *callback* emitidas, enviando um *callback* para cada um – um *callback* é uma chamada de procedimento remoto de um servidor para um processo Venus. Quando o processo Venus recebe um *callback*, ele configura o *token* de *promessa de callback* do arquivo em questão como *cancelado*.

Quando Venus trata uma operação *open* em nome de um cliente, ele verifica a cache. Se o arquivo solicitado for encontrado na cache, então seu *token* é verificado. Se o valor for *cancelado*, então uma

Processo de usuário	*Núcleo do UNIX*	*Venus*	*Rede*	*Vice*
open(FileName, mode)	Se FileName se refere a um arquivo no espaço de arquivos compartilhado, passa a requisição para Venus. Abre o arquivo local e retorna o descritor de arquivo para o aplicativo.	Verifica a lista de arquivos na cache local. Se não estiver presente, ou não houver uma *promessa de callback* válida, envia uma requisição do arquivo para o servidor Vice que é o depositário do volume que contém o arquivo. Coloca a cópia do arquivo no sistema de arquivos local, insere seu nome local na lista da cache e retorna o nome local para o UNIX.	→ ←	Transfere uma cópia do arquivo e uma *promessa de callback* para a estação de trabalho. Registra a promessa de callback.
read(FileDescriptor, Buffer, length)	Executa uma operação de leitura UNIX normal na cópia local.			
write(FileDescriptor, Buffer, length)	Executa uma operação de escrita UNIX normal na cópia local.			
close(FileDescriptor)	Fecha a cópia local e notifica Venus de que o arquivo foi fechado.	Se a cópia local tiver sido alterada, envia uma cópia para o servidor Vice que é o depositário do arquivo.	→	Substitui o conteúdo do arquivo e envia um *callback* para todos os outros clientes que contêm *promessas de callback* no arquivo.

Figura 8.14 Implementação de chamadas de sistema de arquivos no AFS.

cópia nova do arquivo deve ser buscada no servidor Vice, mas se o *token* for *válido*, então a cópia armazenada na cache poderá ser aberta e usada, sem referência ao Vice.

Quando uma estação de trabalho é reiniciada após uma falha, ou desligamento, Venus tenta manter o máximo possível dos arquivos armazenados na cache no disco local, mas não pode presumir que os *tokens* de promessa de *callback* estejam corretos, pois alguns *callbacks* podem ter sido perdidos. Portanto, antes do primeiro uso de cada arquivo, ou diretório, armazenados na cache, após uma reinicialização, Venus gera uma requisição de validação de cache contendo o *timestamp* da modificação do arquivo para o servidor que é o depositário do arquivo. Se o *timestamp* estiver atualizado, o servidor responderá com *válido* e o *token* será reinstalado. Se o *timestamp* mostrar que o arquivo está desatualizado, então o servidor responderá com *cancelado* e o *token* será configurado como *cancelado*. Os *callbacks* devem ser renovados antes de uma operação *open*, caso um tempo T (normalmente, da ordem de alguns minutos) tenha decorrido desde que o arquivo foi posto na cache sem ter havido comunicação do servidor. Isso serve para tratar de possíveis falhas de comunicação, as quais podem resultar na perda de mensagens de *callback*.

Esse mecanismo baseado em *callback* para manutenção da consistência da cache foi adotado como a estratégia que oferecia maior escalabilidade, de acordo com a avaliação no protótipo (AFS-1) de um mecanismo baseado em *timestamp* semelhante àquele usado no NFS. No AFS-1, um processo Venus, contendo uma cópia de um arquivo na cache, interroga o processo Vice em cada operação *open* para determinar se a indicação de tempo na cópia local está de acordo com a do servidor. A estratégia baseada em *callback* tem maior escalabilidade porque resulta na comunicação entre cliente e servidor e na atividade do servidor somente quando o arquivo tiver sido atualizado, enquanto a estratégia de

timestamp causa uma interação cliente-servidor em cada operação *open*, mesmo quando existe uma cópia local válida. Como a maioria dos arquivos não é acessada concorrentemente, e as operações *read* predominam sobre as operações *write* na maioria dos aplicativos, o mecanismo de *callback* causa uma importante redução no número de interações cliente-servidor.

O mecanismo de *callback* usado no AFS-2, e em versões posteriores do AFS, exige que os servidores Vice mantenham algum estado em nome de seus clientes Venus, ao contrário do que ocorre no AFS-1, no NFS e em nosso modelo de serviço de arquivos. O estado dependente do cliente exigido consiste em uma lista dos processos Venus para os quais foram emitidas promessas de *callback* para cada arquivo. Essas listas de *callback* devem ser mantidas no caso de falhas do servidor – elas são mantidas nos discos do servidor e atualizadas com operações atômicas.

A Figura 8.15 mostra as chamadas de RPC fornecidas pelos servidores AFS para operações em arquivos (isto é, a interface fornecida pelos servidores AFS para processos Venus).

Semântica de atualização ◊ O objetivo desse mecanismo de consistência de cache é obter a melhor aproximação possível da semântica de arquivo de cópia única, sem uma degradação séria de desempenho. Uma implementação estrita da semântica de cópia única, para primitivas de acesso a arquivo do UNIX, exigiria que os resultados de cada operação *write* em um arquivo fossem distribuídos para todos os sites que o contêm em suas caches, antes que mais acessos pudessem ocorrer. Isso não pode ser feito em sistemas de larga escala; em vez disso, o mecanismo de promessa de *callback* mantém uma aproximação da semântica de cópia única.

Para o AFS-1, a semântica de atualização pode ser declarada formalmente em termos muito simples. Para um cliente *C* operando em um arquivo *F*, cujo depositário é um servidor *S*, as seguintes garantias de atualização das cópias de *F* são mantidas:

após uma operação *open* bem-sucedida:	*latest(F, S)*
após uma operação *open* mal sucedida:	*failure(S)*
após uma operação *close* bem-sucedida:	*updated(F, S)*
após uma operação *close* mal sucedida:	*failure(S)*

Onde *latest(F, S)* denota uma garantia de que o valor corrente de *F*, em *C*, é igual ao valor em *S*, *failure(S)* denota que a operação *open* (ou *close*) não foi efetuada em *S* (e a falha pode ser detectada por *C*) e *updated(F, S)* denota que o valor de *C* de *F* foi propagado com êxito para *S*.

Para o AFS-2, a garantia de atualização de *open* é ligeiramente mais fraca, e a declaração formal da garantia correspondente é mais complexa. Isso porque um cliente pode abrir uma cópia antiga de

Fetch(fid) → attr, data	Retorna os atributos (status) e, opcionalmente, o conteúdo do arquivo identificado pelo *fid* e grava nele uma promessa de *callback*.
Store(fid, attr, data)	Atualiza os atributos e (opcionalmente) o conteúdo de um arquivo especificado.
Create() → fid	Cria um novo arquivo e registra nele uma promessa de *callback*.
Remove(fid)	Exclui o arquivo especificado.
SetLock(fid, mode)	Estabelece uma trava (*lock*) no arquivo ou diretório especificado. O modo da trava pode ser compartilhado ou exclusivo. As travas que não são removidas expiram após 30 minutos.
ReleaseLock(fid)	Destrava o arquivo ou diretório especificado.
RemoveCallback(fid)	Informa o servidor que um processo Venus removeu um arquivo de sua cache.
BreakCallback(fid)	Esta chamada é feita por um servidor Vice para um processo Venus. Ela cancela a promessa de *callback* no arquivo identificado por *fid*.

Figura 8.15 Os principais componentes da interface do serviço Vice.
Nota: As operações de diretório e administrativas (*Rename, Link, Makedir, Removedir, GetTime, CheckToken*, etc.) não são mostradas.

um arquivo após ele ter sido atualizado por outro cliente. Isso ocorre se uma mensagem de *callback* é perdida, por exemplo, como resultado de uma falha da rede. Mas existe um tempo máximo *T* durante o qual um cliente pode permanecer sem saber de uma versão mais recente de um arquivo. Portanto, temos a seguinte garantia:

após uma operação *open* bem-sucedida: *latest(F, S, 0)*
ou (lostCallback(S, T) e inCache(F) e
latest(F, S, T))

Onde *latest(F, S, T)* denota que a cópia de *F* vista pelo cliente não está há mais do que *T* segundos desatualizada, *lostCallback(S, T)* denota que uma mensagem de *callback* de *S* para *C* foi perdida em algum momento durante os últimos *T* segundos e *inCache(F)* que o arquivo *F* estava na cache em *C* antes que a operação *open* fosse tentada. A declaração formal anterior expressa o fato de que a cópia colocada na cache de *F* em *C*, após uma operação *open*, é a versão mais recente presente no sistema, ou que uma mensagem de *callback* foi perdida (devido a uma falha de comunicação) e a versão que já estava na cache foi utilizada. A versão colocada na cache não estará há mais do que *T* segundos desatualizada (*T* é uma constante do sistema representando o intervalo em que as promessas de *callback* devem ser renovadas. Na maioria das instalações, o valor de *T* é configurado em cerca de 10 minutos.)

De acordo com seu objetivo – fornecer um serviço de arquivo distribuído de larga escala, compatível com o UNIX – o AFS não fornece mais nenhum mecanismo para o controle de atualizações concorrentes. O algoritmo de consistência de cache descrito anteriormente entra em ação somente nas operações *open* e *close*. Uma vez que um arquivo tiver sido aberto, o cliente pode acessar e atualizar a cópia local da maneira que quiser, sem o conhecimento de quaisquer processos em outras estações de trabalho. Quando o arquivo é fechado, uma cópia é retornada para o servidor substituindo a versão corrente.

Se clientes, em diferentes estações de trabalho, abrem, gravam e fecham (com as operações *open*, *write* e *close*) o mesmo arquivo concorrentemente, todas as atualizações, menos a resultante da última operação *close*, serão perdidas silenciosamente (nenhum relatório de erro é fornecido). Os clientes devem implementar o controle de concorrência independentemente, caso exijam isso. Por outro lado, quando dois processos clientes na mesma estação de trabalho abrem um arquivo, eles compartilham a mesma cópia colocada na cache, e as atualizações são realizadas da maneira normal do UNIX – bloco por bloco.

Embora a semântica de atualização seja diferente, dependendo das localizações dos processos concorrentes que estão acessando um arquivo, e não seja precisamente igual àquela fornecida pelo sistema de arquivos padrão do UNIX, elas são suficientemente parecidas para que a maioria dos programas UNIX existentes funcione corretamente.

8.4.3 Outros aspectos

Modificações no núcleo do UNIX ◊ Notamos que o servidor Vice é um processo em nível de usuário executando no computador servidor e o *host* servidor é dedicado ao estabelecimento de um serviço AFS. O núcleo do UNIX nos *hosts* AFS é alterado para que o Vice possa efetuar operações de arquivo em termos de manipuladores de arquivo, em vez dos descritores de arquivo convencionais do UNIX. Essa é a única modificação no núcleo exigida pelo AFS e é necessária se o Vice não precisar manter o estado do cliente (como os descritores de arquivo).

Banco de dados de localização ◊ Cada servidor contém uma cópia de um banco de dados de localização totalmente replicado, fornecendo um mapeamento de nomes de volume para servidores. Quando um volume é movido, podem ocorrer imprecisões temporárias nesse banco de dados, mas elas são inofensivas, pois o encaminhamento de informações é deixado para trás, no servidor a partir do qual o volume é movido.

Threads ◊ As implementações de Vice e Venus utilizam uma biblioteca de *threads* não-preemptiva para permitir que as requisições sejam processadas concorrentemente no cliente (onde vários processos de usuário podem ter requisições de acesso a arquivo em andamento concorrentemente) e no servidor. No cliente, as tabelas que descrevem o conteúdo da cache e o banco de dados de volumes são mantidos na memória compartilhada entre as *threads* do Venus.

Réplicas somente de leitura ◊ Os volumes que contêm arquivos lidos freqüentemente, mas raramente modificados, como os diretórios de comandos de sistema UNIX */bin* e */usr/bin* e o diretório de páginas de manual */man*, podem ser replicados como volumes somente de leitura em vários servidores. Quando isso é feito, existe apenas uma réplica para leitura e escrita, e todas as atualizações são direcionadas a ela. A propagação das alterações nas réplicas somente de leitura é realizada após a atualização por parte de um procedimento operacional explícito. As entradas no banco de dados de localização para volumes replicados dessa maneira são do tipo uma para muitas, e o servidor para cada requisição de cliente é selecionado de acordo com as cargas e a acessibilidade do servidor.

Transferências de grandes volumes ◊ O AFS transfere arquivos entre clientes e servidores em porções de 64 Kbytes. O uso de um pacote desse tamanho é uma ajuda importante para o desempenho, minimizando o efeito da latência da rede. Assim, o projeto do AFS permite que o uso da rede seja otimizado.

Uso de cache parcial de arquivo ◊ A necessidade de transferir o conteúdo inteiro dos arquivos para os clientes, mesmo quando o requisito do aplicativo é ler apenas uma pequena parte do arquivo, é uma fonte de ineficiência óbvia. A versão 3 do AFS eliminou esse requisito, permitindo que os dados dos arquivos sejam transferidos e colocados na cache em blocos de 64 Kbytes, enquanto ainda mantém a semântica de consistência e outras características do protocolo AFS.

Desempenho ◊ O principal objetivo do AFS é a escalabilidade; portanto, seu desempenho com grandes números de usuários tem particular interesse. Howard *et al.* [1988] fornecem detalhes de extensivas medidas de desempenho, que foram feitas usando um *benchmark* especialmente desenvolvido para o AFS, que posteriormente foi usado para a avaliação de sistemas de arquivos distribuídos. Evidentemente, o uso de cache para o arquivo inteiro e o protocolo de *callback* levaram a cargas substancialmente menores nos servidores. Satyanarayanan [1989a] informa que foi medida uma carga de servidor de 40%, com 18 nós de cliente, executando um *benchmark* padrão, contra uma carga de 100% para o NFS executando o mesmo *benchmark*. Satyanarayanan atribui grande parte da vantagem do desempenho do AFS à redução na carga do servidor derivada do uso de *callbacks* para notificar os clientes sobre atualizações nos arquivos, se comparada ao mecanismo de *timestamps* usado no NFS para verificar a validade das páginas colocadas na cache nos clientes.

Suporte remoto ◊ A versão 3 do AFS suporta várias células administrativas, cada uma com seus próprios servidores, clientes, administradores de sistema e usuários. Cada célula é um ambiente completamente autônomo, mas uma aliança de células pode cooperar para apresentar aos usuários um espaço de nomes de arquivo uniforme e transparente. O sistema resultante foi amplamente implantado pela Transarc Corporation e foi publicado um levantamento detalhado dos padrões de utilização e desempenho resultantes [Spasojevic e Satyanarayanan 1996]. O sistema foi instalado em mais de 1000 servidores, em mais de 150 sites. O levantamento mostrou taxas de utilização da cache no intervalo de 96–98% para acessos a uma amostra de 32.000 volumes de arquivo, contendo 200 Gbytes de dados.

8.5 Aprimoramentos e mais desenvolvimentos

Vários avanços foram feitos no projeto de sistemas de arquivos distribuídos desde a aparição do NFS e do AFS. Nesta seção, descreveremos os avanços que melhoram o desempenho, a disponibilidade e a escalabilidade dos sistemas de arquivos distribuídos convencionais. Avanços mais radicais são descritos em outras partes deste livro, incluindo a manutenção da consistência em sistemas de arquivos de leitura e escrita replicados para suportar operação desconectada e alta disponibilidade, nos sistemas Bayou e Coda (seções 14.4.2 e 14.4.3), e uma arquitetura com grande escalabilidade para a implantação de fluxos de dados em tempo real com garantia de qualidade, no servidor de arquivos de vídeo Tiger (Seção 17.6).

Melhorias no NFS ◊ Vários projetos de pesquisa trataram da questão da semântica de atualização de cópia única, ampliando o protocolo NFS para incluir as operações *open* e *close* e adicionar um mecanismo de *callback* para permitir que o servidor notifique os clientes sobre a necessidade de

invalidar entradas da cache. Descreveremos aqui dois desses trabalhos; os resultados deles parecem indicar que esses aprimoramentos podem ser acomodados sem complexidade indesejada ou custos de comunicação extras.

Alguns trabalhos recentes da Sun, e de outros desenvolvedores de NFS, foram direcionados de forma a tornar os servidores NFS mais acessíveis e úteis em redes de longa distância. Embora o protocolo HTTP, suportado pelos servidores web, ofereça um método eficiente e com alta escalabilidade para tornar arquivos inteiros disponíveis para os clientes por meio da Internet, ele é menos útil para programas aplicativos que exigem acesso a partes de arquivos grandes, ou para aqueles que atualizam partes de arquivos. O desenvolvimento do WebNFS (descrito a seguir) possibilita que programas aplicativos se tornem clientes de servidores NFS, em qualquer parte da Internet (usando o protocolo NFS diretamente, em vez de usar indiretamente, por meio de um módulo do núcleo). Isso, junto com bibliotecas apropriadas da linguagem Java, e de outras linguagens de programação de rede, deve oferecer a possibilidade de implementar aplicativos de Internet que compartilhem dados diretamente, como jogos para vários usuários ou clientes de grandes bancos de dados dinâmicos.

Obtendo a semântica de atualização de cópia única: a arquitetura de servidor sem estado do NFS trouxe grandes vantagens na robusteza e na facilidade de implementação do NFS, mas impossibilita a obtenção de uma semântica de atualização de cópia única precisa (não há garantia de que os efeitos de gravações concorrentes feitas por diferentes clientes no mesmo arquivo sejam os mesmos de um sistema UNIX único, quando vários processos gravam em um arquivo local). Ela também impede o uso de *callbacks* notificando os clientes sobre alterações em arquivos, e isso resulta em freqüentes requisições de *getattr* dos clientes para verificar a existência de modificação no arquivo.

Dois sistemas de pesquisa foram desenvolvidos para tratar desses inconvenientes. O Spritely NFS [Srinivasan e Mogul 1989, Mogul 1994] é uma evolução do sistema de arquivos desenvolvido na Berkeley para o sistema operacional distribuído Sprite [Nelson *et al.* 1988]. O Spritely NFS é uma implementação do protocolo NFS com a inclusão de chamadas *open* e *close*. Os módulos dos clientes devem enviar uma operação *open* quando um processo local em nível de usuário abre um arquivo que está no servidor. Os parâmetros da operação *open* do Sprite especificam um modo (leitura, gravação ou ambos) e incluem contagens do número de processos locais que têm o arquivo aberto correntemente para leitura e para escrita. Analogamente, quando um processo local fecha um arquivo remoto, uma operação *close* é enviada para o servidor, com contagens atualizadas de leitores e gravadores. O servidor registra esses números em uma *tabela de arquivos abertos*, com o endereço IP e o número de porta do cliente.

Quando o servidor recebe uma operação *open*, ele verifica a *tabela de arquivos abertos* de outros clientes que tem o mesmo arquivo aberto e envia mensagens de *callback* para esses clientes, instruindo-os a modificar suas estratégias de uso de cache. Se a operação *open* especificar modo de escrita, então ela falhará, caso qualquer outro cliente tenha o arquivo aberto para escrita. Outros clientes que tiverem o arquivo aberto para leitura serão instruídos a invalidar as partes do arquivo colocadas na cache de forma local.

Para operações *open* que especificam o modo de leitura, o servidor envia uma mensagem de *callback* para todo cliente que esteja escrevendo, instruindo-o a parar de usar a cache (isto é, para usar um modo de operação rigorosamente de escrita direta – *write-through*), e instrui a todos os clientes que estão lendo para que parem de colocar o arquivo na cache (para que todas as chamadas de leitura locais resultem em uma requisição para o servidor).

Essas medidas resultam em um serviço de arquivo que mantém a semântica de atualização de cópia única do UNIX às custas do transporte de algum estado relacionado ao cliente no servidor. Elas também permitem alguns ganhos de eficiência no tratamento de escritas na cache. Se o estado relacionado ao cliente for mantido em memória volátil no servidor, ele será vulnerável às falhas do servidor. O Spritely NFS implementa um protocolo de recuperação que interroga uma lista de clientes que abriram arquivos recentemente no servidor, para recuperar a *tabela de arquivos abertos* inteira. A lista de clientes é armazenada no disco, sendo atualizada de forma relativamente rara, e é pessimista – ela pode incluir, com segurança, mais clientes do que aqueles que tinham arquivos abertos no momento da falha. Clientes defeituosos também podem resultar em excesso de entradas na *tabela de arquivos abertos*, mas elas serão removidas quando o cliente reiniciar.

Quando o Spritely NFS foi avaliado com relação ao NFS, versão 2, apresentou uma melhoria de desempenho modesta. Isso se deu graças ao uso melhorado da escrita em cache. Mudanças no NFS, versão 3, provavelmente resultariam pelo menos em uma melhoria igual, mas os resultados do projeto Spritely NFS certamente indicam que é possível obter semântica de atualização de cópia única sem perda de desempenho substancial, embora às custas de alguma complexidade de implementação extra nos módulos cliente e servidor e da necessidade de um mecanismo de recuperação para restaurar o estado após uma falha de servidor.

NQNFS: o projeto NQNFS (Not Quite NFS) [Macklem 1994] tinha objetivos semelhantes ao Spritely NFS – adicionar consistência de cache mais precisa ao protocolo NFS e melhorar o desempenho por intermédio do uso melhor da cache. Um servidor NQNFS mantém estado similar relacionado ao cliente, a respeito de arquivos abertos, mas utiliza arrendamentos (Seção 5.2.6) para ajudar na recuperação, após uma falha de servidor. O servidor estabelece um limite superior para o tempo durante o qual um cliente pode manter um arrendamento para um arquivo aberto. Se o cliente quiser continuar além desse tempo, precisará renovar o arrendamento. *Callbacks* são usadas de maneira semelhante ao Spritely NFS, para pedir aos clientes para que invalidem suas caches quando ocorrer uma requisição de escrita, mas se os clientes não responderem, o servidor simplesmente esperará até que seus arrendamentos expirem, antes de responder a nova requisição de gravação.

WebNFS: o advento da web e dos *applets* Java levou ao reconhecimento por parte da equipe de desenvolvimento do NFS e outros, que alguns aplicativos de Internet poderiam tirar proveito do acesso direto aos servidores NFS sem grande parte das sobrecargas associadas à simulação das operações de arquivo do UNIX incluídas nos clientes NFS padrão.

O objetivo do WebNFS (descrito nos RFCs 2055 e 2056 [Callaghan 1996a, 1996b]) é permitir que navegadores web, programas Java e outros aplicativos interajam diretamente com um servidor NFS para acessar arquivos que são publicados, usando um *manipulador de arquivo público* para acessar arquivos relativos a um diretório raiz público. Esse modo de uso anula o serviço *mount* e o serviço mapeador de porta (descritos no Capítulo 5). Os clientes WebNFS interagem com um servidor NFS em um número de porta conhecido (2049). Para acessar arquivos pelo nome de caminho, eles emitem requisições de *lookup* usando um manipulador de arquivo público. O manipulador de arquivo público tem um valor conhecido que é interpretado de forma especial pelo sistema de arquivos virtual no servidor. Devido à alta latência das redes de longa distância, é usada uma variante da operação *lookup* para pesquisar um nome de caminho de várias partes em um único pedido.

Assim, o WebNFS permite escrever clientes que acessam partes de arquivos armazenados em servidores NFS de sites remotos com sobrecargas de configuração mínimas. Há suporte para controle de acesso e autenticação, mas em muitos casos o cliente solicitará apenas acesso de leitura para arquivos públicos e, nesse caso, a opção de autenticação poderá ser desativada. Ler uma parte de um único arquivo localizado em um servidor NFS que suporta WebNFS exige o estabelecimento de uma conexão TCP e duas chamadas de RPC – uma para a operação *lookup* de várias partes e outra para a operação *read*. O tamanho do bloco de dados lido não é limitado pelo protocolo NFS.

Por exemplo, um serviço de previsão de tempo poderia publicar um arquivo em seu servidor NFS, contendo um grande banco de dados de dados climáticos atualizados freqüentemente, com um URL do tipo:

nfs://data.weather.gov/weatherdata/global.data

Um cliente WeatherMap interativo, que mostrasse mapas do clima, poderia ser construído em Java, ou em qualquer outra linguagem, que ofereça suporte a biblioteca de procedimentos WebNFS. O cliente lê apenas as partes do arquivo /weatherdata/global.data que são necessárias para construir os mapas específicos solicitados por um usuário, enquanto um aplicativo semelhante, que usasse HTTP para acessar um servidor de dados climáticos, teria que transferir o banco de dados inteiro para o cliente, ou solicitar o suporte de um programa servidor de propósito específico para fornecer os dados solicitados.

NFS versão 4: uma nova versão do protocolo NFS estava em desenvolvimento no momento da publicação deste livro. Os objetivos do NFS versão 4 estão descritos no RFC 2624 [Shepler 1999] e no livro de Brent Callaghan [Callaghan 1999]. Assim como o WebNFS, ele pretende facilitar o uso do NFS em redes de longa distância e em aplicativos de Internet. Ele incluirá os recursos do WebNFS,

mas a introdução de um novo protocolo também apresenta a oportunidade de fazer aprimoramentos mais radicais. (O WebNFS ficou restrito às alterações no servidor que não envolviam essa adição de novas operações no protocolo.)

O grupo de trabalho que está desenvolvendo o NFS versão 4 pretende explorar os resultados que surgiram das pesquisas feitas no projeto do servidor de arquivos na última década, como o uso de *callbacks* ou de arrendamentos para manter a consistência. O NFS versão 4 suportará recuperação dinâmica de falhas do servidor, permitindo que os sistemas de arquivos sejam movidos para novos servidores de forma transparente. A capacidade de mudança de escala será aprimorada, usando servidores *proxies* de uma maneira análoga ao seu uso na web.

Aprimoramentos do AFS ◊ Mencionamos que o DCE/DFS, o sistema de arquivos distribuído incluído no *Distributed Computing Environment* da Open Software Foundation [www.opengroup.org], era baseado no Andrew File System. O projeto do DCE/DFS vai além do AFS, particularmente em sua estratégia para consistência da cache. No AFS, os *callbacks* são gerados apenas quando o servidor recebe uma operação *close* para um arquivo que foi atualizado. O DFS adotou uma estratégia semelhante para o Spritely NFS e para o NQNFS, isto é, gera *callbacks* assim que um arquivo é atualizado. Para atualizar um arquivo, um cliente deve obter um *token write* do servidor, especificando um intervalo de bytes que o cliente pode atualizar no arquivo. Quando um *token write* é solicitado, os clientes que contêm cópias do mesmo arquivo para leitura, recebem *callbacks* de revogação. *Tokens* de outros tipos são usados para obter consistência de atributos de arquivo colocados na cache e outros metadados. Todos os *tokens* têm um tempo de vida associado e os clientes devem renová-los após sua expiração.

Aprimoramentos na organização do armazenamento ◊ Houve um progresso considerável na organização dos dados de arquivos armazenados em discos. O impulso para grande parte desse trabalho surgiu das cargas maiores e da maior confiabilidade que os sistemas de arquivos distribuídos precisam suportar, e ele resultou em sistemas de arquivos com desempenho substancialmente melhor. Os principais resultados desse trabalho são:

RAID (Redundant Arrays of Inexpensive Disks): trata-se de um modo de armazenamento [Patterson *et al.* 1988, Chen *et al.* 1994] no qual blocos de dados são segmentados em porções de tamanho fixo e armazenados em "tiras" de vários discos (*striping*), junto com códigos redundantes de correção de erro, que permitem aos blocos de dados serem completamente reconstruídos e, no caso de falhas de disco, as operações de leitura e escrita continuarem normalmente. O RAID também produz um desempenho consideravelmente melhor do que um único disco, pois as tiras (*strips*) que constituem um bloco são lidas e gravadas concorrentemente.

LFS (Log-structured file storage): assim como o Spritely NFS, esta técnica foi originada no projeto do sistema operacional distribuído Sprite, em Berkeley [Rosenblum e Ousterhout 1992]. Os autores observaram que quando grandes quantidades de memória principal eram empregadas como cache em servidores de arquivos, melhor era o desempenho de leitura, porém o das operações de escrita continuava medíocre. Isso era proveniente das altas latências associadas à escrita de blocos de dados individuais no disco e às atualizações associadas nos blocos de metadados (isto é, os blocos conhecidos como *i-nodes*, que contêm atributos de arquivo e ponteiros para os blocos em um arquivo).

A solução do LFS é acumular um conjunto de gravações na memória e então efetivá-las no disco em grandes segmentos, adjacentes e de tamanho fixo. Eles são chamados de *segmentos de log* porque os blocos de dados e metadados são armazenados rigorosamente na ordem em que foram atualizados. Um segmento de *log* tem 1 Mbyte, ou mais, de tamanho e é armazenado em uma única trilha do disco, eliminando as latências do cabeçote do disco associadas à escrita de blocos individuais. Os blocos de dados e metadados, novos ou atualizados, são sempre escritos no disco exigindo a manutenção de um mapa dinâmico (na memória, com um *backup* persistente) apontando para os blocos de *i-nodes*. Também é exigida a coleta de lixo de blocos antigos, com compactação de blocos "ativos", para deixar áreas de armazenamento adjacentes livres para o armazenamento de segmentos de *log*. Este último é um processo bastante complexo; ele é executado em *background* por um componente chamado *limpador* (*cleaner*). Alguns algoritmos limpadores sofisticados foram desenvolvidos para isso, com base nos resultados de simulações.

Apesar desses custos extras, o ganho de desempenho geral é significativo; Rosenblum e Ousterhout mediram um desempenho de escrita de até 70% da largura de banda disponível no disco, comparado com menos de 10% para um sistema de arquivos UNIX convencional. A estrutura do *log* também simplifica a recuperação após falhas do servidor. O sistema de arquivos Zebra [Hartman e Ousterhout 1995], desenvolvidos como uma continuação do trabalho original do LFS, combina escritas estruturadas em *log* com uma estratégia de RAID distribuído – os segmentos de *log* são subdivididos em seções com dados de correção de erro e escritos nos discos em diferentes nós de rede. É obtido um desempenho de quatro a cinco vezes o do NFS para gravação de arquivos grandes, com menores ganhos para arquivos curtos.

Novas estratégias de projeto ◊ A disponibilidade de redes comutadas de alto desempenho (como a ATM e a Gigabit Ethernet) tem estimulado vários esforços para fornecer sistemas de armazenamento persistente que distribuem dados de arquivo de maneira altamente favorável à escalabilidade e à tolerância a falhas entre muitos nós em uma intranet, separando as responsabilidades da leitura e da escrita de dados das de gerenciar os metadados e atender as requisições dos clientes. A seguir, esboçamos dois desses desenvolvimentos.

Essas estratégias escalam melhor do que os servidores mais centralizados que descrevemos nas seções anteriores. Geralmente, elas exigem um alto nível de confiança entre os computadores que cooperam para fornecer o serviço, pois incluem um protocolo de nível mais baixo para a comunicação com os nós que contêm dados (algo muito parecido com uma API de "disco virtual"). Portanto, sua abrangência provavelmente é limitada a uma única rede local.

xFS: um grupo da Universidade da Califórnia, em Berkeley, propôs uma arquitetura de sistema de arquivos de rede sem servidor e desenvolveu um protótipo de implementação, o xFS [Anderson *et al.* 1996]. Sua estratégia foi motivada por três fatores:

1. a oportunidade oferecida pelas rápidas redes locais comutadas de permitir que vários servidores de arquivos transfiram grandes volumes de dados de forma concorrente para os clientes;
2. a maior demanda de acesso a dados compartilhados;
3. as limitações fundamentais dos sistemas baseados em servidores de arquivos centralizados.

A respeito de (3), eles se referem ao fato de que a construção de servidores NFS de alto desempenho exige hardware relativamente caro, com várias CPUs, controladores de discos e de rede, e que existem limites para o processo de particionamento do espaço de arquivos – a distribuição de arquivos compartilhados em sistemas de arquivos separados, montados em diferentes servidores. Eles também apontam para o fato de que um servidor central representa um único ponto de falha.

O xFS é sem servidor, no sentido de que ele distribui as responsabilidades de processamento de um servidor de arquivos para um conjunto de computadores disponíveis em uma rede local, com uma granularidade de arquivos individuais. As responsabilidades de armazenamento são distribuídas independentemente do gerenciamento e de outras responsabilidades de serviço: o xFS implementa um sistema de armazenamento RAID em software, dividindo dados de arquivo entre discos de vários computadores (a esse respeito, ele é um precursor do Tiger Video File System, descrito no Capítulo 17), junto com uma técnica de estruturação de *log* de uma maneira semelhante ao sistema de arquivos Zebra.

A responsabilidade pelo gerenciamento de cada arquivo pode ser alocada para qualquer um dos computadores que suportam o serviço xFS. Isso é obtido por meio de uma estrutura de metadados chamada *mapa de gerenciador*, que é replicada em todos os clientes e servidores. Os identificadores de arquivo incluem um campo que atua como índice no mapa de gerenciador, e cada entrada no mapa identifica o computador correntemente responsável por gerenciar o arquivo correspondente. Várias outras estruturas de metadados, semelhantes àquelas encontradas em outros sistemas de armazenamento estruturado em *log* e RAID, são usadas para o gerenciamento do armazenamento de arquivos estruturado em *log* e no armazenamento em tiras de discos.

Foi construído um protótipo preliminar do xFS e seu desempenho foi avaliado. O protótipo estava incompleto no momento em que a avaliação foi realizada – a implementação da recuperação de falha estava inacabada e o esquema de armazenamento estruturado em *log* não tinha um componente limpador para recuperar o espaço ocupado por *logs* antigos e arquivos compactados.

As avaliações de desempenho realizadas com esse protótipo usaram 32 Sun SPARCStations, com um e com dois processadores (*dual*), conectados em uma rede de alta velocidade. As avaliações compararam o serviço de arquivo do xFS executando em até 32 estações de trabalho com o NFS e o AFS executando, cada um, em uma máquina dual. As larguras de banda de leitura e escrita obtidas com o xFS, usando 32 servidores, ultrapassaram as do NFS e do AFS, em um único servidor de processador dual, por um fator de aproximadamente dez. A diferença no desempenho foi muito menos pronunciada quando o xFS foi comparado com o NFS e o AFS usando o *benchmark* padrão do AFS. Mas, de modo geral, os resultados indicam que a arquitetura de processamento e armazenamento altamente distribuída do xFS apresentam uma tendência promissora para a obtenção de uma melhor escalabilidade em sistemas de arquivos distribuídos.

Frangipani: o Frangipani é um sistema de arquivos distribuído com alta escalabilidade, desenvolvido e implantado no *Digital Systems Research Center* (agora, *Compaq Systems Research Center*) [Thekkath *et al.* 1997]. Seus objetivos são muito parecidos com os do xFS e, assim como o xFS, ele os aborda com um projeto que separa as responsabilidades de armazenamento persistente das outras ações de um serviço de arquivos. Mas o serviço do Frangipani é estruturado como duas camadas totalmente independentes. A camada inferior é fornecida pelo sistema de disco virtual distribuído Petal [Lee e Thekkath 1996].

O Petal fornece uma abstração de disco virtual distribuído sobre os discos localizados em vários servidores, em uma rede local comutada. A abstração de disco virtual tolera a maioria das falhas de hardware e de software com a ajuda de réplicas dos dados armazenados e harmoniza, por meio do reposicionamento dos dados, a carga de forma autônoma nos servidores. Os discos virtuais do Petal são acessados por meio de um *driver* de disco UNIX, usando operações de entrada e saída de blocos padrão, de modo que eles podem ser usados para suportar a maioria dos sistemas de arquivos. O Petal acrescenta entre 10 e 100% na latência de acessos a disco, mas a estratégia de uso da cache resulta em desempenhos de saída de leitura e escrita pelo menos tão bons quanto os das unidades de disco locais.

Os módulos servidores do Frangipani são executados dentro do núcleo do sistema operacional. Assim como no xFS, a responsabilidade por gerenciar arquivos e as tarefas associadas (incluindo serviço de travamento de arquivos para os clientes) é atribuída dinamicamente aos *hosts*, e todas as máquinas enxergam um espaço de nomes de arquivo unificado, com acesso coerente (com semântica aproximada à da cópia única) para os arquivos compartilhados que podem ser atualizados. Os dados são armazenados em um formato estruturado em *log* e em tiras, no disco virtual Petal. O uso do Petal desobriga o Frangipani da necessidade de gerenciar espaço físico em disco, resultando em uma implementação de sistema de arquivos distribuído muito mais simples. O Frangipani pode simular as interfaces de vários serviços de arquivo existentes, incluindo o NFS e o DCE/DFS. O desempenho do Frangipani é pelo menos tão bom quanto o da implementação da Digital do sistema de arquivos UNIX.

8.6 Resumo

Os principais problemas de projeto para sistemas de arquivos distribuídos são:

- o uso eficiente de cache no cliente para obter um desempenho igual ou melhor do que o dos sistemas de arquivos locais;
- a manutenção da consistência entre várias cópias de arquivos de cliente colocadas em cache, quando essas cópias são atualizadas;
- a recuperação após uma falha do cliente ou do servidor;
- alto desempenho de saída para leitura e escrita de arquivos de todos os tamanhos;
- capacidade de mudança de escala.

Os sistemas de arquivos distribuídos são bastante empregados em sistemas computacionais institucionais e seu desempenho tem sido o assunto de muita otimização. O NFS possui um protocolo sem estado (*stateless*), mas tem mantido, com a ajuda de alguns aprimoramentos relativamente pequenos

no protocolo, sua posição inicial como tecnologia de sistema de arquivos distribuído dominante contando com implementações otimizadas e suporte de hardware de alto desempenho.

O AFS demonstrou a exeqüibilidade de uma arquitetura relativamente simples, usando o estado do servidor para reduzir o custo da manutenção de caches de cliente coerentes. O AFS supera o NFS em muitas situações. Avanços recentes têm empregado a disposição de dados em tiras (*striping*) entre vários discos e gravação estruturada em *log* para melhorar ainda mais o desempenho e a capacidade de mudança de escala.

Os sistemas de arquivos distribuídos mais modernos têm alta escalabilidade, fornecem bom desempenho entre redes locais e remotas, mantêm a semântica de atualização de arquivos com cópia única e toleram e se recuperam de falhas. Os requisitos futuros incluem o suporte para usuários móveis com operação desconectada, e reintegração automática e garantia de qualidade do serviço para satisfazer a necessidade de armazenamento persistente e a distribuição de fluxos de dados multimídia e outros fluxos dependentes do tempo. As soluções para esses requisitos serão discutidas nos Capítulos 15 e 17.

Exercícios

8.1 Por que não existe nenhuma operação *open* ou *close* em nossa interface para o serviço de arquivos simples ou para o serviço de diretório? Quais são as diferenças entre nossa operação de serviço de diretório *Lookup* e a operação *open* do UNIX? *páginas 293–296*

8.2 Descreva em linhas gerais os métodos pelos quais um módulo cliente poderia simular a interface de sistema de arquivos UNIX usando nosso modelo de serviço de arquivos. *páginas 293–296*

8.3 Escreva uma função *PathLookup*(*Pathname, Dir*) → *UFID* que implemente a operação *Lookup* para nomes de caminho do tipo UNIX, com base em nosso modelo de serviço de diretório. *páginas 293–296*

8.4 Por que os UFIDs devem ser exclusivos entre todos os sistemas de arquivos possíveis? Como a exclusividade dos UFIDs é garantida? *página 296–297*

8.5 Até que ponto o Sun NFS se desvia da semântica de atualização de arquivo com cópia única? Construa um cenário no qual dois processos em nível de usuário compartilhando um arquivo funcionariam corretamente em um único *host* UNIX, mas observariam inconsistências ao serem executados em *hosts* diferentes. *página 302–303*

8.6 O Sun NFS pretende suportar sistemas distribuídos heterogêneos por meio de um serviço de arquivo independente do sistema operacional. Quais são as principais decisões que o desenvolvedor de um servidor NFS para um sistema operacional que não seja o UNIX teria de tomar? Quais restrições um sistema de armazenamento deve obedecer para ser conveniente para a implementação de servidores NFS? *página 297–298*

8.7 Quais dados o módulo cliente NFS deve conter sobre cada processo em nível de usuário? *páginas 297–298*

8.8 Descreva em linhas gerais as implementações de módulo cliente para as chamadas de sistema UNIX *open*() e *read*(), usando as chamadas de RPC do NFS da Figura 8.9, (i) sem e (ii) com uma cache no cliente. *páginas 299, 303*

8.9 Explique por que a interface RPC das primeiras implementações do NFS é potencialmente insegura. A brecha de segurança foi fechada no NFS 3 pelo uso de criptografia. Como a chave de criptografia é mantida em segredo? A segurança da chave é adequada? *páginas 299, 305*

8.10 Após o *timeout* de uma chamada de RPC para acessar um arquivo em um sistema de arquivos montado incondicionalmente, o módulo cliente NFS não retorna o controle para o processo em nível de usuário que originou a chamada. Por quê? *página 299*

8.11 Como o *automounter* do NFS ajuda a melhorar o desempenho e a capacidade de mudança de escala do NFS? *página 301–302*

8.12 Quantas chamadas de *Lookup* são necessárias para solucionar um nome de caminho de cinco partes (por exemplo, /usr/users/jim/code/xyz.c) para um arquivo armazenado em um servidor NFS? Qual é o motivo para realizar a transformação passo a passo? *página 300–301*

8.13 Qual condição deve ser atendida pela configuração das tabelas de montagem nos computadores clientes para que a transparência de acesso seja obtida em um sistema de arquivos baseado no NFS? *página 300–301*

8.14 Como o AFS obtém o controle, quando é executada por um cliente uma chamada de sistema de abertura (ou fechamento) referindo-se a um arquivo no espaço de arquivo compartilhado?*página 308–310*

8.15 Compare a semântica de atualização do UNIX ao acessar arquivos locais com as do NFS e do AFS. Sob quais circunstâncias os clientes poderiam conhecer as diferenças? *páginas 302, 313*

8.16 Como o AFS trata com o risco de perda de mensagens de *callback*? *página 311–313*

8.17 Quais características do projeto do AFS o tornam mais escalável que o NFS? Quais são os limites para sua escalabilidade, supondo que servidores podem ser adicionados conforme for exigido? Quais desenvolvimentos recentes oferecem maior escalabilidade? *páginas 305–306, 314–315, 318–319*

9 Serviço de Nomes

Este capítulo apresenta o serviço de nomes como um serviço distinto a ser usado por processos clientes para, a partir do nome simbólico de um recurso ou objeto, obter seus atributos como, por exemplo, endereços. As entidades nomeadas podem ser de vários tipos e podem ser gerenciadas por diferentes serviços. Por exemplo, os serviços de nomes são freqüentemente usados para conter os endereços, e outros detalhes, de usuários, computadores, domínios de rede, serviços e objetos remotos. Além dos serviços de nomes, descreveremos os serviços de diretório, os quais pesquisam por serviços a partir de alguns de seus atributos.

Os problemas básicos de projeto do serviço de nomes, como a estrutura e o gerenciamento do espaço de nomes, e as operações suportadas por ele, são descritos em linhas gerais e ilustrados no contexto do *Domain Name Service* da Internet.

Também examinaremos como os serviços de nomes são implementados, abordando aspectos como a navegação por um conjunto de servidores de nome ao resolver um nome, armazenamento em cache dos resultados da resolução e a replicação de nomes e atributos para aumentar o desempenho e a disponibilidade.

São incluídos mais dois estudos de caso: o Global Name Service e o X.500 Directory Service, incluindo o LDAP.

9.1 Introdução

Em um sistema distribuído, são usados nomes para fazer referência a uma ampla variedade de recursos, como computadores, serviços, objetos remotos e arquivos, assim como usuários. A atribuição de nomes é um problema facilmente desprezado, mas com certeza fundamental no projeto de sistemas distribuídos. Os nomes facilitam a comunicação e o compartilhamento de recursos. É necessário um nome para pedir a um sistema de computador atuar sobre um recurso específico escolhido entre muitos; por exemplo, é necessário um nome na forma de um URL para acessar uma página web específica. Os processos não podem compartilhar recursos em particular a não ser que possam atribuir nomes a eles de forma consistente. Os usuários não podem se comunicar por meio de um sistema distribuído, a menos que possam dar nomes uns aos outros, por exemplo, com endereços de e-mail.

Os nomes não são a única maneira útil de identificação: outras formas são os atributos descritivos. Às vezes, os clientes não sabem o nome da entidade em particular que procuram, mas têm alguma informação que a descreve. Ou então, o cliente solicita um serviço (em vez de uma entidade em particular que o implemente) e conhece algumas das características que o serviço exigido deve ter.

Este capítulo apresenta os serviços de nomes, os quais fornecem aos clientes dados sobre objetos nomeados em sistemas distribuídos, e o conceito relacionado dos serviços de diretório, os quais fornecem dados sobre objetos que satisfazem determinada descrição. Descreveremos estratégias adotadas no projeto e na implementação desses serviços, usando o DNS (*Domain Name Service*), o GNS e o X500 como estudos de caso. Começaremos examinando os conceitos fundamentais dos nomes e atributos.

9.1.1 Nomes, endereços e outros atributos

Todo processo que exige acesso a um recurso específico deve possuir um nome ou um identificador para ele. Exemplos de nomes legíveis por seres humanos são nomes de arquivo, como */etc/passwd*, URLs, como *http://www.cdk4.net/*, e nomes de domínio Internet, como *www.cdk4.net*. O termo *identificador* às vezes é usado para se referir aos nomes que são interpretados apenas por programas. Referências de objeto remotas e manipuladores de arquivo NFS são exemplos de identificadores. Os identificadores são escolhidos por causa da eficiência com que podem ser pesquisados e armazenados pelo software.

Needham [1993] faz uma distinção entre um nome *puro* e outros tipos de nomes. Os nomes puros são simplesmente padrões de bits não interpretados. Os nomes que não são puros contêm informações sobre o objeto que nomeiam; em particular, eles podem conter informações sobre a localização do objeto. Os nomes puros sempre precisam ser pesquisados antes de poderem ser usados. No outro extremo de um nome puro está o *endereço* de um objeto: um valor que identifica a localização do objeto, em vez do objeto em si. Os endereços são eficientes para acessar objetos, mas os objetos às vezes podem ser movidos; portanto, os endereços são inadequados como meio de identificação. Por exemplo, os endereços de e-mail dos usuários normalmente precisam mudar quando eles trocam de organizações ou de provedores de serviços de Internet; eles não são auto-suficientes para fazer referência a um indivíduo específico com o passar do tempo.

Dizemos que um nome é *resolvido*, quando ele é convertido em dados sobre o recurso ou objeto nomeado, freqüentemente para invocar uma ação sobre ele. A associação entre um nome e um objeto é chamada de *vínculo* (*binding*). Em geral, os nomes são vinculados a *atributos* dos objetos nomeados, em vez da implementação dos objetos em si. Atributo é o valor de uma propriedade associada a um objeto. Um atributo importante de uma entidade, que normalmente é relevante em um sistema distribuído, é seu endereço. Por exemplo:

- O DNS faz o mapeamento de nomes de domínio Internet para atributos de um computador *host*: seu endereço IP, o tipo de entrada (por exemplo, uma referência a um servidor de correio eletrônico ou a outro *host*) e, por exemplo, o período de tempo durante o qual a entrada desse *host* permanecerá válida.
- O serviço de diretório X500 pode ser usado para fazer o mapeamento do nome de uma pessoa em atributos, incluindo seu endereço de e-mail e número de telefone.

- O serviço de nomes (*Naming Service*) e o serviço de negociação (*Trading Service*) do CORBA serão apresentados no Capítulo 20. O serviço de nomes faz o mapeamento do nome de um objeto remoto para sua referência de objeto remota, enquanto o serviço de negociação (*Trading Service*) faz esse mesmo mapeamento junto com um número arbitrário de atributos descrevendo o objeto em termos inteligíveis por usuários humanos.

Note que um "endereço" freqüentemente pode ser considerado apenas como outro nome que deve ser pesquisado, ou que pode conter tal nome. Um endereço IP deve ser pesquisado para se obter um endereço físico de rede, como um endereço Ethernet (MAC). Analogamente, os navegadores web e os clientes de e-mail utilizam o DNS para interpretar os nomes de domínio nos URLs e endereços de e-mail. A Figura 9.1 mostra a resolução da primeira parte de um nome de domínio de um URL, primeiro por meio do DNS, em um endereço IP, e depois, por meio de ARP, em um endereço Ethernet, para o servidor web. A última parte do URL é resolvida pelo sistema de arquivos do servidor web para localizar o arquivo relevante.

Nomes e serviços ◊ Muitos nomes usados em um sistema distribuído são específicos para algum serviço em particular. Um cliente usa tal nome ao solicitar que um serviço efetue uma operação sobre um objeto ou recurso nomeado que gerencia. Por exemplo, para solicitar a exclusão de um arquivo deve-se fornecer o seu nome para o serviço de arquivos; e, ao enviar um sinal para um processo é necessário identificá-lo junto ao serviço de gerenciamento de processos. Esses nomes são usados apenas no contexto do serviço que gerencia os objetos nomeados, exceto quando os clientes se comunicam a respeito de objetos compartilhados.

Os nomes também são necessários para referenciar às entidades de um sistema distribuído que estão fora da abrangência de um serviço. Os principais exemplos dessas entidades são usuários (com nomes próprios, nomes de *login*, identificadores de usuário e endereços de correio eletrônico), computadores (com nomes de *host* como *bruno*, *bronwyn*) e os próprios serviços (como *serviço de arquivos*, *serviço de impressora*). No *middleware* baseado em objetos, os nomes se referem aos objetos remotos que fornecem serviços ou aplicativos. Note que todos esses nomes devem ser legíveis e significativos para seres humanos, pois os usuários e administradores de sistema precisam se referir aos principais componentes e a configuração de sistemas distribuídos; os programadores precisam se referir a serviços em programas; e os usuários precisam se comunicar por meio do sistema distribuído e discutir quais serviços estão disponíveis em diferentes partes dele. Dada a conectividade fornecida pela Internet, esses requisitos de atribuição de nomes são potencialmente mundiais na abrangência.

Uniform Resource Identifiers ◊ Os URIs (*Uniform Resource Identifiers*) [Berners Lee *et al.* 2005] surgiram da necessidade de identificar recursos web e outros recursos de Internet, como as caixas de

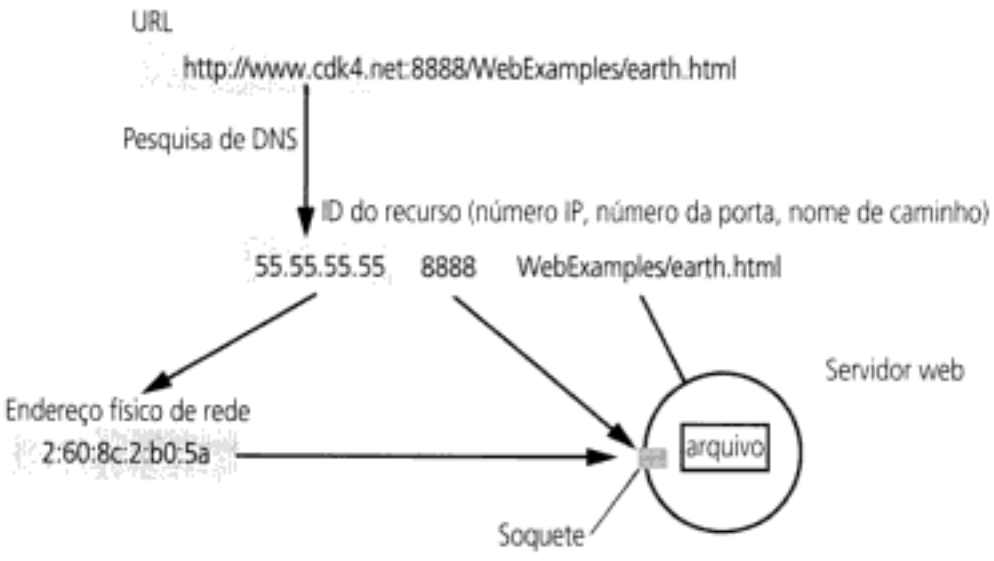


Figura 9.1 Composição de um nome de domínio para identificar um recurso a partir de um URL.

correio eletrônicas. Um objetivo importante era identificar esses recursos de maneira coerente, de modo que todos pudessem ser processados por um software comum, como os navegadores. Os URIs são uniformes no sentido de que sua sintaxe incorpora a de muitos tipos de identificadores de recurso individuais (isto é, *esquemas* de URI) e que existem procedimentos para gerenciar o espaço de nomes global desses esquemas. A vantagem da uniformidade é que ela facilita a introdução de novos tipos de identificador, assim como o uso em novos contextos dos tipos de identificador já existentes, sem invalidar a sua atual utilização.

Por exemplo, se alguém inventasse um novo tipo de URI *widget*, então os URIs que começassem com *widget:* teriam que obedecer a sintaxe de URI global, assim como todas as regras locais definidas para o esquema identificador de *widget*. Esses URIs identificariam recursos de *widget* de uma maneira bem definida. Mas mesmo um software já existente que não acessasse recursos de *widget* ainda poderia processar URIs *widget* – por exemplo, gerenciando diretórios que os contivesse. Recorrendo a um exemplo de incorporação de identificadores existente, isso foi feito para os números de telefone, prefixando-se o nome de esquema *tel* e padronizando-se a representação de números de telefone, como em *tel:+1-816-555-1212*. Esses URIs *tel* se destinam a usos como *links* web que fazem ligações telefônicas quando invocados.

Uniform Resource Locators: alguns URIs fornecem informações para localizar um recurso (como um nome de *host* DNS e um nome de caminho nessa máquina), enquanto outros são usados como nomes de recurso puros. O conhecido termo *Uniform Resource Locator* (URL) é reservado para identificadores que são localizadores de recursos, incluindo os URLs *http*, apresentados na Seção 1.3, como *http://www.cdk4.net/*, que identifica a página web no caminho dado (/), no *host www.cdk4.net*. Outro exemplo são os URLs *mailto*, como *mailto:fred@flintstone.org*, que identifica a caixa de correio eletrônico no endereço dado.

Os URLs são identificadores eficientes para acessar recursos, mas eles sofrem da desvantagem de que, se um recurso for excluído ou mudar, digamos, de um site web para outro, então poderão existir *links* "quebrados" para o recurso, contendo o URL antigo. Se um usuário clicar em um *link* quebrado de um recurso web, o servidor web responderá que o recurso não pode ser encontrado ou – talvez, pior – fornecerá um recurso diferente, que agora ocupa a mesma localização.

Uniform Resource Names: os URNs (*Uniform Resource Names*) são URIs utilizados como nomes de recurso puros, em vez de localizadores. Por exemplo, o URI:

mid:0E4FC272-5C02-11D9-B115-000A95B55BC8@hpl.hp.com

é um URN que identifica a mensagem de e-mail contida em seu campo *Message-Id*. O URI serve para distinguir essa mensagem de qualquer outra mensagem de e-mail. Mas ele em si não fornece o endereço da mensagem em algum meio de armazenamento e é necessária uma operação de pesquisa para encontrá-la.

Uma subárvore especial dos URIs que começam com *urn:* foi reservada para os URNs – embora, como mostra o exemplo *mid:*, nem todos os URNs são URIs *urn:*. Todos estes URIs prefixados com *urn* são da forma *urn:espaçodeNome:espaçodeNome-nomeEspecífico*. Por exemplo, urn:ISBN:0-201-62433-8 identifica livros que apresentam o nome 0-201-62433-8 no esquema de atribuição de nomes padrão ISBN. Como outro exemplo, o nome (inventado) *urn:doi:10.555/music-pop-1234* se refere à publicação chamada *music-pop-1234* no esquema de atribuição de nomes da editora conhecida como *10.555* no esquema DOI (*Digital Object Identifier*) [www.doi.org].

Existem serviços de resolução (serviços de nomes, na terminologia deste capítulo), como o *Handle System*, para resolver URNs do tipo DOI [www.handle.net] em atributos de recurso, mas nenhum deles é amplamente usado. Na verdade, continua o debate nas comunidades de pesquisa da web e Internet sobre até que ponto uma categoria separada de URNs é necessária. Uma escola de pensamento diz que "os bons URLs não mudam" – em outras palavras, que todo mundo deve atribuir URLs aos recursos com garantias sobre a continuidade de sua referência. Contra esse ponto de vista está a observação de que nem todo mundo está em uma posição favorável para dar tais garantias, pois exige os meios para se manter o controle de um nome de domínio e administrar os recursos com cuidado.

9.2 Serviços de nomes e o Domain Name System

Um serviço de nomes armazena um conjunto de um ou mais *contextos* de atribuição de nomes – conjuntos de vínculos entre nomes textuais e atributos de objetos, como usuários, computadores, serviços e objetos remotos. A principal operação que um serviço de nomes suporta é a resolução de um nome – isto é, pesquisar atributos de determinado nome. Descreveremos a implementação da resolução de nomes na Seção 9.2.2. Também são necessárias operações para criar novos vínculos, excluir vínculos, listar nomes vinculados e para adicionar e excluir contextos.

Devido ao requisito dos sistemas distribuídos serem abertos, o gerenciamento de nomes é completamente separado dos outros serviços, o que traz as seguintes motivações:

Unificação: freqüentemente, é conveniente que recursos gerenciados por diferentes serviços utilizem o mesmo esquema de atribuição de nomes. Os URIs são um bom exemplo disso.

Integração: nem sempre é possível prever a abrangência do compartilhamento em um sistema distribuído. Pode-se tornar necessário compartilhar e, portanto, nomear recursos que foram criados em diferentes domínios administrativos. Sem um serviço de nomes comum, os domínios administrativos podem usar convenções de atribuição de nomes totalmente diferentes.

Requisitos gerais do serviço de nomes ◊ Originalmente, os serviços de nomes eram muito simples, pois foram projetados apenas para atender a necessidade de vincular nomes a endereços em um único domínio de gerenciamento, correspondendo a uma única LAN ou WAN. A interconexão de redes, e a maior escala dos sistemas distribuídos, geraram um problema de mapeamento de nomes muito maior.

O Grapevine [Birrell *et al.* 1982] foi um dos primeiros serviços de nomes extensíveis e de múltiplos domínios. Ele foi explicitamente projetado para ser escalável, em pelo menos duas ordens de grandeza, no número de nomes e na carga de pedidos que poderia manipular.

O *Global Name Service*, desenvolvido no Digital Equipment Corporation Systems Research Center [Lampson 1986], é um descendente do Grapevine com objetivos mais ambiciosos, incluindo:

Manipular um número essencialmente arbitrário de nomes e servir a um número arbitrário de organizações administrativas. Por exemplo, o sistema deve ser capaz, dentre outras coisas, de manipular os endereços de correio eletrônico de todos os usuários de computador do mundo.

Um tempo de ciclo de vida longo. Muitas mudanças irão ocorrer na organização do conjunto de nomes e nos componentes que implementam o serviço, durante seu tempo de ciclo de vida.

Alta disponibilidade. A maior parte dos outros sistemas depende do serviço de nomes; e eles não podem funcionar quando o serviço de nomes está com defeito.

Isolamento de falhas. Para que falhas locais não façam o serviço inteiro falhar.

Tolerância à suspeita. Um sistema aberto grande não pode ter nenhum componente que seja confiável para todos os clientes do sistema.

Dois exemplos de serviços de nome que se concentraram no objetivo de escalabilidade para grandes números de objetos, como endereços de e-mail ou documentos de usuários, são o serviço de nomes Globe [van Steen *et al.* 1998] e o Handle System [www.handle.net]. O DNS (*Domain Name System*) da Internet, apresentado no Capítulo 3, nomeia objetos (na prática, computadores) na Internet. Para fornecer um serviço satisfatório, ela conta muito com a replicação e o uso de cache. O projeto do DNS, e de outros serviços de nome, pressupõe que a consistência da cache não precisa ser rigorosamente mantida, como no caso das cópias de arquivos armazenadas em cache, pois as atualizações são menos freqüentes e o uso de uma cópia desatualizada de uma resolução de nome geralmente pode ser detectado pelo software cliente.

Nesta seção, discutimos os principais problemas de projeto de serviços de nomes, dando exemplos do DNS. A seguir, apresentaremos um estudo de caso detalhado do DNS.

9.2.1 Espaços de nomes

Espaço de nome é o conjunto de todos os nomes válidos reconhecidos por um serviço em particular. Se um nome é válido, isso significa que o serviço tentará pesquisá-lo, mesmo que o nome não venha a corresponder a nenhum objeto – ou seja, está *desvinculado*. Os espaços de nomes exigem uma definição sintática. Por exemplo, o nome "Dois" poderia não ser o nome de um processo do UNIX, enquanto o inteiro "2", poderia ser. Analogamente, o nome "..." não é aceitável como nome DNS de um computador.

Os nomes podem ter uma estrutura interna que represente sua posição em um espaço de nomes hierárquico, como no sistema de arquivos do UNIX, ou em uma hierarquia organizacional, como no caso dos nomes de domínio Internet; ou ainda, eles podem ser escolhidos em um conjunto simples de identificadores numéricos ou simbólicos. A vantagem mais importante dos espaços de nomes hierárquicos é que cada parte de um nome é resolvida em relação a um contexto separado e o mesmo nome pode ser usado com diferentes significados, em diferentes contextos. No caso dos sistemas de arquivos, cada diretório representa um contexto. Assim, */etc/passwd* é um nome hierárquico com dois componentes. O primeiro, *etc*, é transformado em relação ao contexto /, ou raiz, e a segunda parte, *passwd*, é relativa ao contexto */etc*. O nome */oldetc/passwd* pode ter um significado diferente, pois seu segundo componente é resolvido em outro contexto. Analogamente, o mesmo nome */etc/passwd* pode ser resolvido para arquivos diferentes nos contextos de dois computadores diferentes.

Os espaços de nome hierárquicos são potencialmente infinitos, de modo que eles permitem que um sistema cresça indefinidamente. Normalmente, os espaços de nomes planos são finitos; seu tamanho é determinado pela fixação de um comprimento máximo permitido para os nomes. Se nenhum limite for estabelecido para o comprimento dos nomes em um espaço de nomes planos, então ele também será potencialmente infinito. Outra vantagem em potencial de um espaço de nome hierárquico é que diferentes contextos podem ser gerenciados por diferentes pessoas.

A estrutura dos URLs http foi apresentada no Capítulo 1. O espaço de nomes URL inclui *nomes relativos*, como *../images/figure1.jpg*. Nesse esquema de URL, o nome de *host* do servidor, e o diretório de servidor ao qual esse nome de caminho se refere, são considerados por um navegador como iguais aos do documento em que ele está incorporado.

Os nomes DNS são chamados de *nomes de domínio* – são strings semelhantes aos nomes de arquivo absolutos do UNIX. Alguns exemplos deles são: *www.cdk4.net* (um computador), *net*, *com* e *ac.uk* (os três últimos são domínios).

O espaço de nomes DNS tem uma estrutura hierárquica: um nome de domínio consiste em um ou mais strings chamados de *componentes do nome* ou *rótulos*, separados pelo delimitador. Não há nenhum delimitador no início ou no final de um nome de domínio, embora a raiz do espaço de nome DNS às vezes seja referenciada como '.' para propósitos administrativos. Os componentes do nome são strings imprimíveis não-nulos que não contêm '.'. Em geral, o prefixo de um nome é a sua parte inicial que contêm zero ou mais componentes. Por exemplo, no DNS, *www* e *www.cdk4* são prefixos de *www.cdk4.net*. Os nomes DNS não levam em consideração letras maiúsculas e minúsculas; portanto, *www.cdk4.net* e *WWW.CDK4.NET* têm o mesmo significado.

Os servidores DNS não reconhecem nomes relativos: todos os nomes se referem à raiz global. Entretanto, na prática, as implementações dos softwares cliente mantém uma lista de nomes de domínio que são anexados automaticamente a todo nome de componente único, antes da transformação. Por exemplo, o nome *www* apresentado no domínio *cdk4.net* provavelmente se refere a *www.cdk4.net*; o software cliente anexará o domínio padrão *cdk4.net* e tentará resolver esse nome. Se isso falhar, então mais nomes de domínio padrão poderão ser anexados; finalmente, o nome (absoluto) *www* é apresentado à raiz para resolução (uma operação que, é claro, falhará neste caso). Contudo, nomes com mais de um componente normalmente são apresentados intactos para o DNS – como nomes absolutos.

Aliases ◊ Infelizmente, nomes com mais de um ou dois componentes são complicados de digitar e lembrar. Em geral, um *alias* é semelhante a um vínculo simbólico do estilo UNIX, permitindo que um nome conveniente substitua um mais complicado. O DNS permite fazer *aliases* em que um nome de domínio é definido para representar outro. A razão para se ter *aliases* é propiciar transparência. Por exemplo, os *aliases* são geralmente usados para especificar os nomes das máquinas que executam um servidor web ou um servidor FTP. O nome *www.example.net* poderia ser um *alias* para um computa-

dor chamado *fred.example.net*. Isso tem a vantagem de que os clientes podem se referir ao servidor web por meio de um nome genérico, que não se refere a uma máquina em particular e, se o servidor web mudar para outro computador, basta atualizar o *alias* no banco de dados DNS.

Domínios de atribuição de nomes ◊ Um *domínio de atribuição de nomes* é um espaço de nome para o qual existe uma única autoridade administrativa global para atribuir nomes dentro dele. Essa autoridade tem o controle geral de quais nomes podem ser vinculados dentro do domínio, mas está livre para delegar essa tarefa.

No DNS, os domínios são conjuntos de nomes de domínio; sintaticamente, o nome de um domínio é o sufixo comum dos nomes de domínio que estão dentro dele, mas não pode ser distinguido de outra forma, por exemplo, do nome de um computador. Por exemplo, *net* é um domínio que contém *cdk4.net*. Note que o termo "nome de domínio" é potencialmente confuso, pois apenas alguns nomes de domínio identificam domínios (outros identificam computadores).

A administração de domínios pode ser delegada para subdomínios. O domínio *dcs.qmul.ac.uk* – o Departamento de Ciência da Computação do Queen Mary College (Universidade de Londres), no Reino Unido – pode conter qualquer nome que o departamento queira. Mas o nome de domínio *dcs.qmul.ac.uk* em si teve que ser combinado junto as autoridades da escola que gerenciam o domínio *qmul.ac.uk*. Analogamente, *qmul.ac.uk* teve que ser aceito pela autoridade registrada para *ac.uk*, etc.

A responsabilidade por um domínio de atribuição de nomes normalmente anda lado a lado com a de gerenciar e manter atualizada a parte correspondente do banco de dados armazenado em um servidor de nome autoridade sobre o domínio (*authoritative name server*) e usado pelo serviço de nomes. Em geral, os nomes pertencentes a diferentes domínios de atribuição de nomes são armazenados por servidores de nome distintos, gerenciados pelas autoridades correspondentes.

Combinando e personalizando espaços de nomes ◊ O DNS fornece um espaço de nomes global e homogêneo, no qual um determinado nome se refere à mesma entidade, independentemente de quem pesquise o nome. Em contraste, alguns serviços de nomes permitem que espaços de nomes distintos – às vezes, espaços de nomes heterogêneos – sejam incorporados a eles; e alguns serviços de nomes permitem ainda que o espaço de nomes seja personalizado de acordo com as necessidades de grupos individuais, usuários ou mesmo processos.

Integração: a prática de montar sistemas de arquivos no UNIX e no NFS (veja a Seção 8.3) fornece um exemplo no qual uma parte de um espaço de nome é convenientemente incorporada à outra. Mas considere a integração dos sistemas de arquivos inteiros de dois (ou mais) computadores UNIX chamados *red* e *blue*. Cada computador tem sua própria raiz, com nomes de arquivo coincidentes. Por exemplo, */etc/passwd* se refere a um arquivo em *red* e a um arquivo diferente em *blue*. A maneira óbvia de integrar os sistemas de arquivos é substituir a raiz de cada computador por uma "super-raiz" e montar o sistema de arquivos de cada computador nessa super-raiz, digamos, como */red* e */blue*. Então, os usuários e programas poderiam se referir a */red/etc/passwd* e a */blue/etc/passwd*. Mas a nova convenção de atribuição de nomes, por si só, faria os programas dos dois computadores que ainda usam o nome antigo */etc/passwd* funcionar de forma errada. Uma solução é deixar o conteúdo da raiz antiga em cada computador e incorporar os sistemas de arquivos montados */red* e */blue* dos dois computadores (supondo que isso não produza conflitos de nome com o conteúdo da raiz antiga).

A moral é que sempre podemos integrar espaços de nomes criando um contexto de raiz de nível mais alto, mas isso pode provocar um problema de compatibilidade com versões anteriores. A correção do problema da compatibilidade, por sua vez, nos deixa com espaços de nomes mistos e a inconveniência de ter de transladar nomes antigos entre os usuários dos dois computadores.

Heterogeneidade: o espaço de nome DCE (*Distributed Computing Environment*) [OSF 1997] permite a incorporação de espaços de nomes heterogêneos dentro dele. Os nomes DCE podem conter *junções*, que são semelhantes aos pontos de montagem do NFS e do UNIX (veja a Seção 8.3), exceto que permitem a montagem de espaços de nomes heterogêneos. Por exemplo, considere o nome DCE completo */.../dcs.qmul.ac.uk/principals/Jean.Dollimore*. A primeira parte desse nome, */.../dcs.qmul.ac.uk*, denota um contexto chamado *célula*. O componente seguinte é uma junção. Por exemplo, a junção *principals* é um contexto contendo principais de segurança, no qual o componente final, *Jean.Dollimore*, pode ser pesquisado. Analogamente, em */.../dcs.qmul.ac.uk/files/pub/reports/TR2000-99*,

a junção *files* é um contexto correspondente a um diretório de sistema de arquivos, no qual o componente final, *pub/reports/TR2000-99*, é pesquisado. As duas junções, *principals* e *files*, são as raízes de espaços de nomes heterogêneos, implementados por serviços de nomes heterogêneos.

Customização: vimos, no exemplo anterior da incorporação de sistemas de arquivos montados com o NFS, que às vezes os usuários preferem construir seus espaços de nomes de forma independente, em vez de compartilhar um único espaço de nomes. A montagem do sistema de arquivos permite que os usuários importem arquivos que estão armazenados em servidores e são compartilhados, enquanto os outros nomes continuam a fazer referência a arquivos locais, não-compartilhados, e que podem ser administrados de forma autônoma. Mas até os mesmos arquivos acessados a partir de dois computadores diferentes podem ser montados em diferentes pontos e, assim, ter nomes diferentes. Não compartilhando o espaço de nomes inteiro, os usuários precisam traduzir nomes entre computadores.

Outro exemplo de motivo para a personalização é quando o mesmo nome é feito de modo a se referir a arquivos distintos, em diferentes computadores. Por exemplo, o mesmo nome */bin/netscape* poderia, em princípio, ser vinculado a um programa no formato binário do x86, em um computador Pentium, e em um binário do PowerPC, em um computador Mac OS X. Esse mapeamento dos mesmos nomes para arquivos diferentes torna possível escrever *scripts* envolvendo esses nomes, que são executados corretamente em todas as configurações de máquina.

O serviço de nomes Spring [Radia *et al.* 1993] apresenta a capacidade de construir espaços de nomes dinamicamente e compartilhar contextos de atribuição de nomes individuais de forma seletiva. Ao contrário dos exemplos que acabamos de dar, dois processos diferentes, em um mesmo computador, podem ter contextos de nomes diferentes. Os contextos de nomes do Spring são objetos de primeira classe que podem ser compartilhados em torno de um sistema distribuído. Por exemplo, suponha que um usuário no computador *red* deseje executar um programa em *blue* que usa nome de caminhos como */etc/passwd*, mas que esses nomes sejam resolvidos para arquivos do sistema de arquivos de *red* e não no de *blue*. Isso pode ser obtido no Spring, passando-se uma referência do contexto de nomes local de *red* para *blue* e usando-o como contexto de nomes do programa. O Plan 9 [Pike *et al.* 1993] também permite que os processos tenham seus próprios espaços de nomes do sistema de arquivos. Um recurso novo do Plan 9 (mas que também pode ser implementado no Spring) é que os diretórios físicos podem ser ordenados e fusionados em um único diretório lógico. O efeito é que um nome no diretório lógico único é pesquisado na sucessão de diretórios físicos até que haja uma correspondência, quando então os atributos são retornados. Isso elimina a necessidade de fornecer listas de caminhos ao procurar arquivos de programa ou de biblioteca.

9.2.2 Resolução de nomes

Em geral, a resolução é um processo iterativo pelo qual um nome é repetidamente apresentado a diferentes contextos de atribuição de nomes. Um contexto de atribuição de nomes faz o mapeamento de determinado nome diretamente em um conjunto de atributos primitivos (como os de um usuário) ou faz o mapeamento a um novo contexto de atribuição de nomes e em um nome derivado a ser apresentado a esse contexto. Para resolver um nome, ele é primeiro apresentado a um contexto de atribuição de nomes inicial; enquanto o resultado da resolução for um novo contexto, ou um nome derivado, o processo continua de forma iterativa. Ilustramos isso no início da Seção 9.2.1, com o exemplo de */etc/passwd*, no qual *etc* é apresentado ao contexto */* e depois *passwd* é apresentado ao contexto */etc*.

Outro exemplo da natureza iterativa da transformação é o uso de *alias*. Por exemplo, quando um servidor de DNS é solicitado a resolver um *alias* como *www.dcs.qmul.ac.uk*, ele primeiro resolve o *alias* no outro nome de domínio (neste caso, *apricot.dcs.qmul.ac.uk*), o qual deve ser novamente resolvido para produzir um endereço IP.

Em geral, o uso de alias possibilita a presença de ciclos no espaço de nomes, caso em que a resolução poderá nunca terminar. Duas soluções são, primeiro, abandonar um processo de resolução após um número limite de sucessivas resoluções; e, segundo, deixar os administradores vetarem os *aliases* que introduziriam ciclos.

Servidores de nomes e navegação ◊ Qualquer serviço de nomes, como o DNS, que armazene um banco de dados muito grande e que seja usado por uma população grande, não armazenará todas

as suas informações de atribuição de nomes em um único computador servidor. Tal servidor seria um gargalo e um ponto de falha crítico. Todos os serviços de nomes muito utilizados devem usar replicação para obter alta disponibilidade. Veremos que o DNS especifica que cada subconjunto de seu banco de dados é replicado em pelo menos dois servidores independentes.

Mencionamos anteriormente que os dados pertencentes a um domínio de atribuição de nomes normalmente é armazenado por um servidor de nome local gerenciado pela autoridade responsável por esse domínio. Embora, em alguns casos, um servidor de nome possa armazenar dados de mais de um domínio, geralmente pode-se dizer que os dados são particionados nos servidores de acordo com seu domínio. Veremos que, no DNS, a maioria das entradas é relativa a computadores locais. Mas também existem servidores de nome para os domínios superiores, como *yahoo.com*, *ac.uk*, e para a raiz.

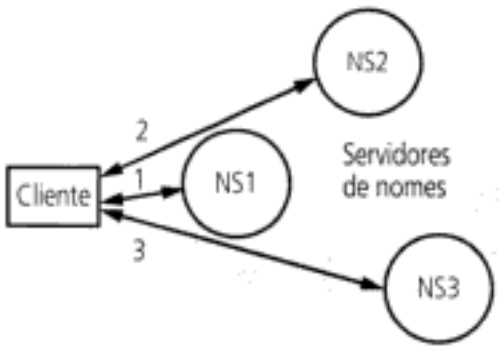
O particionamento de dados implica que o servidor de nome local não pode responder a todas as solicitações sem a ajuda de outros servidores de nome. Por exemplo, um servidor de nome no domínio *dcs.qmul.ac.uk* não seria capaz de fornecer o endereço IP de um computador no domínio *cs.purdue.edu*, a menos que estivesse armazenado em sua cache – certamente não na primeira vez que fosse solicitado.

O processo de localizar dados de atribuição de nomes dentre mais de um servidor para transformar um nome é chamado de *navegação*. O software cliente de resolução de nomes realiza a navegação em nome do cliente. Ele se comunica com servidores de nome, conforme for necessário, para transformar um nome. Ele pode ser fornecido como código de biblioteca e vinculado aos clientes como, por exemplo, na implementação BIND do DNS (veja a Seção 9.2.3) ou no Grapevine [Birrell *et al.* 1982]. A alternativa usada no X500 é fornecer a resolução de nomes em um processo separado, compartilhado por todos os processos clientes nesse computador.

O DNS suporta o modelo conhecido como *navegação iterativa* (veja a Figura 9.2). Para transformar um nome, um cliente apresenta um nome ao servidor de nome local, o qual tenta resolvê-lo. Se o servidor de nome local tiver o nome, ele retornará o resultado imediatamente. Se não tiver, ele sugerirá outro servidor que poderá ajudar. A resolução prossegue no novo servidor. Esse procedimento é repetido, tantas vezes quanto necessário, até que o nome seja localizado ou se descubra que ele é desvinculado.

Como o DNS é projetado para conter entradas para milhões de domínios e é acessado por um número enorme de clientes, não seria possível ter todas as consultas começando em um servidor raiz, mesmo que ele fosse bastante replicado. O banco de dados DNS é particionado entre os servidores de maneira a permitir que muitas consultas sejam atendidas de forma local e outras sejam atendidas sem necessidade de resolver cada parte do nome separadamente. O esquema de resolução de nomes do DNS será descrito com mais detalhes na Seção 9.2.3.

O NFS também emprega navegação iterativa na resolução de um nome de arquivo, componente por componente (veja o Capítulo 8). Isso porque o serviço de arquivo pode encontrar um vínculo simbólico ao resolver um nome. Um vínculo simbólico deve ser interpretado no espaço de nomes do sistema de arquivos do cliente, pois ele pode apontar para um arquivo em um diretório armazenado em outro servidor. O computador cliente deve determinar qual é esse servidor, pois somente o cliente conhece seus pontos de montagem.



Um cliente entra em contato iterativamente com os servidores de nomes NS1-NS3 para resolver um nome

Figura 9.2 Navegação iterativa.

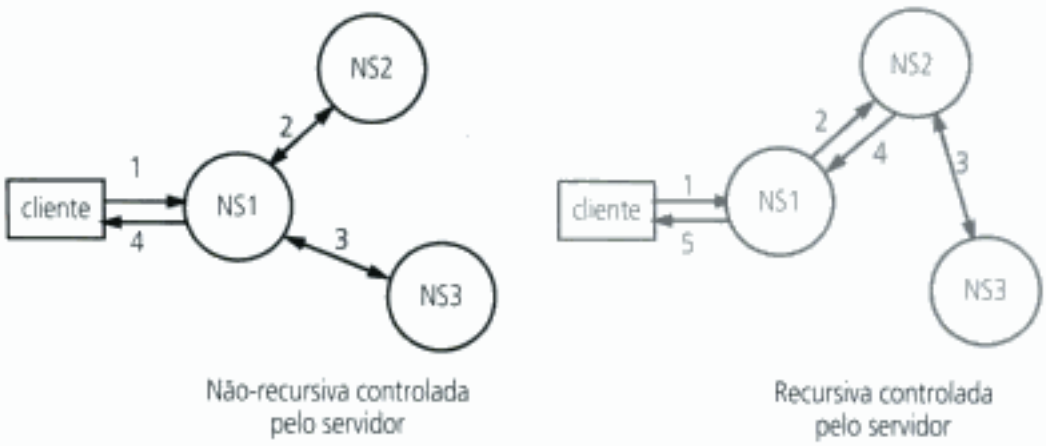
Na *navegação por multicast*, um cliente envia o nome a ser transformado e o tipo de objeto exigido para um grupo de servidores de nomes. Somente o servidor que contém os atributos nomeados responde a requisição. Infelizmente, entretanto, se o nome for desvinculado, a requisição será respondida com silêncio. Cheriton e Mann [1989] descrevem um esquema de navegação baseado em *multicast* no qual um servidor é incluído no grupo para responder quando o nome exigido for desvinculado.

Outra alternativa ao modelo de navegação iterativa é aquele em que um servidor de nome coordena a resolução do nome e devolve o resultado para o agente usuário. Ma [1992] distingue as *navegações controladas pelo servidor não-recursiva* e *recursiva* (Figura 9.3). Na navegação não-recursiva controlada pelo servidor, qualquer servidor de nome pode ser escolhido pelo cliente. Esse servidor se comunica por *multicast* ou iterativamente com seus pares, no estilo descrito anteriormente, como se fosse um cliente. Na navegação recursiva controlada pelo servidor, o cliente entra em contato com um único servidor. Se esse servidor não armazenar o nome, ele entra em contato com um de seus pares que armazena um prefixo (maior) do nome, o qual, por sua vez, tenta resolvê-lo. Esse procedimento continua recursivamente até que o nome seja resolvido.

Se um serviço de nomes abrange domínios administrativos distintos, então os clientes em execução em um domínio administrativo podem ser proibidos de acessar servidores de nomes pertencentes a outro domínio. Além disso, até os servidores de nomes podem ser proibidos de descobrir a disposição dos dados de atribuição de nomes nos servidores de nomes de outro domínio administrativo. Então, tanto a navegação controlada pelo cliente, como a navegação não-recursiva controlada pelo servidor, são inadequadas e deverá ser usada a navegação recursiva controlada pelo servidor. Servidores de nome autorizados solicitam os dados do serviço de nomes dos servidores de nome designados, gerenciados por diferentes administrações, o qual retorna os atributos sem revelar onde estão armazenadas as diferentes partes do banco de dados de atribuição de nomes.

Uso de cache ◊ No DNS e em outros serviços de nomes, o software cliente de resolução de nomes e os servidores mantêm uma cache com os resultados das resoluções de nomes anteriores. Quando um cliente solicita uma pesquisa de nome, o software de resolução de nomes consulta sua cache. Se contiver um resultado recente de uma pesquisa anterior do nome, ele o retornará para o cliente; caso contrário, começará a procurá-lo em um servidor. Esse servidor, por sua vez, pode retornar dados armazenados na cache de outros servidores.

O uso da cache é importante para o desempenho do serviço de nomes e ajuda na manutenção da disponibilidade tanto do serviço de nomes, como de outros serviços, a despeito de falhas do servidor de nome. Sua função na melhoria dos tempos de resposta, economizando na comunicação com servidores de nome, é clara. A cache pode ser utilizada para eliminar servidores de nome de alto nível – o servidor raiz, em particular – do caminho de navegação, permitindo que a resolução prossiga, apesar de algumas falhas de servidor.



Um servidor de nomes NS1 se comunica com outros servidores de nomes a pedido de um cliente

Figura 9.3 Navegação controlada pelo servidor não-recursiva e recursiva.

O uso de cache por resolvedores de nome clientes é amplamente aplicado nos serviços de nomes e é particularmente bem-sucedido, pois os dados de atribuição de nomes são alterados de forma relativamente rara. Por exemplo, é provável que informações como endereços de computadores, ou de serviços, permaneçam inalteradas por vários meses ou anos. Entretanto, existe a possibilidade de um serviço de nomes retornar atributos desatualizados, por exemplo, um endereço obsoleto, durante a resolução.

9.2.3 O Domain Name System

O *Domain Name System* é um projeto de serviço de nomes cujo banco de dados de atribuição de nomes é usado na Internet. Ele foi planejado por Mockapetris [1987], no RFC 1034, principalmente para substituir o esquema de atribuição de nomes original da Internet, no qual todos os nomes de *host* e de endereços eram mantidos em um único arquivo mestre central, e carregado por *download*, via FTP, em todos os computadores que dele necessitassem [Harrenstien *et al.* 1985]. Logo se viu que esse esquema original sofria de três defeitos importantes:

- Ele não tinha capacidade de escalabilidade para suportar grandes números de computadores.
- Organizações locais desejavam administrar seus próprios sistemas de atribuição de nomes.
- Era necessário um serviço de nomes geral – e não um que servisse apenas para pesquisar endereços de computadores.

Os objetos nomeados pelo DNS são principalmente computadores – para os quais, basicamente, os endereços IP são armazenados como atributos – e o que nos referimos neste capítulo como domínios de atribuição de nomes são chamados simplesmente de *domínios* no DNS. Em princípio, entretanto, qualquer tipo de objeto pode ser nomeado e sua arquitetura fornece abrangência para uma variedade de implementações. As organizações e os departamentos dentro delas podem gerenciar seus próprios dados de atribuição de nomes. Na Internet, milhões de nomes são vinculados ao DNS e nele pesquisados. Qualquer nome pode ser resolvido por qualquer cliente. Isso é obtido pelo particionamento hierárquico do banco de dados de nomes, pela replicação dos dados de atribuição de nomes e pelo uso de cache.

Nomes de domínio ◊ O DNS é projetado para ser usado em várias implementações, cada uma das quais podendo ter seu próprio espaço de nomes. Na prática, entretanto, somente um é amplamente usado e é o de atribuição de nomes da Internet. O espaço de nomes DNS da Internet é particionado tanto de forma organizacional como geográfica. Os nomes estão escritos com o domínio de nível mais alto à direita. Os domínios organizacionais de nível superior (também chamados de *domínios genéricos* ou *top-level domain*) usados na Internet, originalmente, eram:

com	–	Organizações comerciais
edu	–	Universidades e outras instituições educacionais
gov	–	Órgãos do governo norte-americano
mil	–	Organizações militares dos EUA
net	–	Principais centros de suporte à rede
org	–	Organizações não mencionadas anteriormente
int	–	Organizações internacionais

Novos domínios de nível superior foram adicionados no início dos anos 2000. Uma lista completa dos nomes de domínio genéricos atuais está disponível no *Internet Assigned Numbers Authority* no URL [www.iana.org].

Além disso, cada país tem seus próprios domínios:

us	–	Estados Unidos
uk	–	Reino Unido
fr	–	França
br	–	Brasil
...	–	...

Os países, particularmente outros que não os EUA, usam seu próprio domínio para distinguir suas organizações. O Reino Unido, por exemplo, tem os domínios *co.uk* e *ac.uk*, que correspondem a *com* e *edu* respectivamente (*ac* significa *academic community* – comunidade acadêmica). Note que, apesar de seu sufixo de caráter geográfico *uk*, um domínio como *doit.co.uk* poderia ter dados localizados no escritório espanhol da Doit Ltd, uma empresa britânica. Em outras palavras, mesmo os nomes de domínio de caráter geográfico são convencionais e completamente independentes de suas localizações físicas.

Consultas DNS ◊ O DNS é usado na Internet principalmente para a resolução de nomes de *host* e para pesquisar servidores de correio eletrônico, como segue:

Resolução de nomes de host: em geral, os aplicativos usam o DNS para transformar nomes de *host* em endereços IP. Por exemplo, quando um navegador web recebe um URL contendo o nome de domínio *www.dcs.qmul.ac.uk*, ele faz uma solicitação de DNS e obtém o endereço IP correspondente. Conforme foi apontado no Capítulo 4, os navegadores usam HTTP para se comunicar com os servidores web em um endereço IP com um número de porta reservado. Os serviços FTP e SMTP funcionam de maneira semelhante; por exemplo, um programa FTP pode receber o nome de domínio *ftp.dcs.qmul.ac.uk*, fazer uma solicitação de DNS para obter seu endereço IP e depois usar TCP para se comunicar com ele no número de porta reservado. Os nomes *www*, *ftp* e *smtp* podem ser *aliases* dos nomes de domínio dos computadores reais em que esses serviços são executados. Para um exemplo sem *alias*, considere o caso em que um usuário recebe um programa *telnet* para entrar em contato com um *host* cujo nome de domínio é *jeans-pc.dcs.qmul.ac.uk*; o programa *telnet* faz uma solicitação de DNS para obter o endereço IP correspondente e usa o número de porta padrão.

Localização de servidores de correio eletrônico: o software de correio eletrônico usa o DNS para resolver nomes de domínio nos endereços IP de servidores de correio eletrônico – computadores que aceitam correspondência para esses domínios. Por exemplo, quando o endereço *tom@dcs.rnx.ac.uk* precisa ser resolvido, o DNS é consultado com o endereço *dcs.rnx.ac.uk* e a designação de tipo *mail*. Ele retorna uma lista de nomes de domínio de servidores que podem aceitar correspondência de *dcs.rnx.ac.uk*, se existir (e, opcionalmente, os endereços IP correspondentes). O DNS pode retornar mais de um nome de domínio para que o software de correio possa tentar alternativas, caso o servidor principal de correio eletrônico esteja inacessível por algum motivo. O DNS retorna um valor de preferência (inteiro) para cada servidor de correio, indicando a ordem em que os servidores de correio eletrônicos devem ser tentados.

Outros tipos de consulta são implementadas em algumas instalações, mas são consideravelmente menos utilizadas do que as que acabamos de mencionar:

Resolução reversa: alguns programas de software exigem que, dado um endereço IP, um nome de domínio seja retornado. Isso é apenas o inverso da consulta de nome de *host* normal, mas o servidor de nomes que recebe a consulta só responde se o endereço IP estiver em seu próprio domínio.

Informações sobre hosts: o DNS pode armazenar o tipo de arquitetura da máquina e o sistema operacional relacionados aos nomes de domínio dos *hosts*. Foi sugerido que essa opção não seja implementada, pois fornece informações úteis para quem está tentando obter acesso não autorizado aos computadores.

Serviços conhecidos: dado o nome de domínio do computador, uma lista dos serviços executados por um computador (por exemplo, telnet, FTP) e o protocolo usado para obtê-los (isto é, UDP ou TCP, na Internet) podem ser retornados. Isso é possível desde que o servidor de nome suporte essa informação.

Em princípio, o DNS pode ser usado para armazenar atributos arbitrários. Uma consulta é especificada por um nome de domínio, classe e tipo. Para nomes de domínio na Internet, a classe é IN. O tipo de consulta especifica se um endereço IP, um servidor de correio eletrônico, um servidor de nome ou algum outro tipo de informação é exigida. Existe um domínio especial, *in-addr.arpa*, para conter endereços IP para as pesquisas reversas. O atributo da classe é usado para distinguir, por exemplo, o banco de dados de atribuição de nomes da Internet de outros bancos de dados de atribuição de nomes DNS experimentais. É definido um conjunto de tipos para determinado banco de dados; os que servem para banco de dados da Internet aparecem na Figura 9.5.

Servidores de nome DNS ◊ Os problemas de escalabilidade são tratados por uma combinação do particionamento do banco de dados de atribuição de nomes e a replicação e armazenamento em cache de partes dele, próximo dos pontos onde ele é acessado. O banco de dados DNS é distribuído em uma rede lógica de servidores. Cada servidor contém parte do banco de dados de atribuição de nomes – principalmente dados do domínio local. A maioria das consultas diz respeito aos computadores do domínio local e são atendidas pelos servidores de dentro desse domínio. Entretanto, cada servidor registra os nomes de domínio e endereços de outros servidores de nome para que as consultas pertinentes a objetos de fora do domínio possam ser atendidas.

Os dados de atribuição de nomes DNS são divididos em zonas. Uma zona contém os seguintes dados:

- Dados de atributo de nomes em um domínio, menos os subdomínios administrados por autoridades de nível mais baixo. Por exemplo, uma zona poderia conter dados do Queen Mary College, Universidade de Londres – *qmul.ac.uk* – menos os dados mantidos pelos departamentos, por exemplo, o Departamento da Ciência da Computação – *dcs.qmul.ac.uk*.
- Os nomes e endereços de pelo menos dois servidores de nome que possuem autoridade sobre dados da zona. São versões de dados de zona que podem ser considerados razoavelmente atualizados.
- Os nomes de servidores de nome que contêm autoridade sobre dados de subdomínios delegados; e dados "de cola", fornecendo os endereços IP desses servidores.
- Parâmetros de gerenciamento de zona, como aqueles que governam o uso da cache e a replicação de dados de zona.

Um servidor pode ter autoridade sobre dados de zero ou mais zonas. Para que os dados de atribuição de nomes estejam disponíveis, mesmo quando um único servidor falha, a arquitetura do DNS especifica que cada zona deve ser replicada de forma que pelo menos dois servidores sejam autoridades sobre seus dados.

Os administradores de sistema inserem os dados de uma zona em um arquivo mestre, o qual é a origem a autoridade de dados da zona. Existem dois tipos de servidor que são considerados como autoridade de dados. Um *servidor principal,* ou *mestre,* lê dados de zona diretamente de um arquivo mestre local. Os *servidores secundários* fazem o *download* dos dados de zona de um servidor principal. Estes se comunicam periodicamente com o servidor principal para verificar se sua versão armazenada corresponde àquela mantida pelo servidor principal. Se a cópia de um secundário estiver desatualizada, o principal enviará a ele a versão mais recente. A freqüência da verificação do secundário é configurada pelos administradores como um parâmetro da zona e seu valor normalmente é uma ou duas vezes por dia.

Qualquer servidor está livre para armazenar em *cache* dados de outros servidores, para evitar a necessidade de entrar em contato com eles, quando a resolução de nomes exigir os mesmos dados novamente. Nesse caso, o servidor indica nas respostas enviadas aos clientes, quando fornecidas, de que ele é um servidor não-autoridade sobre os dados. Cada entrada em uma zona tem um valor de tempo de vida. Quando um servidor não-autoridade armazena dados na cache provenientes de um servidor com autoridade, ele anota o tempo de vida. Ele só fornecerá para os clientes os dados de sua cache durante esse tempo; quando consultado após o período de tempo ter expirado, ele entra em contato novamente com o servidor autorizado para verificar seus dados. Essa é uma característica útil que minimiza o volume do tráfego da rede, enquanto mantém a flexibilidade para os administradores de sistema. Quando a expectativa é de que os atributos sejam raramente alterados, eles podem receber um tempo de vida correspondentemente grande. Se um administrador souber que os atributos provavelmente mudarão logo, então poderá reduzir o tempo de vida de acordo com isso.

A Figura 9.4 mostra a organização de parte do banco de dados DNS, conforme se encontrava no ano de 2001. Note que, na prática, os servidores raízes, como *a.root-servers.net*, contêm entradas para vários níveis de domínio, assim como entradas para nomes de domínio de primeiro nível. Isso serve para reduzir o número de etapas de navegação, quando os nomes de domínio são resolvidos. Os servidores de nome raiz mantêm entradas com autoridade para os servidores de nome dos domínios de nível superior (*top-level domain*). Eles também são servidores de nome com autoridade para os do-

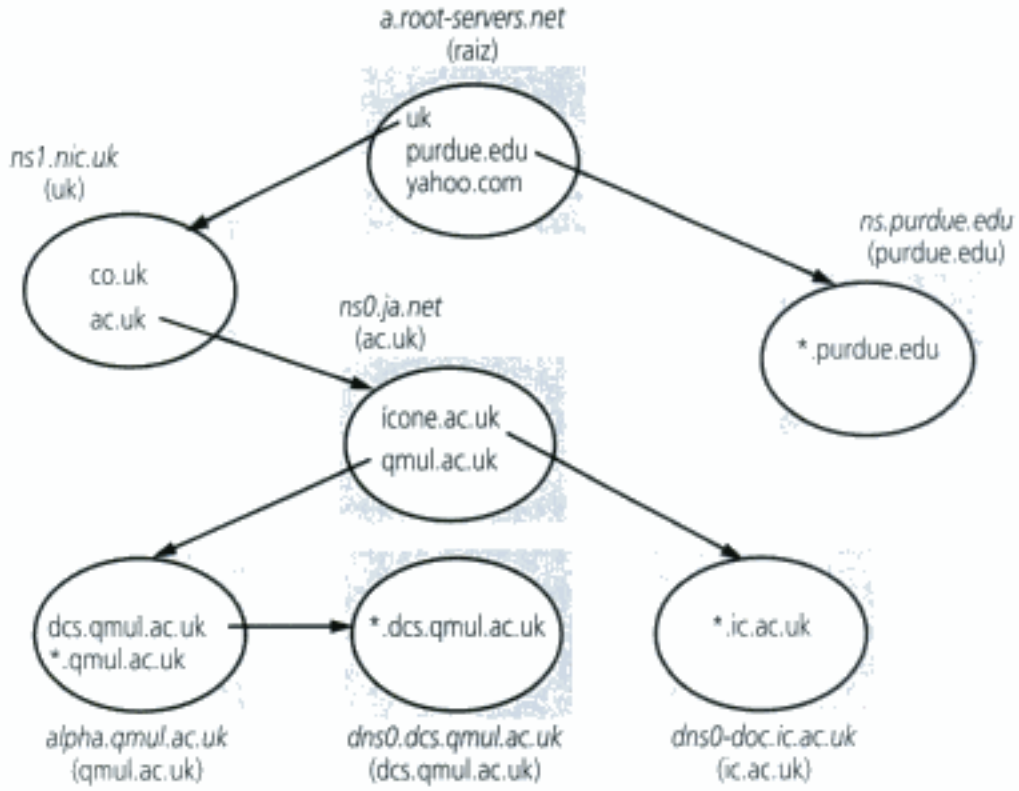


Figura 9.4 **Servidores de nome DNS (conforme se encontravam no ano de 2001).**
Nota: Os nomes de servidor de nomes aparecem em itálico e os domínios correspondentes, entre parênteses. As setas denotam entradas no servidor de nome.

mínios de nível superior genéricos, como *com* e *edu*. Entretanto, os servidores de nome raízes não são servidores de nome dos domínios de país. Por exemplo, o domínio *uk* tem um conjunto de servidores de nome, um dos quais é chamado *ns1.nic.net*. Esses servidores de nome conhecem os servidores de nome dos domínios de segundo nível no Reino Unido, como *ac.uk* e *co.uk*. Os servidores de nome do domínio *ac.uk* conhecem os servidores de nome de todos os domínios de universidade daquele país, como *qmul.ac.uk* ou *ic.ac.uk*. Em alguns casos, um domínio de universidade delega parte de suas responsabilidades para um subdomínio, como *dcs.qmul.ac.uk*.

Tipo de registro	*Significado*	*Conteúdo principal*
A	Endereço de computador	Número IP
NS	Servidor de nome com autoridade	Nome de domínio do servidor
CNAME	Nome canônico de um *alias*	Nome de domínio do *alias*
SOA	Marca o início dos dados de uma zona	Parâmetros que governam a zona
WKS	Descrição de um serviço conhecido	Lista de nomes de serviço e protocolos
PTR	Ponteiro de nome de domínio (pesquisas reversas)	Nome de domínio
HINFO	Informações de *host*	Arquitetura de máquina e sistema operacional
MX	Responsável pelo correio eletrônico	Lista de pares *<preferência, host>*
TXT	String de texto	Texto arbitrário

Figura 9.5 **Registros de recurso DNS.**

As informações do domínio raiz são replicadas por um servidor principal em um conjunto de servidores secundários, conforme descrito anteriormente. Apesar disso, alguns servidores raízes precisam atender a cerca de 1000 consultas por segundo, de acordo com Liu e Albitz [1998]. Todos os servidores de DNS armazenam os endereços de um ou mais servidores de nome raízes, os quais não mudam com muita freqüência. Normalmente, eles também armazenam o endereço de um servidor autorizado do domínio pai. Uma consulta envolvendo um nome de domínio de três componentes, como *www.berkeley.edu*, pode ser atendida usando-se na pior das hipóteses duas etapas de navegação: uma para um servidor raiz que armazena uma entrada de servidor de nome apropriada e uma segunda para o servidor cujo nome é retornado.

Com referência à Figura 9.4, o nome de domínio *jeans-pc.dcs.qmul.ac.uk* pode ser pesquisado dentro de *dcs.qmul.ac.uk*, usando-se o servidor local *dns0.dcs.qmul.ac.uk*. Esse servidor não armazena uma entrada para o servidor web *www.ic.ac.uk*, mas mantém uma entrada na cache para *ic.ac.uk* (que é obtida do servidor com autoridade *ns0.ja.net*). O servidor *dns0-doc.ic.ac.uk* pode ser contatado para transformar o nome completo.

Navegação e processamento de consulta ◊ Um cliente de DNS é chamado de *resolvedor*. Normalmente, ele é implementado como software de biblioteca. Ele aceita consultas, as formata nas mensagens esperadas no protocolo DNS e se comunica com um ou mais servidores de nome para atender as pesquisas. É usado um protocolo de requisição e resposta simples, normalmente utilizando pacotes UDP na Internet (os servidores de DNS usam um número de porta conhecido). Caso a resposta não venha dentro de certo tempo (*timeout*) a consulta é enviada novamente, se necessário. O resolvedor pode ser configurado para entrar em contato com uma lista de servidores de nome iniciais, em ordem de preferência, para o caso de um ou mais estarem indisponíveis.

A arquitetura do DNS possibilita navegação recursiva, assim como navegação iterativa. O resolvedor especifica qual tipo de navegação é exigido, ao entrar em contato com um servidor de nome. Entretanto, os servidores de nome não são obrigados a implementar navegação recursiva. Conforme foi mencionado anteriormente, a navegação recursiva pode manter ocupadas as *threads* do servidor, significando que outras requisições podem ser retardadas.

Para economizar comunicação na rede, o protocolo DNS permite que várias consultas sejam empacotadas na mesma mensagem de requisição e que os servidores de nome enviem várias respostas correspondentes, em suas mensagens de resposta.

Registros de recurso ◊ Os dados de zona são armazenados pelos servidores de nome em arquivos, em um de vários tipos de registro de recurso. Para o banco de dados da Internet, isso inclui os tipos que aparecem na Figura 9.5. Cada registro se refere a um nome de domínio, o qual não é mostrado. As entradas da tabela se referem aos itens já mencionados, exceto que as entradas *TXT* foram incluídas para permitir que outras informações arbitrárias sejam armazenadas com relação aos nomes de domínio. As tabelas a seguir são baseadas nos dados existentes em 2001.

Os dados de uma zona começam com um registro de tipo *SOA*, o qual contém os parâmetros de zona que especificam, por exemplo, o número da versão e com que freqüência os servidores secundários devem atualizar suas cópias. Isso é seguido por uma lista de registros de tipo *NS*, especificando os servidores de nome do domínio, e por uma lista de registros de tipo *MX*, fornecendo as preferências e os nomes de domínio dos *hosts* de correio eletrônicos. Por exemplo, parte do banco de dados do domínio *dcs.qmul.ac.uk* aparece na Figura 9.6, onde o tempo de vida *1D* significa 1 dia.

Na seqüência do banco de dados, os registros de tipo *A* fornecem os endereços IP dos dois servidores de nome *dns0* e *dns1*. Os endereços IP dos *hosts* de correio eletrônico e do terceiro servidor de nome são dados no banco de dados correspondente aos seus domínios.

A maior parte dos registros em uma zona de nível inferior, como *dcs.qmul.ac.uk*, será de tipo *A* e farão o mapeamento do nome de domínio de um computador para seu endereço IP. Eles podem conter alguns *alias* para os serviços conhecidos, por exemplo:

nome de domínio	*tempo de vida*	*classe*	*tipo*	*valor*
www	1D	IN	CNAME	apricot
apricot	1D	IN	A	138.37.88.248

nome de domínio	*tempo de vida*	*classe*	*tipo*	*valor*
	1D	IN	NS	dns0
	1D	IN	NS	dns1
	1D	IN	NS	cancer.ucs.ed.ac.uk
	1D	IN	MX	1 mail1.qmul.ac.uk
	1D	IN	MX	2 mail2.qmul.ac.uk

Figura 9.6 Registros de dados de zona DNS.

Se o domínio tiver subdomínios, existirão mais registros de tipo *NS* especificando seus servidores de nome, os quais também terão entradas *A* individuais. Por exemplo, o banco de dados de *qmul.ac.uk* contém os seguintes registros para os servidores de nome em seu subdomínio *dcs.qmul.ac.uk*:

nome de domínio	*tempo de vida*	*classe*	*tipo*	*valor*
dcs	1D	IN	NS	dns0.dcs
dns0.dcs	1D	IN	A	138.37.88.249
dcs	1D	IN	NS	dns1.dcs
dns1.dcs	1D	IN	A	138.37.94.248
dcs	1D	IN	NS	cancer.ucs.ed.ac.uk

Compartilhamento de carga de servidores de nome: em alguns sites, os serviços muito utilizados, como web e FTP, são suportados por um grupo de computadores na mesma rede. Nesse caso, o mesmo nome de domínio é usado para cada membro do grupo. Quando um nome de domínio é compartilhado por vários computadores, existe um registro para cada computador do grupo fornecendo seu endereço IP. O servidor de nome responde às consultas relacionadas a vários registros com o mesmo nome, retornando os endereços IP de acordo com um programa de rodízio. Sucessivos clientes recebem acesso a diferentes servidores para que estes possam compartilhar a carga de trabalho. O uso da cache tem o potencial de estragar esse esquema, pois uma vez que um servidor de nome não-autoridade, ou um cliente, tenha o endereço do servidor em sua cache, ele continuará a usá-lo. Para neutralizar esse efeito, os registros recebem um tempo de vida curto.

A implementação BIND do DNS ◊ O BIND (*Berkeley Internet Name Domain*) é uma implementação do DNS para computadores que executam UNIX. Os programas clientes que executam resolução de nomes devem ser ligados a essa biblioteca. Os computadores servidores de nome DNS executam o *daemon* denominado de *named*.

O BIND permite três categorias de servidor de nome: servidores principais, servidores secundários e servidores somente de cache; o *named* implementa apenas um desses tipos, de acordo com o conteúdo de um arquivo de configuração. As duas primeiras categorias são conforme descrito anteriormente. Os servidores somente de cache lêem, em um arquivo de configuração, nomes e endereços de servidores com autoridades suficientes para resolver qualquer nome. Depois disso, eles apenas armazenam esses dados e os dados que aprendem resolvendo nomes para clientes.

Uma organização típica tem um servidor principal, com um ou mais servidores secundários, que fornecem nomes atendendo diferentes redes locais no site. Além disso, computadores individuais freqüentemente executam seus próprios servidores somente de cache para reduzir o tráfego da rede e acelerar ainda mais os tempos de resposta.

Discussão sobre o DNS ◊ A implementação DNS na Internet atinge tempos de resposta médios relativamente curtos para pesquisas, considerando o volume de dados de atribuição de nomes e a escala das redes envolvidas. Vimos que ela obtém isso por meio de uma combinação de particionamento, replicação e uso de cache para os dados de atribuição de nomes. Os objetos nomeados são principal-

mente computadores, servidores de nome e *hosts* de correio eletrônico. Os mapeamentos de nome de computador (*host*) para endereço IP mudam de forma relativamente rara, assim como as identidades dos servidores de nome e de correio eletrônico; portanto, o uso de cache e a replicação ocorrem em um ambiente relativamente estável.

O DNS permite que os dados de atribuição de nomes se tornem inconsistentes. Isto é, se os dados de atribuição de nomes forem alterados, outros servidores poderão fornecer dados antigos aos clientes por períodos na ordem de dias. Nenhuma das técnicas de replicação exploradas no Capítulo 15 é aplicada. Entretanto, a inconsistência não tem nenhuma conseqüência até o momento em que um cliente tenta utilizar dados antigos. O DNS não trata por si mesmo do modo como os endereços antigos são detectados.

Além dos computadores, o DNS também nomeia um tipo de serviço em particular: o serviço de correio eletrônico, um para cada domínio. O DNS presume que há apenas um serviço de correio eletrônico por domínio endereçado; portanto, os usuários não precisam incluir o nome desse serviço explicitamente. Os aplicativos de correio eletrônico selecionam esse serviço de forma transparente, usando o tipo de consulta apropriado ao entrar em contato com os servidores de DNS.

Em resumo, o DNS armazena uma variedade limitada de dados de atribuição de nomes, mas isso é suficiente, na medida em que os aplicativos, como o correio eletrônico, impõem seus próprios esquemas de atribuição de nomes sobre os nomes de domínio. Poderia ser argumentado que o banco de dados DNS representa o mínimo denominador comum do que seria considerado útil por muitas comunidades de usuário na Internet. O DNS não foi projetado para ser o único serviço de nomes da Internet; ele coexiste com serviços de diretório e de nome locais que armazenam os dados mais pertinentes às necessidades locais (como o Sun Network Information Service, que armazena senhas codificadas, por exemplo, ou o Active Directory Service da Microsoft [www.microsoft.com I], que armazena informações detalhadas sobre todos os recursos dentro de um domínio).

O que permanece como um problema em potencial para o projeto DNS é sua rigidez com relação às alterações na estrutura do espaço de nomes e a falta de capacidade de personalizar o espaço de nomes de acordo com as necessidades locais. Esses aspectos do projeto de atribuição de nomes são considerados pelo estudo de caso do serviço de nomes global, na Seção 9.4. Antes disso, consideraremos os serviços de diretório.

9.3 Serviços de diretório

Descrevemos como os serviços de nome armazenam conjuntos de pares <*nome, atributo*> e como os atributos são pesquisados a partir de um nome. É natural considerar a dualidade dessa organização, na qual os *atributos* são usados como os valores a serem pesquisados. Nesses serviços, os nomes textuais podem ser considerados apenas como outro atributo. Às vezes, os usuários desejam encontrar uma pessoa, ou um recurso em particular, mas não sabem seu nome, apenas alguns de seus outros atributos. Por exemplo, um usuário pode perguntar: "qual é o nome do usuário com número de telefone 020-555 9980?". Às vezes, os usuários exigem um serviço, mas não estão preocupados com qual entidade do sistema fornece esse serviço, desde que ele esteja convenientemente acessível. Por exemplo, um usuário poderia perguntar "quais computadores neste prédio são Macintosh executando o sistema operacional Mac OS X?" ou "onde posso imprimir uma imagem colorida de alta-resolução?".

Um serviço que armazena conjuntos de vínculos entre nomes e atributos e que pesquisa entradas que correspondem a especificações baseadas no atributo é chamado de *serviço de diretório*. Exemplos são o Active Directory Services da Microsoft, o X.500 e seu primo LDAP (descrito na Seção 9.5), o Univers [Bowman *et al.* 1990] e o Profile [Peterson 1988]. Às vezes, os serviços de diretório são chamados de *serviços de páginas amarelas* e os serviços de nomes convencionais são chamados, correspondentemente, de *serviços de páginas brancas*, em uma analogia óbvia com os diferentes tipos de catálogos telefônicos. Às vezes, os serviços de diretório são conhecidos como *serviços de nomes baseados em atributo*.

Um serviço de diretório retorna os conjuntos de atributos de todos os objetos encontrados que correspondam a alguns atributos especificados. Portanto, o pedido 'TelephoneNumber = 020-555 9980' poderia retornar { 'Name = John Smith', 'TelephoneNumber = 020-555 9980', 'emailAddress = john@

dcs.gormenghast.ac.uk', ...}, por exemplo. O cliente pode especificar que apenas um subconjunto dos atributos tem interesse – por exemplo, apenas os endereços de e-mail dos objetos correspondentes. O X.500, e alguns outros serviços de diretório, também permitem que objetos sejam pesquisados por nomes textuais hierárquicos convencionais. O UDDI (*Universal Directory and Discovery Service*), que será apresentado na Seção 19.4, apresenta serviços de páginas brancas e de páginas amarelas para fornecer informações sobre organizações e os serviços web de que dispõe.

UDDI à parte, o termo *serviço de descoberta* normalmente denota o caso especial de um serviço de diretório para serviços fornecidos por dispositivos em um ambiente de interligação em rede espontânea. Conforme a Seção 2.2.3 descreveu, os dispositivos nas redes espontâneas estão sujeitos a se conectar e desconectar de forma imprevisível. Uma diferença fundamental entre um serviço de descoberta e serviços de diretório é que o endereço de um serviço de diretório normalmente é conhecido e previamente configurado nos clientes, enquanto um dispositivo que entre em um ambiente de interligação em rede espontânea precisa contar com a navegação por *multicast*, pelo menos na primeira vez que acessa o serviço de descoberta local. A Seção 16.2 descreverá os serviços de descoberta em detalhes.

Os atributos são claramente mais poderosos do que os nomes, como designadores de objetos: podem ser escritos programas para selecionar objetos de acordo com especificações de atributo precisas, onde os nomes podem não ser conhecidos. Outra vantagem dos atributos é que eles não expõem a estrutura das organizações para o mundo exterior, como acontece com os nomes particionados em termos de organização. Entretanto, a relativa simplicidade de uso dos nomes textuais torna improvável que eles sejam substituídos pela atribuição de nomes baseada em atributo, em muitas aplicações.

9.4 Estudo de caso: Global Name Service

O GNS (Global Name Service) foi projetado e implementado por Lampson e colegas, no DEC Systems Research Center [Lampson 1986], para fornecer facilidades para a localização de recursos, endereços de e-mail e autenticação. Os objetivos de projeto do GNS já foram listados no final da Seção 9.1; eles refletem o fato de que um serviço de nomes para uso em uma rede interligada deve suportar um banco de dados de atribuição de nomes que possa ser ampliado para incluir os nomes de milhões de computadores e (eventualmente) endereços de e-mail de bilhões de usuários. Os projetistas do GNS também consideraram que o banco de dados de atribuição de nomes provavelmente terá um longo tempo de vida, e que ele deve continuar a operar eficientemente, enquanto cresce de pequena para grande escala, e a rede em que é baseado também evolui. A estrutura do espaço de nomes pode mudar durante esse tempo, para refletir alterações nas estruturas organizacionais. O serviço deve acomodar mudanças nos nomes dos indivíduos, das organizações e dos grupos que contém, e as alterações na estrutura de atribuição de nomes, como aquelas que ocorrem quando uma empresa é assumida por outra. Nesta descrição, vamos focalizar as características do projeto que permitem acomodar tais alterações.

O banco de dados de atribuição de nomes potencialmente grande e a escala do ambiente distribuído em que o GNS é feito para operar tornam fundamental o uso de cache. Entretanto, fica extremamente difícil manter a consistência completa entre todas as cópias de uma entrada do banco de dados. A estratégia de consistência de cache adotada conta com a suposição de que as atualizações no banco de dados serão raras e que uma disseminação lenta das atualizações é aceitável, pois os clientes podem detectar e se recuperar do uso de dados de atribuição de nomes desatualizados.

O GNS gerencia um banco de dados de atribuição de nomes composto de uma árvore de diretórios contendo nomes e valores. Os diretórios são nomeados por nomes de caminho de várias partes que se referem a uma raiz, ou relativos a um diretório de trabalho, de forma muito parecida com os nomes de arquivo em um sistema de arquivos UNIX. Cada diretório também recebe um valor inteiro, que serve como *identificador de diretório* (DI) exclusivo. Nesta seção, usaremos nomes em itálico ao nos referirmos ao DI de um diretório, de modo que *EC* é o identificador do diretório EC. Um diretório contém uma lista de nomes e referências. Os valores armazenados nas folhas da árvore de diretório são organizados em *árvores de valores*, para que os atributos associados aos nomes possam ser valores estruturados.

No GNS, os nomes têm duas partes: <*nome do diretório, nome do valor*>. A primeira parte identifica um diretório; a segunda se refere a uma árvore de valores, ou a alguma parte de uma árvore de valores. Por exemplo, veja a Figura 9.7, na qual os DIs estão ilustrados como valores inteiros pequenos, embora na verdade eles sejam escolhidos em um intervalo de valores inteiros maiores para garantir a exclusividade. Os atributos de um usuário Peter.Smith no diretório QMUL seriam armazenados na árvore de valores nomeada <*EC*/UK/AC/QMUL, Peter.Smith>. A árvore de valores inclui uma senha, a qual pode ser referenciada como <*EC*/UK/AC/QMUL, Peter.Smith/senha>, e vários endereços de correspondência, cada um dos quais seria listado na árvore de valores como um único nó, com o nome <*EC*/UK/AC/QMUL, Peter.Smith/caixas de correio>.

A árvore de diretório é particionada e armazenada em muitos servidores, com cada partição replicada em vários servidores. A consistência da árvore é mantida em face de duas ou mais atualizações concorrentes – por exemplo, dois usuários podem tentar criar, simultaneamente, entradas com o mesmo nome e apenas um deve ter êxito. Os diretórios replicados apresentam um segundo problema de consistência; este é tratado por um algoritmo de distribuição de atualização assíncrono que garante a consistência final, mas sem garantia de que todas as cópias sejam sempre atuais. Esse nível de consistência é considerado satisfatório para o objetivo.

Acomodando alterações ◊ Veremos agora os aspectos do projeto relacionados ao ajuste do crescimento e da mudança na estrutura do banco de dados de atribuição de nomes. No nível dos clientes e administradores, o crescimento é acomodado por meio da ampliação da árvore de diretório da maneira usual. Mas podemos querer integrar as árvores de atribuição de nomes de dois serviços GNS anteriormente separados. Por exemplo, como poderíamos integrar o banco de dados cuja raiz está no diretório *EC* mostrado na Figura 9.7, com outro banco de dados de *AMÉRICA DO NORTE*? A Figura 9.8 mostra uma nova raiz *MUNDO*, introduzida acima das raízes existentes das duas árvores a serem integradas. Essa é uma técnica simples, mas como ela afeta os clientes que continuam a usar nomes que são referenciados ao que era "a raiz", antes que a integração ocorresse? Por exemplo, </UK/AC/QMUL, Peter.Smith> era um nome usado pelos clientes antes da integração. Trata-se de um nome absoluto (pois começa com o símbolo da raiz '/'), mas a raiz a que ele se refere é *EC* e não *MUNDO*. *EC* e *AMÉRICA DO NORTE* são *raízes de trabalho* – contextos iniciais nos quais os nomes que começam com a raiz '/' devem ser pesquisados.

A existência de identificadores de diretório exclusivos pode ser usada para resolver esse problema. A raiz de trabalho de cada programa deve ser identificada como uma parte de seu ambiente de execução (de forma muito parecida com o que é feito para o diretório de trabalho de um programa). Quando um

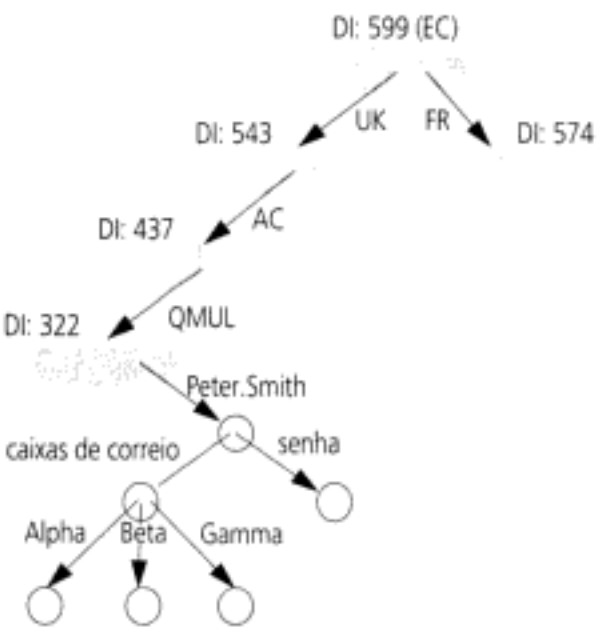


Figura 9.7 Árvore de diretório GNS e árvore de valores do usuário Peter.Smith.

Figura 9.8 Integrando árvores sob uma nova raiz.

cliente na Comunidade Européia usa um nome da forma </UK/AC/QMUL, Peter.Smith>, seu agente de usuário local, que sabe da existência da raiz de trabalho, prefixa o identificador de diretório EC (599), produzindo assim o nome <599/UK/AC/QMUL, Peter.Smith>. O agente de usuário passa esse nome derivado no pedido de pesquisa para um servidor GNS. O agente de usuário pode tratar de modo semelhante com nomes relativos que se referem a diretórios de trabalho. Os clientes que sabem da existência da nova configuração também podem fornecer nomes absolutos para o servidor GNS, os quais se referem ao diretório conceitual super-raiz, contendo todos os identificadores de diretório; por exemplo, <*MUNDO*/EC/UK/AC/QMUL, Peter.Smith>, mas o projeto não pode supor que todos os clientes serão atualizados, para levar em conta tal alteração.

A técnica descrita anteriormente resolve o problema lógico, permitindo que usuários e programas clientes continuem a usar nomes definidos em relação a uma raiz antiga, mesmo quando uma nova raiz real é inserida, mas ela deixa um problema de implementação: em um banco de dados de atribuição de nomes distribuído que pode conter milhões de diretórios, como o serviço GNS pode localizar um diretório, dado apenas seu identificador, como 599? A solução adotada pelo GNS é listar os diretórios usados como raízes de trabalho, como EC, em uma tabela de "diretórios bem-conhecidos" mantida no diretório-raiz real corrente do banco de dados de atribuição de nomes. Quando a raiz real do banco de dados de atribuição de nomes muda, como na Figura 9.8, todos os servidores GNS são informados da nova localização da raiz real. Então, eles podem interpretar normalmente nomes da forma MUNDO/EC/UK/AC/QMUL (referido à raiz real) e interpretar nomes da forma #599/UK/AC/QMUL usando a tabela de "diretórios bem-conhecidos" para transformá-los nos nomes de caminho completos que começam na raiz real.

O GNS também suporta a reestruturação do banco de dados para acomodar mudança organizacional. Suponha que os Estados Unidos se tornem parte da Comunidade Européia(!). A Figura 9.9 mostra a nova árvore de diretório. Mas se a subárvore US for simplesmente movida para o diretório EC, os nomes que começam com *MUNDO*/AMÉRICA DO NORTE/US não funcionarão mais. A solução adotada pelo GNS foi inserir um "vínculo simbólico" no lugar da entrada US original (mostrada em

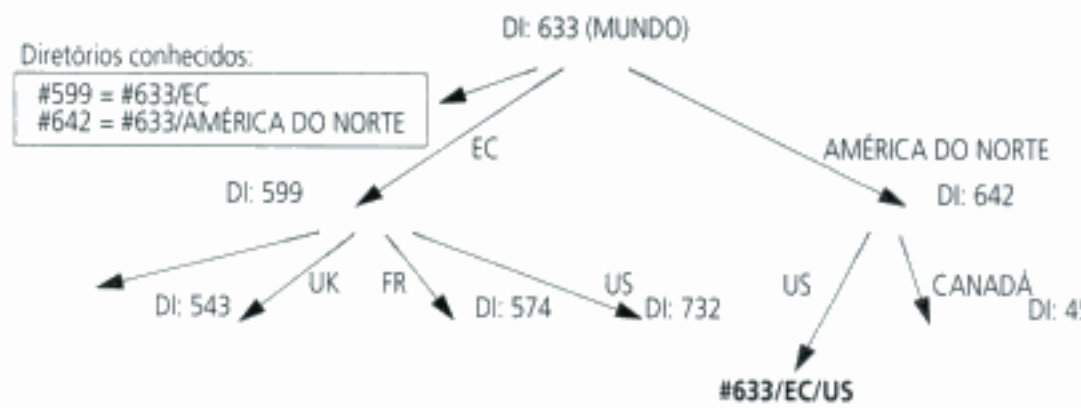


Figura 9.9 Reestruturação do diretório.

negrito na Figura 9.9). O procedimento de pesquisa de diretório do GNS interpreta o vínculo como um redirecionamento para o diretório US em sua nova localização.

Discussão sobre o GNS ◊ O GNS é descendente do Grapevine [Birrell *et al.* 1982] e do Clearinghouse [Oppen e Dalal 1983], dois sistemas de atribuição de nomes bem-sucedidos, desenvolvidos pela Xerox Corporation, principalmente para propósitos de distribuição de e-mail. O GNS trata com êxito das necessidades de capacidade de mudança de escala e reconfiguração, mas a solução adotada para integrar e mover árvores de diretório resultou em um requisito de um banco de dados (a tabela de diretórios bem-conhecidos) que deve ser replicado em cada nó. Em uma rede de larga escala, as reconfigurações podem ocorrer em qualquer nível e essa tabela poderia crescer bastante, entrando em conflito com o objetivo da capacidade de mudança de escala.

9.5 Estudo de caso: X.500 Directory Service

O X.500 é um serviço de diretório no sentido definido na Seção 9.3. Ele pode ser usado da mesma maneira que um serviço de nomes convencional, mas é utilizado principalmente para atender consultas descritivas, projetadas para descobrir os nomes e atributos de outros usuários ou recursos de sistema. Os usuários podem usar uma variedade de requisitos de busca e navegação em um diretório de usuários, de organizações e recursos de sistema para obter informações sobre as entidades contidas nesse diretório. É provável que os usos de tal serviço sejam bastante diversificados. Eles variam desde solicitações diretamente análogas ao uso de catálogos telefônicos, como um acesso simples às "páginas brancas" para obter o endereço de correio eletrônico de um usuário, ou uma consulta de "páginas amarelas" destinada, por exemplo, a obter os nomes e números de telefone de oficinas especializadas no reparo de uma marca de carro em particular, até o uso do diretório para acessar detalhes pessoais, como funções de trabalho, hábitos dietéticos, ou mesmo fotografias das pessoas.

Tais consultas podem se originar a partir de usuários, como o caso das "páginas amarelas" exemplificado pela pergunta sobre oficinas mencionada anteriormente, ou de processos, quando elas podem ser usadas para identificar serviços para atender um determinado requisito funcional.

Indivíduos e organizações podem usar um serviço de diretório para tornar disponível uma ampla variedade de informações sobre si mesmos e sobre os recursos que desejam oferecer para uso na rede. Os usuários podem pesquisar o diretório em busca de informações específicas com conhecimento apenas parcial de seu nome, estrutura ou conteúdo.

As organizações de padronização ITU e ISO definiram o *X.500 Directory Service* [ITU/ISO 1997] como um serviço de rede destinado a atender esses requisitos. O padrão se refere a ele como um serviço para acessar informações sobre "entidades do mundo real", mas provavelmente também pode ser usado para acessar informações sobre serviços de hardware e software e dispositivos. O X.500 é especificado como um serviço em nível de aplicação no conjunto de padrões OSI (Open Systems Interconnection), mas seu projeto não depende significativamente dos outros padrões OSI e pode ser visto como um projeto de um serviço de diretório de propósito geral. Aqui, vamos descrever em linhas gerais o projeto do serviço de diretório X.500 e sua implementação. Os leitores que estiverem interessados em uma descrição mais detalhada do X.500 e dos métodos para sua implementação devem estudar o livro sobre o assunto [Rose 1992]. O X.500 também é a base do LDAP (discutido a seguir); e é usado no serviço de diretório DCE [OSF 1997].

Os dados armazenados nos servidores X.500 são organizados em uma estrutura em árvore com nós nomeados, como no caso dos outros servidores de nome discutidos neste capítulo, mas no X.500, uma grande variedade de atributos é armazenada em cada nó da árvore e o acesso não é apenas pelo nome, mas também pela busca de entradas com qualquer combinação de atributos exigida.

A árvore de nomes do X.500 é chamada de DIT (*Directory Information Tree*) e a estrutura de diretório inteira, incluindo os dados associados aos nós, é chamada de DIB (*Directory Information Base*). Ela se destina a ser uma única DIB integrada, contendo informações fornecidas por organizações de todo o mundo, com partes da DIB localizadas em servidores X.500 individuais. Normalmente, uma organização grande, ou de tamanho médio, forneceria pelo menos um servidor. Os clientes acessam o diretório estabelecendo uma conexão com um servidor e emitindo requisições de acesso. Os

clientes podem entrar em contato com qualquer servidor com uma solicitação. Se os dados exigidos não estiverem no segmento da DIB mantida pelo servidor contatado, ele invocará outros servidores para solucionar a consulta ou redirecionará o cliente para outro servidor.

Na terminologia do padrão X.500, os servidores são DSAs (*Directory Service Agents* – agentes de serviço de diretório) e seus clientes se chamam DUAs (*Directory User Agents* – agentes de usuário de diretório). A Figura 9.10 mostra a arquitetura de software e um dos vários modelos de navegação possíveis, com cada processo cliente DUA interagindo com um único processo DSA, o qual acessa outros DSAs, conforme for necessário, para atender os pedidos.

Cada entrada na DIB consiste em um nome e um conjunto de atributos. Assim como nos outros servidores de nome, o nome completo de uma entrada corresponde a um caminho pela DIT, da raiz da árvore até a entrada. Além dos nomes completos, ou *absolutos*, um DUA pode estabelecer um contexto, o qual inclui um nó de base, e depois usar nomes relativos mais curtos que forneçam o caminho do nó de base para a entrada nomeada.

A Figura 9.11 mostra a parte da *Directory Information Tree* que inclui a Universidade de Gormenghast, Grã-Bretanha, e a Figura 9.12 é uma das entradas de DIB associadas. A estrutura de dados das entradas na DIB e na DIT é muito flexível. Uma entrada de DIB consiste em um conjunto de atributos, onde um atributo tem um *tipo* e um ou mais *valores*. O tipo de cada atributo é denotado por um nome (por exemplo, *nomePaís*, *nomeOrganização*, *nomeComum*, *númeroTelefone*, *caixaCorreio*, *classeObjeto*). Novos tipos de atributo podem ser definidos, caso sejam exigidos. Para cada nome de tipo distinto existe uma definição de tipo correspondente, a qual inclui uma descrição do tipo e uma definição da sintaxe no ASN.1 Notation (uma notação padrão para definições de sintaxe), definindo representações para todos os valores permitido do tipo.

As entradas de DIB são classificadas de maneira semelhante às estruturas de classe de objeto encontradas nas linguagens de programação orientadas a objetos. Cada entrada inclui um atributo *objectClass*, que determina a classe (ou classes) do objeto a que uma entrada se refere. *Organization*, *organizationalPerson* e *document* são exemplos de valores de *objectClass*. Mais classes podem ser definidas, conforme forem exigidas. A definição de uma classe determina quais atributos são obrigatórios e quais são opcionais para as entradas da classe dada. As definições das classes são organizadas em uma hierarquia de herança na qual todas as classes, exceto uma (chamada *topClass*), devem conter um atributo *objectClass*, e o valor do atributo *objectClass* deve ser o nome de uma ou mais classes. Se existirem diversos valores de *objectClass*, o objeto herdará os atributos obrigatórios e opcionais de cada uma das classes.

O nome de uma entrada de DIB (o nome que determina sua posição na DIT) é determinado pela escolha de um ou mais de seus atributos como *atributos distintos*. Os atributos escolhidos para esse propósito são referidos como DN (*Distinguished Name*) da entrada.

Agora, podemos considerar os métodos pelos quais o diretório é acessado. Existem dois tipos principais de requisições de acesso:

Figura 9.10 Arquitetura do serviço X.500.

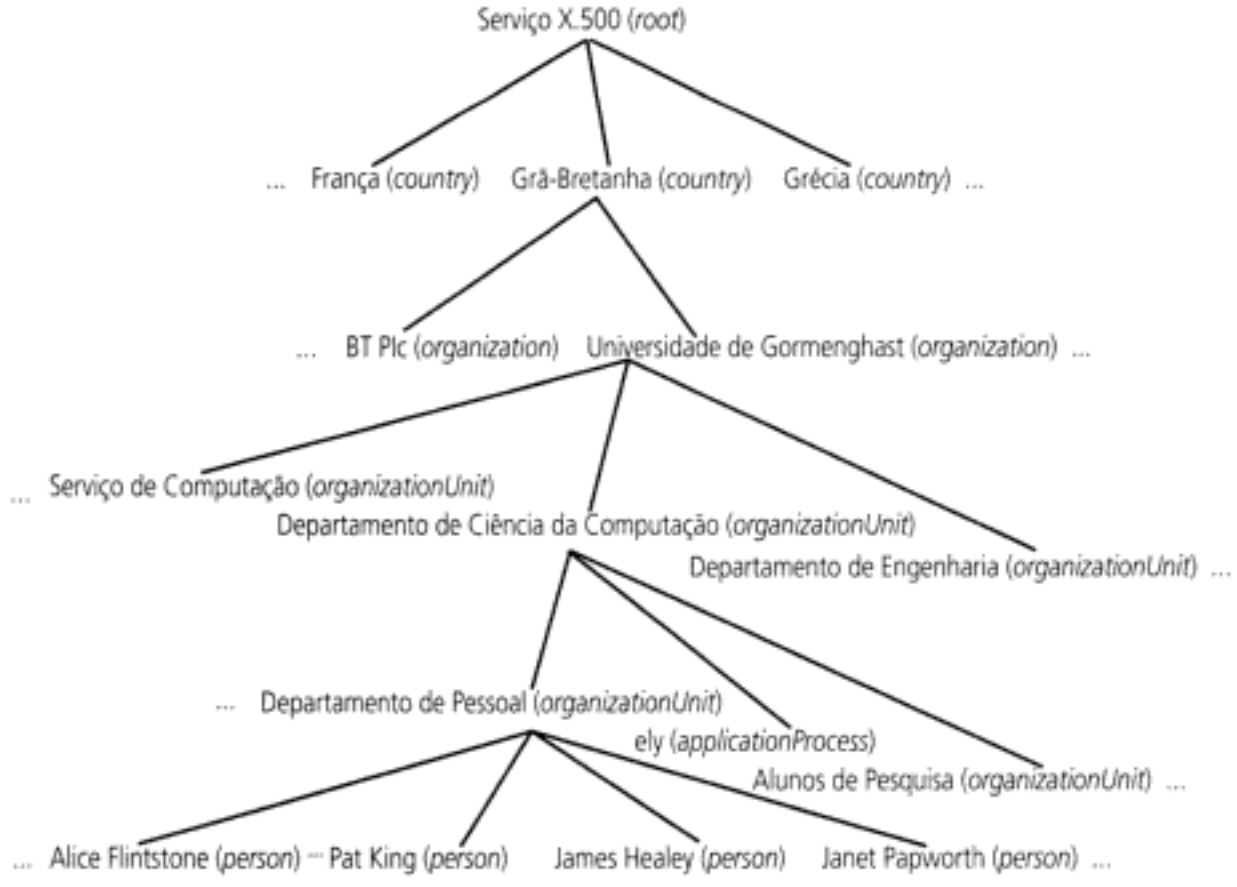


Figura 9.11 **Parte da árvore de informações de diretório (DIT) do X.500.**

read: um nome absoluto ou relativo (um *nome de domínio* na terminologia do X.500) de uma entrada é fornecido, junto com uma lista dos atributos a serem lidos (ou uma indicação de que todos os atributos são exigidos). O DSA localiza a entrada nomeada navegando na DIT, fazendo requisições para outros servidores DSA quando não contém as partes relevantes da árvore. Ele recupera os atributos solicitados e os retorna para o cliente.

search: esta é uma requisição de acesso baseada em atributo. Um nome de base e uma expressão de filtragem são fornecidos como argumentos. O nome de base especifica o nó na DIT a partir do qual a busca deve começar; a expressão de filtragem é uma expressão booleana que vai ser avaliada para cada nó abaixo do nó de base. O filtro especifica um critério de busca: uma combinação lógica de testes nos valores de todos os atributos de uma entrada. O comando *search* retorna uma lista de nomes (nomes de domínio) para todas as entradas abaixo do nó de base para as quais o filtro é avaliado como *VERDADEIRO*.

Por exemplo, poderia ser construído e aplicado um filtro para encontrar os *commonNames* de todos os membros do pessoal que ocupam o quarto Z42 no Departamento de Ciência da Computação da Universidade de Gormenghast (Figura 9.12). Então, uma requisição de leitura poderia ser usada para se obter qualquer um ou todos os atributos dessas entradas de DIB.

A busca pode ser muito dispendiosa quando aplicada em grandes partes da árvore de diretório (a qual pode residir em diversos servidores). Argumentos adicionais podem ser fornecidos para a operação *search*, para restringir a abrangência de sua busca, o tempo durante o qual uma busca pode continuar e o tamanho da lista de entradas retornada.

Administração e atualização da DIB ◊ A interface DSA inclui operações para adicionar, excluir e modificar entradas. O controle de acesso é fornecido tanto para consultas como para operações de atualização; portanto, o acesso a partes da DIT pode ser restrito a certos usuários ou classes de usuário.

info

Alice Flintstone, Departamento de Pessoal, Departamento de Ciência da Computação, Universidade de Gormenghast, GB

commonName
Alice.L.Flintstone
Alice.Flintstone
Alice Flintstone
A. Flintstone

surname
Flintstone

telephoneNumber
+44 986 33 4604

uid
alf

mail
alf@dcs.gormenghast.ac.uk
Alice.Flintstone@dcs.gormenghast.ac.uk

roomNumber
Z42

userClass
Aluno de Pesquisa

Figura 9.12 Uma entrada DIB do X.500.

A DIB é particionada com a expectativa de que cada organização forneça pelo menos um servidor contendo os detalhes das entidades nela presentes. Partes da DIB podem ser replicadas em vários servidores.

Como padrão (ou recomendação, na terminologia do CCITT), o X.500 não trata de problemas de implementação. Entretanto, é bastante claro que qualquer implementação envolvendo vários servidores em uma rede de longa distância deve contar com o uso extensivo de técnicas de replicação e uso de cache, para evitar o redirecionamento demasiado das consultas.

Uma implementação, descrita em Rose [1992], é um sistema desenvolvido no University College, Londres, conhecido como QUIPU [Kille 1991]. Nessa implementação, o uso de cache e a replicação são realizados no nível das entradas de DIB individuais e no nível dos conjuntos de entradas descendentes do mesmo nó. Pressupõe-se que os valores podem se tornar inconsistentes após uma atualização e o intervalo de tempo durante o qual a consistência é restaurada pode ser de vários minutos. Essa forma de disseminação da atualização geralmente é considerada aceitável para aplicações de serviço de diretório.

Lightweight Directory Access Protocol ◊ A interface padrão do X.500 usa um protocolo que envolve as camadas superiores da pilha de protocolos ISO. Um grupo da Universidade de Michigan propôs uma estratégia mais leve, chamada LDAP (*Lightweight Directory Access Protocol*), na qual um DUA acessa serviços de diretório X.500 diretamente sobre TCP/IP. Isso está descrito no RFC 2251 [Wahl *et al.* 1997]. O LDAP simplifica a interface do X.500 de outras maneiras; por exemplo, ele fornece uma API relativamente simples e substitui a codificação ASN.1 por codificação textual.

Embora a especificação LDAP seja baseada no X.500, o LDAP não o exige. Uma implementação pode usar qualquer outro servidor de diretório que obedeça a especificação LDAP mais simples – em oposição à especificação X.500. Por exemplo, o Active Directory Service da Microsoft fornece uma interface LDAP. O LDAP tem sido amplamente adotado, particularmente para serviços de diretório de intranet. Ele fornece acesso seguro aos dados do diretório, por meio de autenticação.

9.6 Resumo

Este capítulo descreveu o projeto e a implementação de serviços de nomes em sistemas distribuídos. Os serviços de nomes armazenam atributos de objetos em um sistema distribuído – em particular, seus endereços – e retornam esses atributos quando é fornecido um nome textual para ser pesquisado.

Os principais requisitos do serviço de nomes são a capacidade de manipular um número arbitrário de nomes, um tempo de vida longo, alta disponibilidade, o isolamento de falhas e a tolerância à desconfiança.

Os principais problemas de projeto são, primeiro, a estrutura do espaço de nomes – as regras sintáticas que governam os nomes. Um problema relacionado é o modelo de resolução: as regras pelas quais um nome com múltiplos componentes é transformado em um conjunto de atributos. O conjunto de nomes vinculados deve ser gerenciado. A maioria dos projetos considera que o espaço de nomes deve ser dividido em domínios – seções distintas do espaço de nomes, cada uma das quais associada a uma única autoridade de controle dos vínculos dos nomes que estão sob sua responsabilidade direta.

A implementação do serviço de nomes pode abranger diferentes organizações e comunidades de usuário. Em outras palavras, o conjunto de vínculos entre nomes e atributos é armazenado em vários servidores de nomes, cada um dos quais armazenando pelo menos parte do conjunto de nomes dentro de um domínio de atribuição de nomes. Portanto, surge a questão da navegação – do procedimento pelo qual um nome é resolvido quando as informações necessárias estão armazenadas em vários sites. Os tipos de navegação suportados são: iterativa, *multicast*, recursiva controlada pelo servidor e não-recursiva controlada pelo servidor.

Outro aspecto importante da implementação de um serviço de nomes é o uso de replicação e cache. Ambos ajudam a tornar o serviço altamente disponível e também reduzem o tempo que leva para resolver um nome.

Este capítulo considerou dois casos principais de projetos e implementações de serviço de nomes. O Domain Name System é amplamente usado para atribuição de nomes a computadores e para endereçamento de correio eletrônico na Internet; ele obtém bons tempos de resposta por meio de replicação e uso de cache. O Global Name Service é um projeto que ataca o problema da reconfiguração do espaço de nomes quando ocorrem alterações organizacionais.

Este capítulo também considerou os serviços de diretório, que fornecem dados sobre objetos e serviços correspondentes, quando os clientes fornecem descrições baseadas em atributos. O X.500 é um modelo para serviços de diretório que podem variar na abrangência desde organizações individuais até diretórios globais. Ele tem sido tipicamente usado em intranets desde a chegada do software LDAP.

Exercícios

9.1 Descreva os nomes (incluindo os identificadores) e atributos usados em um serviço de arquivos distribuído, como o NFS (veja o Capítulo 8). *página 324*

9.2 Discuta os problemas levantados pelo uso de *alias* em um serviço de nomes e indique como eles podem ser superados, se houver solução. *página 328–329*

9.3 Explique por que a navegação iterativa é necessária em um serviço de nomes no qual diferentes espaços de nomes são parcialmente integrados, como o esquema de atribuição de nomes de arquivos fornecido pelo NFS. *página 330–331*

9.4 Descreva o problema dos nomes desvinculados na navegação por *multicast*. O que envolve a instalação de um servidor para responder às pesquisas de nomes desvinculados? *página 331–332*

9.5 Como o uso da cache ajuda a disponibilidade de um serviço de nomes? *página 332–333*

9.6 Discuta a ausência de uma distinção sintática (como o uso de um '.' final) entre os nomes absolutos e relativos no DNS. *página 333–334*

9.7 Investigue sua configuração local de domínios e servidores DNS. Você pode encontrar um programa, como o *nslookup*, instalado em sistemas UNIX, o qual permite realizar consultas individuais a servidores de nomes. *página 335*

9.8 Por que os servidores de DNS raízes contêm entradas para nomes de dois níveis, como *yahoo.com* e *purdue.edu*, em vez de nomes de um nível, como *edu* e *com*? *página 335*

9.9 Quais outros endereços de servidores de nomes os servidores DNS contêm por padrão e por quê? *página 335*

9.10 Por que um cliente de DNS poderia escolher a navegação recursiva em vez da navegação iterativa? Qual é a relevância da opção da navegação recursiva para a concorrência dentro de um servidor de nomes? *página 336–337*

9.11 Quando um servidor DNS poderia dar várias respostas para uma única pesquisa de nome e por quê? *página 336–337*

9.12 O GNS não garante que todas as cópias das entradas no banco de dados de atribuição de nomes sejam atualizadas. Como os clientes do GNS sabem que provavelmente receberam uma entrada desatualizada? Sob quais circunstâncias isso poderia ser prejudicial? *página 340–341*

9.13 Discuta as vantagens e inconvenientes em potencial no uso de um serviço de diretório X.500 no lugar do DNS e dos programas de distribuição de e-mail na Internet. Esboce o projeto de um sistema de distribuição de e-mail para um conjunto de redes interligadas na qual todos os usuários e servidores de correio eletrônico são registrados em um banco de dados X.500. *página 342–343*

9.14 Quais problemas de segurança provavelmente são relevantes para um serviço de diretório, como o X500, operando dentro de uma organização como uma universidade? *página 342–343*

Sistemas *Peer-to-peer* 10

Os sistemas *peer-to-peer* representam um paradigma para a construção de sistemas e de aplicativos distribuídos onde dados e recursos computacionais são provenientes da colaboração de muitos *hosts* na Internet de maneira uniforme. Seu aparecimento é uma conseqüência do crescimento muito rápido da Internet, abrangendo milhões de computadores e números semelhantes de usuários exigindo acesso a recursos compartilhados.

Um problema importante dos sistemas *peer-to-peer* é a distribuição de objetos de dados em muitos *hosts* e o subseqüente acesso a eles de uma maneira que equilibre a carga de trabalho e garanta a disponibilidade sem adicionar sobrecargas indevidas. Descreveremos vários sistemas e aplicativos desenvolvidos recentemente projetados para tal.

Vários *middlewares* para sistema *peer-to-peer* tem surgido oferecendo capacidade para compartilhar recursos computacionais, armazenamento e dados presentes em computadores "nos limites da Internet", em uma escala global. Eles exploram de novas maneiras as técnicas de atribuição de nomes, roteamento, replicação de dados e segurança existentes para construir uma camada de compartilhamento de recursos confiável em um conjunto de computadores e redes inseguros e não confiáveis.

Os aplicativos *peer-to-peer* têm sido usados para fornecer compartilhamento de arquivos, uso de cache web, distribuição de informações e outros serviços que exploram os recursos de dezenas de milhares de máquinas pela Internet. Eles demonstram sua maior eficácia quando usados para armazenar conjuntos muito grandes de dados imutáveis. Seu projeto diminui sua eficácia para aplicações que armazenam e atualizam objetos de dados mutáveis.

10.1 Introdução

É esperado que a demanda por serviços na Internet cresça em uma escala limitada apenas pelo tamanho da população mundial. O objetivo dos sistemas *peer-to-peer* é permitir o compartilhamento de dados e recursos em uma escala muito grande, eliminando qualquer exigência de servidores gerenciados separadamente e sua infra-estrutura associada.

A abrangência da expansão de serviços populares pelo acréscimo do número de computadores que os contêm é limitada quando todos os *hosts* devem pertencer e ser gerenciados pelo provedor de serviço. Os custos de administração e recuperação de falhas tendem a influenciar. A largura de banda de rede que pode ser fornecida para um único site servidor sobre os enlaces físicos disponíveis também é uma restrição importante. Os serviços em nível de sistema, como o Sun NFS (Seção 8.3), o Andrew File System (Seção 8.4) ou os servidores de vídeo (Seção 17.6), e os serviços em nível de aplicação, como Google, Amazon ou eBay, apresentam esse problema em diversos graus.

Os sistemas *peer-to-peer* têm como objetivo suportar serviços e aplicativos distribuídos úteis, usando dados e recursos computacionais disponíveis nos computadores pessoais e estações de trabalho que estão presentes em números cada vez maiores na Internet e em outras redes. Isso é cada vez mais atraente, à medida que diminui a diferença de desempenho entre as máquinas *desktop* e servidores e que as conexões a redes de banda larga proliferam.

Mas existe outro objetivo mais amplo: um autor [Shirky 2000] definiu os aplicativos *peer-to-peer* como "aplicativos que exploram os recursos disponíveis nos limites da Internet – armazenamento, ciclos de processamento, conteúdo, presença humana". Cada tipo de compartilhamento de recurso mencionado nessa definição já está representado pelos aplicativos distribuídos disponíveis para a maioria dos tipos de computador pessoal. O objetivo deste capítulo é descrever algumas técnicas gerais que simplificam a construção de aplicativos *peer-to-peer* e que melhoram sua escalabilidade, sua confiabilidade e sua segurança.

Os sistemas cliente-servidor tradicionais gerenciam e fornecem acesso a recursos como arquivos, páginas web ou outros objetos de informação localizados em um único computador servidor ou em um pequeno agrupamento de servidores fortemente acoplados. Em tais projetos centralizados, são exigidas poucas decisões sobre a distribuição dos recursos ou o gerenciamento dos recursos de hardware do servidor, mas a escala do serviço é limitada pela capacidade do hardware do servidor e pela conectividade da rede. Os sistemas *peer-to-peer* fornecem acesso a recursos de informação localizados em computadores de toda uma rede (seja ela a Internet ou uma rede corporativa). Os algoritmos para a distribuição e subseqüente recuperação de objetos de informação são um aspecto importante do projeto do sistema. Seu projeto tem como objetivo distribuir um serviço totalmente descentralizado e organizado, equilibrando, automaticamente, as cargas de armazenamento e processamento de forma dinâmica entre todos os computadores participantes à medida que as máquinas entram e saem do serviço.

Os sistemas *peer-to-peer* compartilham as seguintes características:

- Seu projeto garante que cada usuário contribua com recursos para o sistema.
- Embora eles possam diferir nos recursos com que contribuem, todos os nós em um sistema *peer-to-peer* têm as mesmas capacidades e responsabilidades funcionais.
- Seu correto funcionamento não depende da existência de quaisquer sistemas administrados de forma centralizada.
- Eles podem ser projetados de modo a oferecer um grau limitado de anonimato para os provedores e usuários dos recursos.
- Um problema importante para seu funcionamento eficiente é a escolha de um algoritmo para o arranjo dos dados em muitos *hosts* e o subseqüente acesso a eles, de uma maneira que equilibre a carga de trabalho e garanta a disponibilidade sem adicionar sobrecargas indevidas.

Os computadores e conexões de rede pertencentes e gerenciadas por muitos usuários e organizações diferentes são necessariamente recursos voláteis; seus proprietários não garantem que irão mantê-los ligados, conectados e isentos de falhas. Portanto, a disponibilidade dos processos e computadores participantes dos sistemas *peer-to-peer* é imprevisível. Assim, os serviços *peer-to-peer* não

podem contar com acesso garantido a recursos individuais, embora possam ser projetados de forma a tornar a probabilidade de falha no acesso a uma cópia de um objeto replicado arbitrariamente pequena. É interessante notar que essa deficiência dos sistemas *peer-to-peer* pode se tornar uma vantagem, caso a replicação de recursos que eles solicitam seja explorada de forma a se obter um grau de resistência à falsificação feita por nós mal-intencionados (isto é, por meio de técnicas de tolerância a falhas bizantina, veja o Capítulo 15).

Vários serviços primitivos baseados na Internet, incluindo o DNS (Seção 9.2.3) e Netnews/Usenet [Kantor e Lapsley 1986] adotaram uma arquitetura de vários servidores, escalável e tolerante à falha. O serviço de registro de nomes e distribuição de e-mail Grapevine da Xerox [Birrell *et al.* 1982, Schroeder *et al.* 1984] fornece um exemplo primitivo interessante de serviço distribuído com escalabilidade e tolerância a falhas. O algoritmo de *parlamento de meio expediente* para consenso distribuído de Lamport [Lamport 1989], o sistema de armazenamento replicado Bayou (veja a Seção 14.4.2) e o algoritmo de roteamento de IP entre domínios sem classes (veja a Seção 3.4.3), são todos exemplos de algoritmos distribuídos para a distribuição, ou localização, de informações e podem ser considerados como antecedentes dos sistemas *peer-to-peer*.

Mas o potencial para a implantação de serviços *peer-to-peer,* usando recursos nos limites da Internet, surgiu apenas quando um número significativo de usuários adquiriu conexões de banda larga sempre ativas para a rede, tornando seus computadores *desktop* plataformas convenientes para o compartilhamento de recursos. Isso ocorreu primeiro nos Estados Unidos, por volta de 1999. Em meados de 2004, o número mundial de conexões de banda larga da Internet tinha ultrapassado tranqüilamente os 100 milhões [Internet World Stats 2004].

Podem ser identificadas três gerações de desenvolvimento de sistemas e aplicativos *peer-to-peer*. A primeira geração foi lançada pelo serviço de troca de músicas Napster [OpenNap 2001], que descreveremos na próxima seção. Uma segunda geração de aplicativos de compartilhamento de arquivos, oferecendo maior escalabilidade, anonimato e tolerância a falhas surgiu logo depois, incluindo Freenet [Clarke *et al.* 2000, Freenet 2004], Gnutella, Kazaa [Leibowitz *et al.* 2003] e BitTorrent [Cohen 2003].

Middleware peer-to-peer ◊ A terceira geração é caracterizada pelo aparecimento de camadas de *middleware* para o gerenciamento de recursos distribuídos em uma escala global independente de aplicativos. Agora, várias equipes de pesquisa concluíram o desenvolvimento, a avaliação e o refinamento de plataformas de *middleware peer-to-peer* e as demonstraram, ou implantaram, em diversos serviços de aplicativo. Os exemplos mais conhecidos e totalmente desenvolvidos incluem o Pastry [Rowstron e Druschel 2001], o Tapestry [Zhao *et al.* 2004], o CAN [Ratnasamy *et al.* 2001], o Chord [Stoica *et al.* 2001] e o Kademlia [Maymounkov e Mazieres 2002].

Essas plataformas são projetadas para colocar recursos (objetos de dados, arquivos) em um conjunto de computadores amplamente distribuídos em toda a Internet e para direcionar mensagens a eles em nome de clientes, retirando destes as decisões sobre o posicionamento de recursos e a necessidade de conter informações sobre o paradeiro dos recursos que exigem. Ao contrário dos sistemas de segunda geração, eles dão garantias do envio de pedidos em um número limitado de passos intermediário de rede. Eles colocam réplicas dos recursos de maneira estruturada em computadores *host* disponíveis, levando em conta sua disponibilidade volátil, sua confiabilidade e seus requisitos de equilíbrio de carga variáveis, localização do armazenamento e uso das informações.

Os recursos são identificados por identificadores globalmente exclusivos (GUIDs) e esses identificadores normalmente são originados como um código de resumo (*hashing*) seguro (descrito na Seção 7.4.3), a partir de algum, ou de todos, os estados do recurso. O uso de um código de resumo seguro torna um recurso "automaticamente certificado" – os clientes que recebem um recurso podem verificar a validade do código de resumo. Isso o protege contra falsificação de nós não confiáveis nos quais pode ser armazenado. Mas essa técnica exige que os estados dos recursos sejam imutáveis, pois uma mudança no estado resultaria em um valor de resumo diferente. Por isso, os sistemas de armazenamento *peer-to-peer* são inerentemente mais convenientes para o armazenamento de objetos imutáveis (como arquivos de música ou vídeo). Seu uso em objetos com valores mutáveis é mais desafiador, mas pode ser resolvido com a adição de servidores confiáveis para gerenciar uma seqüência de versões e identificar a versão corrente (como é feito, por exemplo, no OceanStore e no Ivy, descritos nas Seções 10.6.2 e 10.6.3).

O uso de sistemas *peer-to-peer* para aplicações que exigem um alto nível de disponibilidade para os objetos armazenados requer um projeto cuidadoso de aplicativos para evitar situações em que todas as réplicas de um objeto estão indisponíveis simultaneamente. Existe um risco que isso ocorra para objetos que são armazenados em computadores com uma mesma propriedade, localização geográfica, administração, conectividade de rede, país ou jurisdição. O uso de GUIDs distribuídas aleatoriamente ajuda, distribuindo as réplicas dos objetos para nós localizados aleatoriamente na rede subjacente. Se a rede subjacente abrange muitas organizações no mundo, então o risco de indisponibilidade simultânea é muito reduzido.

Sobreposição de roteamento *versus* roteamento de IP ◊ À primeira vista, a sobreposição de roteamento (*overlay routing*) compartilha muitas características com a infra-estrutura de roteamento de pacotes IP, que constitui o principal mecanismo de comunicação da Internet (veja a Seção 3.4.3). Portanto, é legítimo perguntar por que um mecanismo adicional de roteamento em nível de aplicativo é exigido nos sistemas *peer-to-peer*. A resposta está nas várias distinções identificadas na tabela da Figura 10.1. Pode-se argumentar que algumas dessas distinções surgem da natureza "legada" do IP como o principal protocolo da Internet, mas o impacto do legado provavelmente é forte demais para ser superado e para reprojetar o IP para suportar aplicativos *peer-to-peer* mais diretamente.

Computação distribuída ◊ A exploração do poder de computação excedente nos computadores do usuário final tem sido assunto de interesse e experiências há um longo tempo. Um trabalho com os primeiros computadores pessoais no PARC da Xerox [Shoch e Hupp 1982] mostrou a exeqüibilidade da execução de tarefas que exigem alto poder computacional fracamente acopladas, executando proces-

	IP	*Roteamento em nível de aplicativo*
Escala	O IPv4 é limitado a 2^{32} nós endereçáveis. O espaço de nomes IPv6 é muito mais generoso (2^{128}), mas nas duas versões os endereços são estruturados hierarquicamente e grande parte do espaço é previamente alocado de acordo com os requisitos administrativos.	Os sistemas *peer-to-peer* podem endereçar mais objetos. O espaço de nomes do GUID é muito grande e plano ($>2^{128}$), podendo ser ocupado de forma muito mais completa.
Equilíbrio da carga	As cargas sobre os roteadores são determinadas pela topologia da rede e pelos padrões de tráfego vigentes.	Os objetos podem se posicionar aleatoriamente e com isso os padrões de tráfego não têm nada a ver com a topologia da rede.
Dinâmica da rede (adição/exclusão de objetos/nós)	As tabelas de roteamento de IP são atualizadas de forma assíncrona, com base nos melhores esforços, com constantes de tempo na ordem de uma hora.	As tabelas de roteamento podem ser atualizadas de forma síncrona ou assíncrona, com atrasos de frações de segundo.
Tolerância a falhas	A redundância é projetada na rede IP por seus gerentes, garantindo tolerância da falha de conectividade de um único roteador ou da rede. A replicação com fator de multiplicação n é dispendiosa.	Rotas e referências de objeto podem ser replicadas por um fator n, garantindo a tolerância de n falhas de nós ou conexões.
Identificação do destino	Cada endereço IP é mapeado em exatamente um nó de destino.	As mensagens podem ser direcionadas para a réplica mais próxima de um objeto de destino.
Segurança e anonimato	O endereçamento só é seguro quando todos os nós são confiáveis. O anonimato dos proprietários de endereços não pode ser obtido.	A segurança pode ser obtida, mesmo em ambientes de confiança limitada. Um grau limitado de anonimato pode ser fornecido.

Figura 10.1 Distinções entre IP e sobreposição de roteamento em nível de aplicativo para sistemas *peer-to-peer*.

sos *background* em ~100 computadores pessoais interligados em uma rede local. Mais recentemente, números muito maiores de computadores foram usados para efetuar vários cálculos científicos que exigiam quantidades quase ilimitadas de poder de computação.

O trabalho desse tipo mais conhecido é o projeto *SETI@home* [Anderson *et al.* 2002], que faz parte de um projeto mais amplo da *Search for Extra-Terrestrial Intelligence*. O SETI@home particiona um fluxo de dados de radiotelescópio digitalizados, em unidades de trabalho de 107 segundos, cada um com cerca de 350KB, e os distribui para computadores clientes cujo poder de computação recebe a contribuição de voluntários. Cada unidade de trabalho é distribuída de forma redundante para 3 a 4 computadores pessoais, para proteção contra nós errôneos ou mal-intencionados, e verificadas por padrões significativos de sinal. A distribuição das unidades de trabalho e a coordenação dos resultados são manipuladas por um único servidor, que é responsável pela comunicação com todos os clientes. Anderson *et al.* [2002] relataram que 3,91 milhões de computadores pessoais tinham participado do projeto SETI@home em agosto de 2002, resultando no processamento de 221 milhões de unidades de trabalho, representando uma média de 27,36 teraflops de poder computacional durante 12 meses, até julho de 2002. O trabalho concluído até aquela data representou a maior computação jamais registrada.

A computação da SETI é incomum, pois não envolve nenhuma comunicação, ou coordenação entre computadores, enquanto eles estão processando as unidades de trabalho, e os resultados são informados para um servidor central em uma única mensagem curta, que pode ser enviada quando o cliente e o servidor estão disponíveis. Algumas outras tarefas científicas dessa natureza foram identificadas, incluindo a procura de números primos grandes e tentativas de decifração pela força bruta. Mas o desencadeamento do poder computacional na Internet para uma variedade mais ampla de tarefas dependerá do desenvolvimento de uma plataforma distribuída que suporte compartilhamento de dados e a coordenação da computação entre os computadores participantes em larga escala. Esse é o objetivo do projeto Grid, discutido no Capítulo 19.

Neste capítulo, focalizaremos os algoritmos e sistemas desenvolvidos até o momento para o compartilhamento de dados em redes *peer-to-peer*. Na Seção 10.2, resumiremos o projeto do Napster e examinaremos as lições aprendidas com ele. Na Seção 10.3, descreveremos os requisitos gerais das camadas de *middleware peer-to-peer*. As seções seguintes abordarão o projeto e a aplicação de plataformas de *middleware peer-to-peer*, começando com uma especificação abstrata, na Seção 10.4, seguida de descrições detalhadas de dois exemplos totalmente desenvolvidos (Seção 10.5) e algumas aplicações deles (Seção 10.6).

10.2 Napster e seu legado

A primeira aplicação a surgir, na qual a exigência de um serviço de armazenamento e recuperação de informações com capacidade de escala global, foi o *download* de arquivos de música digital. Tanto a necessidade quanto a exeqüibilidade de uma solução *peer-to-peer* foram demonstradas pela primeira vez pelo sistema Napster [OpenNap 2001], que fornecia um meio para os usuários compartilharem arquivos. O Napster se tornou muito popular para troca de música, logo após seu lançamento, em 1999. Em seu auge, vários milhões de usuários estavam registrados e milhares trocavam arquivos de música simultaneamente.

A arquitetura do Napster incluía índices centralizados, mas eram os usuários que forneciam os arquivos, os quais eram armazenados e acessados em seus computadores pessoais. O método de operação do Napster está ilustrado pela seqüência de etapas mostrada na Figura 10.2. Note que, na etapa 5, espera-se que os clientes adicionem seus próprios arquivos de música no pool de recursos compartilhados, transmitindo para o serviço de indexação do Napster um *link* para cada arquivo disponível. Assim, a motivação do Napster, e o segredo de seu sucesso, foi tornar um grande conjunto de arquivos distribuídos disponível para os usuários em toda a Internet, cumprindo a máxima de Shirky pelo fornecimento de acesso a "recursos compartilhados nos limites da Internet".

O Napster foi desativado como resultado de ações jurídicas contra os operadores do serviço Napster, impetradas pelos proprietários dos direitos autorais de parte do material (isto é, música codificada de forma digital) que estava disponível nele. (Veja o quadro: *Os sistemas peer-to-peer e problemas de propriedade de direitos autorais*.)

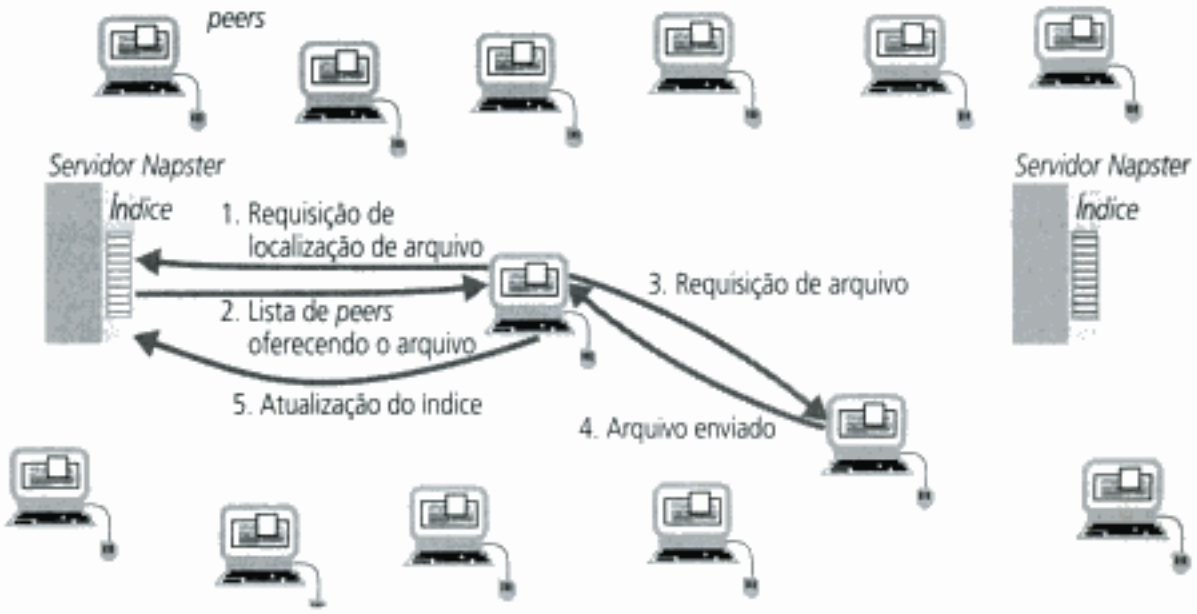


Figura 10.2 Napster: compartilhamento de arquivos *peer-to-peer* com um índice replicado centralizado.

O anonimato dos receptores e dos provedores de dados compartilhados e de outros recursos é uma preocupação dos projetistas de sistemas *peer-to-peer*. Em sistemas com muitos nós, o roteamento das requisições e dos resultados pode se tornar suficientemente tortuoso para ocultar sua fonte e o conteúdo dos arquivos pode ser distribuído em vários nós, dispersando a responsabilidade por torná-los disponíveis. Estão disponíveis mecanismos de comunicação anônima que são resistentes à maioria das formas de análise de tráfego [Goldschlag *et al.* 1999]. Se os arquivos também forem cifrados antes de serem colocados nos servidores, os proprietários dos servidores com certeza poderão negar qualquer conhecimento do conteúdo. Mas essas técnicas de anonimato aumentam o custo do compartilhamento de recursos e um trabalho recente mostrou que o anonimato disponível é insuficiente contra alguns ataques [Wright *et al.* 2002].

Os projetos do Freenet [Clarke *et al.* 2000] e do FreeHaven [Dingledine *et al.* 2000] enfocam o fornecimento de serviços de arquivo na Internet oferecendo anonimato para os provedores e usuários dos arquivos. Ross Anderson propôs o Eternity Service [Anderson 1996], um serviço de armazenamento que proporciona garantias de longo prazo da disponibilidade dos dados, por meio da resistência a todos os tipos de perda acidental de dados e ataques de negação de serviço. Ele baseia a necessidade de tal serviço na observação de que, enquanto a publicação é um estado permanente para informações impressas – é praticamente impossível excluí-la, uma vez que tenha sido publicada e distribuída para alguns milhares de bibliotecas em diversas organizações e jurisdições pelo mundo – as publicações eletrônicas não podem obter facilmente o mesmo nível de resistência à censura ou supressão. Anderson aborda os requisitos técnicos e econômicos para garantir a integridade do armazenamento, e também aponta que o anonimato é freqüentemente um requisito fundamental para a persistência da informação, pois ele proporciona a melhor defesa contra contestações legais ou ações ilegais, como subornos ou ataques contra os criadores, proprietários ou mantenedores dos dados.

Lições aprendidas com o Napster ◊ O Napster demonstrou a exeqüibilidade da construção da utilidade de um serviço de larga escala, que depende quase totalmente de dados e computadores pertencentes a usuários normais da Internet. Para evitar o esgotamento dos recursos computacionais de usuários individuais (por exemplo, o primeiro usuário a oferecer uma música muito procurada) e suas conexões de rede, o Napster levava em conta o caráter local da rede – o número de passos intermediários entre o cliente e o servidor – ao alocar um servidor para um cliente que estivesse solicitando a música. Esse mecanismo simples de distribuição de carga permitia que o serviço mudasse de escala para satisfazer as necessidades de grandes números de usuários.

Os sistemas *peer-to-peer* e problemas de propriedade de direitos autorais

Os desenvolvedores do Napster alegaram que não eram responsáveis pela violação dos direitos autorais dos proprietários porque não participavam do processo de cópia; o qual era realizado inteiramente entre as máquinas dos usuários. A alegação fracassou, porque os servidores de índice foram considerados uma parte essencial do processo. Como os servidores de índice estavam localizados em endereços conhecidos, seus operadores eram incapazes de se manter anônimos e, portanto, poderiam ser alvo de ações judiciais.

Um serviço de compartilhamento de arquivos mais completamente distribuído poderia ter obtido uma separação melhor das responsabilidades jurídicas, dispersando a responsabilidade por todos os usuários do Napster e, assim, tornando a busca de soluções jurídicas muito difícil, se não impossível. Qualquer que seja a visão que se tenha sobre a legitimidade da cópia de arquivos para o propósito de compartilhar material protegido por direitos autorais, existem justificativas sociais e políticas legítimas para o anonimato de clientes e servidores em alguns contextos de aplicação. A justificativa mais persuasiva surge quando o anonimato é usado para vencer a censura e manter a liberdade de expressão dos indivíduos em sociedades ou organizações opressivas.

Sabe-se que o e-mail e os sites web têm desempenhado um papel significativo na obtenção do conhecimento público em tempos de crises políticas em tais sociedades; seu papel poderia ser mais atuante, se os autores pudessem ser protegidos pelo anonimato. O "delato" é um caso relacionado: um "delator" é um funcionário que publica, ou relata, as transgressões de seu empregador para as autoridades, sem revelar sua própria identidade por medo de sanções ou demissão. Em algumas circunstâncias, é razoável que tal ação seja protegida pelo anonimato.

Limitações: o Napster usava um índice unificado (replicado) de todos os arquivos de música disponíveis. Para a aplicação em questão, o requisito da consistência entre as réplicas não era fundamental; portanto, isso não atrapalhava o desempenho, mas para muitas aplicações constituiria uma limitação. A não ser que o caminho de acesso para os objetos de dados seja distribuído, a descoberta e o endereçamento de objetos provavelmente se tornam um gargalo.

O Napster tirava proveito, de outras maneiras, das características especiais da aplicação para a qual foi projetado:

- Os arquivos de música nunca são atualizados, evitando qualquer necessidade de tornar todas as réplicas dos arquivos consistentes após as atualizações.
- Nenhuma garantia é exigida com relação à disponibilidade de arquivos individuais – se um arquivo de música estiver temporariamente indisponível, ele poderá ser carregado por *download* posteriormente. Isso reduz o requisito da confiança dos computadores individuais e de suas conexões com a Internet.

10.3 *Middleware* para *peer-to-peer*

Um problema importante no projeto de aplicativos *peer-to-peer* é o fornecimento de um mecanismo para permitir aos clientes acessarem recursos de dados rapidamente e de maneira segura, quando eles estão localizados por toda a rede. O Napster mantinha para esse propósito um índice unificado dos arquivos disponíveis, fornecendo os endereços de rede de seus *hosts*. Os sistemas *peer-to-peer* de armazenamento de arquivos de segunda geração, como o Gnutella e o Freenet, empregam índices particionados e distribuídos, mas os algoritmos usados são específicos para cada sistema.

Esse problema de localização existia em vários serviços anteriores ao paradigma *peer-to-peer*. Por exemplo, o Sun NFS trata dessa necessidade com a ajuda de uma camada de abstração de sistema de arquivos virtual, em cada cliente, o qual aceita pedidos para acessar arquivos armazenados em vários servidores, em termos de referências de arquivo virtuais (isto é, *v-nodes*, veja a Seção 8.3). Essa solução conta com um volume de configuração prévia em cada cliente substancial e com a necessidade

de intervenção manual quando os padrões de distribuição de arquivo ou o servidor muda. Claramente, ela não tem capacidade de escala além de um serviço gerenciado por uma única organização. O AFS (Seção 8.4) tem propriedades semelhantes.

Os sistemas de *middleware peer-to-peer* são projetados especificamente para atender a necessidade da disposição automática, e da subseqüente localização, dos objetos distribuídos gerenciados por sistemas e aplicativos *peer-to-peer*.

Requisitos funcionais ◊ A função de um *middleware peer-to-peer* é simplificar a construção de serviços implementados em muitos *hosts*, em uma rede amplamente distribuída. Para obter isso, ele deve permitir aos clientes localizarem e se comunicarem com qualquer recurso individual disponibilizado para um serviço, mesmo que os recursos estejam amplamente distribuídos entre os *hosts*. Outros requisitos importantes incluem a capacidade de adicionar novos recursos e removê-los à vontade, e de adicionar *hosts* no serviço e removê-los. Sendo um tipo de *middleware*, o *middleware peer-to-peer* deve oferecer uma interface de programação simples para programadores de aplicações, que seja independente dos tipos de recurso distribuído manipulados pela aplicação.

Requisitos não-funcionais ◊ Para funcionar eficientemente, o *middleware peer-to-peer* também deve tratar dos seguintes requisitos não-funcionais [cf. Kubiatowicz 2003]:

Escalabilidade global: um dos objetivos dos aplicativos *peer-to-peer* é explorar os recursos de hardware de grandes quantidades de *hosts* conectados na Internet. Portanto, o *middleware peer-to-peer* deve ser projetado de modo a suportar aplicativos que acessam milhões de objetos em dezenas ou centenas de milhares de *hosts*.

Balanceamento da carga: o desempenho de qualquer sistema projetado para explorar um grande número de computadores depende da distribuição balanceada da carga de trabalho entre eles. Para os sistemas que estamos considerando, isso será obtido por um posicionamento aleatório dos recursos, junto com o uso de réplicas dos recursos muito utilizados.

Otimização das interações locais entre *peers* vizinhos: a "distância da rede" entre nós que interagem tem um impacto significativo sobre a latência das interações individuais, como as requisições de cliente para acesso aos recursos. As cargas do tráfego da rede também sofrem esse impacto. O *middleware* deve ter como objetivo colocar os recursos próximos dos nós que mais os acessam.

Acomodar a disponibilidade altamente dinâmica de *hosts*: a maioria dos sistemas *peer-to-peer* é construída a partir de computadores *host* que estão livres para entrar ou sair do sistema a qualquer momento. Os *hosts* e os segmentos de rede usados nos sistemas *peer-to-peer* não pertencem, nem são gerenciados, por nenhuma autoridade única; e nem sua confiabilidade, nem sua participação contínua no abastecimento de um serviço são garantidas. Um desafio importante para os sistemas *peer-to-peer* é fornecer um serviço confiável, a despeito desses fatos. Quando os *hosts* entram no sistema, eles devem ser integrados nele e a carga deve ser redistribuída para explorar seus novos recursos. Quando eles saem do sistema voluntária, ou involuntariamente, o sistema deve detectar sua saída e redistribuir sua carga e os recursos.

Estudos de aplicativos e sistemas *peer-to-peer*, como o Gnutella e o Overnet, têm mostrado uma reviravolta considerável dos *hosts* participantes [Saroiu *et al.* 2002, Baghwan *et al.* 2003]. Para o sistema de compartilhamento de arquivos *peer-to-peer* Overnet, com 85.000 *hosts* ativos em toda a Internet, Baghwan *et al.* mediram uma duração média de sessão de 135 minutos (e uma mediana de 79 minutos) para uma amostra aleatória de 1.468 *hosts* em um período de 7 dias, com 260 a 650 dos 1.468 *hosts* disponíveis para o serviço a qualquer momento. (Uma sessão representa um período durante o qual um *host* está disponível, antes de ser voluntária, ou inevitavelmente, desconectado.)

Por outro lado, pesquisadores da Microsoft mediram uma duração de sessão de 37,7 horas para uma amostra aleatória de 20.000 máquinas conectadas na rede corporativa daquela empresa, com 14.700 a 15.600 máquinas disponíveis para o serviço a qualquer dado momento [Castro *et al.* 2003]. Essas medidas são baseadas em um estudo de exeqüibilidade para o sistema de arquivo *peer-to-peer* Farsite [Bolosky *et al.* 2000]. A enorme disparidade entre os valores obtidos nesses estudos pode ser atribuída principalmente às diferenças no comportamento e no ambiente de rede entre os usuários de Internet individuais e os usuários de uma rede corporativa, como a da Microsoft.

Segurança dos dados em um ambiente com confiança heterogênea: em sistemas de escala global, com *hosts* participantes de diversos proprietários, a confiança deve ser estabelecida pelo uso de mecanismos de autenticação e criptografia, para garantir a integridade e a privacidade da informação.

Anonimato, capacidade de negação e resistência à censura: observamos (no quadro anterior) que o anonimato dos proprietários e destinatários dos dados é uma preocupação legítima em muitas situações que exigem resistência à censura. Um requisito relacionado é o de que os *hosts* que contêm dados devem ser capazes de negar a responsabilidade por contê-los ou fornecê-los, de forma plausível. O uso de grandes números de *hosts* nos sistemas *peer-to-peer* pode ser útil na obtenção dessas propriedades.

Portanto, o projeto de uma camada de *middleware* para suportar sistemas *peer-to-peer* em escala global é um problema difícil. Os requisitos de escalabilidade e disponibilidade tornam impraticável manter um banco de dados em todos os nós clientes fornecendo as localizações de todos os recursos (objetos) de interesse.

O conhecimento das localizações dos objetos deve ser particionado e distribuído por toda a rede. Cada nó se torna responsável por manter o conhecimento detalhado das localizações dos nós e objetos em uma parte do espaço de nomes, assim como um conhecimento geral da topologia do espaço de nomes inteiro (Figura 10.3). Um alto grau de replicação desse conhecimento é necessário para garantir a segurança em face da disponibilidade volátil dos *hosts* e da conectividade intermitente da rede. Nos sistemas que vamos descrever a seguir, fatores de replicação de até 16 são normalmente usados.

10.4 Sobreposição de roteamento

O desenvolvimento de um *middleware* que atenda os requisitos anteriores é um tópico de pesquisa ativa e vários sistemas importantes de *middlewares* já surgiram. Neste capítulo, descreveremos dois deles em detalhes.

Um algoritmo distribuído conhecido como *sobreposição de roteamento* (*routing overlay*) assume a responsabilidade por localizar nós e objetos. O nome denota o fato de que o *middleware* assume a

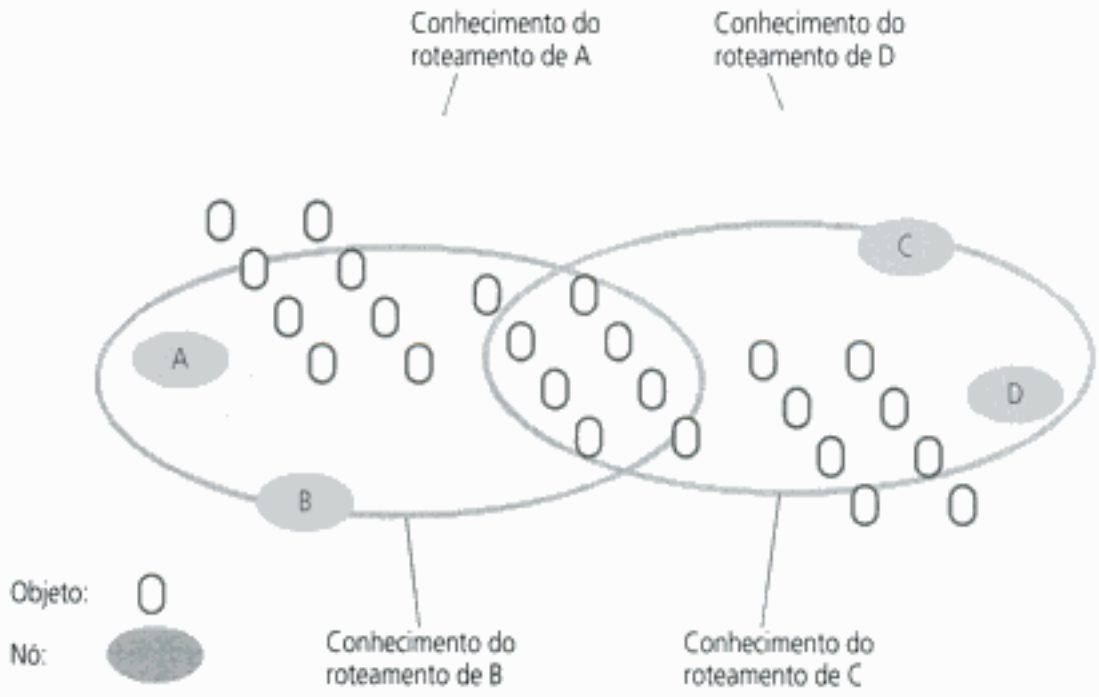


Figura 10.3 Distribuição das informações em uma sobreposição de roteamento.

forma de uma camada que é responsável por direcionar requisições de qualquer cliente para um *host* que contenha o objeto para o qual a requisição é endereçada. Os objetos de interesse podem ser alocados e, subseqüentemente, movidos para qualquer nó da rede, sem envolvimento do cliente. Ele é chamado de sobreposição porque implementa um mecanismo de roteamento na camada de aplicação que é completamente separado de todos os outros mecanismos de roteamento implantados em nível da rede, como o roteamento IP. Essa estratégia de gerenciamento e localização de objetos replicados foi analisada pela primeira vez e mostrou-se eficaz para redes envolvendo muitos nós, em um artigo inédito de Plaxton *et al.* [1997].

A sobreposição de roteamento garante que qualquer nó possa acessar qualquer objeto por meio do roteamento de cada requisição por uma seqüência de nós, explorando o conhecimento existente em cada um deles para localizar o objeto de destino. Os sistemas *peer-to-peer* normalmente armazenam várias réplicas dos objetos para garantir disponibilidade. Nesse caso, a sobreposição de roteamento mantém o conhecimento da localização de todas as réplicas disponíveis e distribui as requisições para o nó ativo mais próximo (isto é, um que não tenha falhado) que tenha uma cópia do objeto relevante.

Os GUIDs usados para identificar nós e objetos são um exemplo de nomes "puros", referidos na Seção 9.1.1, também conhecidos como *identificadores opacos*, pois não revelam nada sobre a localização dos objetos a que se referem.

A principal tarefa de uma sobreposição de roteamento é a seguinte:

1. Um cliente que queira invocar uma operação sobre um objeto envia uma requisição incluindo o GUID do objeto para a sobreposição de roteamento, a qual direciona a requisição para um nó em que resida uma réplica do objeto.

A sobreposição de roteamento também deve executar algumas outras tarefas:

2. Um nó que queira tornar um novo objeto disponível para um serviço *peer-to-peer* calcula um GUID para o objeto e o anuncia para a sobreposição de roteamento, a qual então garante que o objeto possa ser acessado por todos os outros clientes.
3. Quando os clientes solicitam a remoção de objetos do serviço, a sobreposição de roteamento deve torná-los indisponíveis.
4. Os nós (isto é, computadores) podem entrar e sair do serviço. Quando um nó entra no serviço, a sobreposição de roteamento faz preparativos para que ele assuma parte das responsabilidades dos outros nós. Quando um nó sai (voluntariamente, ou como resultado de uma falha de sistema ou da rede), suas responsabilidades são distribuídas entre os outros nós.

O GUID de um objeto é calculado a partir de todo o estado do objeto, ou de parte dele, usando-se uma função que produz um valor que tem probabilidade muito alta de ser exclusivo. A exclusividade é verificada procurando-se outro objeto com o mesmo GUID. Uma função de resumo (*hashing*), como SHA-1 (veja a Seção 7.4), é usada para gerar o GUID a partir do valor do objeto. Como esses identificadores distribuídos aleatoriamente são usados para determinar a colocação de objetos e para recuperá-los, às vezes os sistemas de sobreposição de roteamento são descritos como *tabelas de resumo distribuídas* (*DHT – Distributed Hashing Table*) e isso se reflete pela forma mais simples de API usada para acessá-los, como se vê na Figura 10.4. Com essa API, a operação *put( )* é usada para enviar um item de dados para ser armazenado junto com seu GUID. A camada DHT assume a responsabilidade por escolher um local para ele, armazená-lo (com réplicas para garantir a disponibilidade) e fornecer acesso a ele por meio da operação *get( )*.

Uma forma ligeiramente mais flexível de API é fornecida por uma camada de *localização e roteamento de objeto distribuído* (*DOLR- Distributed Object Location and Routing*), como se vê na Figura 10.5. Com essa interface, os objetos podem ser armazenados em qualquer lugar e a camada DOLR é responsável por manter um mapeamento entre identificadores de objeto (GUIDs) e os endereços dos nós nos quais estão localizadas as réplicas dos objetos. Os objetos podem ser replicados e armazenados com o mesmo GUID em diferentes *hosts* e a sobreposição de roteamento assume a responsabilidade por rotear as requisições para a réplica mais próxima disponível.

Com o modelo DHT, um item de dados com GUID *X* é armazenado no nó cujo GUID é numericamente mais próximo a *X*, e nos *r hosts* com GUIDs numericamente mais próximos a ele, onde *r* é

put(GUID, dados)
Os *dados* são armazenados em réplicas em todos os nós responsáveis pelo objeto identificado pelo *GUID*.

remove(GUID)
Exclui todas as referências para o *GUID* e para os dados associados.

value = get(GUID)
Os dados associados ao *GUID* são recuperados de um dos nós responsáveis por eles.

Figura 10.4 Interface de programação básica para uma tabela de resumo distribuída (DHT), conforme implementada pela API PAST sobre o Pastry.

um fator de replicação escolhido para garantir uma probabilidade de disponibilidade muito alta. Com o modelo DOLR, as localizações das réplicas de objetos de dados são decididas fora da camada de roteamento e o endereço de *host* de cada réplica é notificado para o DOLR com a operação *publish()*.

As interfaces das Figuras 10.4 e 10.5. são baseadas em um conjunto de representações abstratas propostas por Dabek *et al.* [2003] para mostrar que a maioria das implementações de sobreposição de roteamento *peer-to-peer* desenvolvidas até agora fornece funcionalidade muito parecida.

O trabalho no projeto de sistemas de sobreposição de roteamento começou em 2000 e continua sendo uma área de pesquisa bastante ativa. Atualmente, vários deles foram desenvolvidos e avaliados com êxito. As avaliações desses protótipos demonstraram que seu desempenho e confiança são adequados para uso em muitos ambientes de produção. Na próxima seção, descreveremos dois deles em detalhes: o Pastry, que implementa uma API de tabela de resumo distribuída (DHT) semelhante àquela apresentada na Figura 10.4, e o Tapestry, que implementa uma API semelhante àquela mostrada na Figura 10.5. Tanto o Pastry como o Tapestry empregam um mecanismo de roteamento conhecido como *roteamento baseado em prefixo*, para determinar as rotas para a distribuição de mensagens com base nos valores dos GUIDs para os quais elas são endereçadas. O roteamento baseado em prefixo restringe a busca do próximo nó ao longo da rota, aplicando uma máscara binária que seleciona um número cada vez maior de dígitos hexadecimais do GUID de destino, após cada etapa (*hop*). (Essa técnica também é empregada no CIDR, para IP, descrito em linhas gerais na Seção 3.4.3.)

Foram desenvolvidos outros esquemas de roteamento que exploram diferentes medidas de distância para restringir a busca do próximo destino de *hop*. O Chord [Stoica *et al.* 2001] baseia a escolha na diferença numérica entre os GUIDs do nó selecionado e do nó de destino. O CAN [Ratnasamy *et al.* 2001] usa a distância em um hiperespaço *d*-dimensional, no qual os nós são colocados. O Kade-

publish(GUID)
O *GUID* pode ser calculado a partir do objeto (ou de alguma parte dele, por exemplo, seu nome). Esta função faz o nó efetuar uma operação *publish* no *host* para o objeto correspondente ao GUID.

unpublish(GUID)
Torna o objeto correspondente ao *GUID* inacessível.

sendToObj(msg, GUID, [n])
Seguindo o paradigma orientado a objetos, uma mensagem de invocação é enviada a um objeto para acessá-lo. Poderia ser uma requisição para abrir uma conexão TCP para transferência de dados, ou para retornar uma mensagem contendo todo o estado do objeto, ou parte dele. O parâmetro opcional final *[n]*, se estiver presente, solicita a distribuição da mesma mensagem para *n* réplicas do objeto.

Figura 10.5 Interface de programação básica para localização e roteamento de objeto distribuído (DOLR), conforme implementada pelo Tapestry.

mlia [Maymounkov e Mazieres 2002] usa a operação XOR dos pares de GUIDs como métrica para a distância entre os nós. Como a operação XOR é simétrica, o Kademlia pode manter as tabelas de roteamento dos participantes de forma muito simples, pois eles sempre recebem requisições dos mesmos nós contidos em suas tabelas de roteamento.

Os GUIDs não são legíveis para seres humanos; portanto, os aplicativos clientes devem obter os GUIDs dos recursos de interesse por meio de alguma forma de serviço de indexação, usando nomes legíveis para seres humanos ou pedidos de busca. De preferência, esses índices também são armazenados de uma maneira *peer-to-peer* para superar a deficiência dos índices centralizados, evidenciada pelo Napster. Mas, nos casos simples, como música, ou publicações, disponíveis para *download peer-to-peer*, eles podem ser simplesmente indexados nas páginas web (cf. BitTorrent [Cohen 2003]). No BitTorrent, uma pesquisa de índice web leva a um arquivo *stub* contendo detalhes sobre o recurso desejado, incluindo seu GUID e o URL de um *rastreador* (*tracker*) – um *host* que contém uma lista atualizada dos endereços de rede dos provedores que querem fornecer o arquivo.

A descrição precedente em relação a sobreposição de roteamento provavelmente terá levantado dúvidas na cabeça do leitor, sobre seu desempenho e confiabilidade. As respostas para essas perguntas surgirão das descrições dos sistemas práticos de sobreposição de roteamento, na próxima seção.

10.5 Estudos de caso: Pastry, Tapestry

A estratégia do roteamento baseado em prefixo foi adotada pelo Pastry e pelo Tapestry. O Pastry tem um projeto simples, porém eficiente, que o torna um bom primeiro exemplo para estudarmos em detalhes. O Pastry é a infra-estrutura de roteamento de mensagens implantada em várias aplicações, incluindo o PAST [Druschel e Rowstron 2001], um sistema de armazenamento de arquivos em repositório (imutável), implementado como uma tabela de resumo distribuída (DHT) com a API da Figura 10.4, e o Squirrel, um serviço *peer-to-peer* para uso de cache web, descrito na Seção 10.6.1.

O Tapestry é a base do sistema de armazenamento OceanStore, que descreveremos na Seção 10.6.2. Ele tem uma arquitetura mais complexa do que o Pastry, pois seu objetivo é suportar uma variedade mais ampla de estratégias de localidade. Descreveremos isso na Seção 10.5.2, comparando com o Pastry.

10.5.1 Pastry

O Pastry [Rowstron e Druschel 2001, Castro *et al.* 2002, projeto FreePastry 2004] é uma sobreposição de roteamento com as características que descrevemos na Seção 10.4. Todos os nós e objetos que podem ser acessados por meio do Pastry recebem GUIDs de 128 bits. Para nós, eles são calculados por meio da aplicação de uma função de resumo segura (como SHA-1, veja a Seção 7.4.1) na chave pública, com a qual cada nó é fornecido. Para objetos como arquivos, o GUID é calculado por meio da aplicação de uma função de resumo segura no nome do objeto, ou em alguma parte do estado armazenado do objeto. O GUID resultante tem as propriedades normais dos valores de resumo seguros – isto é, eles são distribuídos aleatoriamente no intervalo de 0 a 2^{128}-1. Eles não fornecem nenhum indício sobre o valor a partir do qual foram calculados e conflitos entre GUIDs de diferentes nós, ou objetos, são extremamente improváveis. (Nesse evento improvável, o Pastry o detecta e adota uma ação corretiva.)

Em uma rede com N nós participantes, o algoritmo de roteamento do Pastry direcionará corretamente uma mensagem endereçada para qualquer GUID em $O(\log N)$ etapas. Se o GUID identifica um nó correntemente ativo, a mensagem é enviada para esse nó; caso contrário, ela é enviada para o nó ativo cujo GUID é numericamente mais próximo a ele. Os nós ativos assumem a responsabilidade pelo processamento de requisições endereçadas para todos os objetos em sua vizinhança numérica.

As etapas de roteamento envolvem o uso de um protocolo de transporte subjacente (normalmente UDP), para transferir a mensagem para um nó do Pastry que esteja mais próximo de seu destino. Mas, note que a proximidade referida aqui é um espaço totalmente artificial – o espaço dos GUIDs. O trans-

porte real de uma mensagem pela Internet entre dois nós do Pastry pode exigir um número substancial de etapas de IP. Para minimizar o risco de caminhos de transporte desnecessariamente estendidos, o Pastry usa uma métrica de localidade baseada na distância da rede subjacente (como contagens de etapas – *hop* – ou medidas da latência de viagem de ida e volta) para selecionar os vizinhos apropriados, ao configurar as tabelas de roteamento usadas em cada nó.

Milhares de *hosts* localizados em sites amplamente dispersos podem participar de uma sobreposição Pastry; ela é organizada de forma totalmente automatica – quando novos nós entram na sobreposição, eles obtêm os dados necessários para construir uma tabela de roteamento e outros estados exigidos, a partir dos membros existentes nas *O(log N)* mensagens, onde *N* é o número de *hosts* participantes na sobreposição. No caso de falha, ou saída de um nó, os nós restantes podem detectar sua ausência e reconfigurar cooperativamente, para refletir as mudanças exigidas na estrutura de roteamento, em um número de mensagens semelhante.

Algoritmo de roteamento ◊ O algoritmo de roteamento completo envolve o uso de uma tabela de roteamento em cada nó para direcionar eficientemente as mensagens, mas para os propósitos da explicação, descreveremos o algoritmo em dois estágios. O primeiro descreve uma forma simplificada do algoritmo, que direciona as mensagens correta, mas ineficientemente, sem uma tabela de roteamento, e o segundo estágio descreve o algoritmo de roteamento completo, que direciona uma requisição para qualquer nó em *O(log N)* mensagens.

Estágio I. Cada nó ativo armazena um conjunto de folhas – um vetor *L* (de tamanho 2*l*) contendo os GUIDs e endereços IP dos nós cujos GUIDs são numericamente mais próximos nos dois lados de si próprio (*l* acima e *l* abaixo). Os conjuntos de folhas são mantidos pelo Pastry à medida que os nós se associam e saem. Mesmo após a falha de um nó, eles serão corrigidos dentro de um curto espaço de tempo. (A recuperação de falha será discutida a seguir.) Portanto, é uma constante do sistema Pastry que os conjuntos de folhas reflitam um estado recente do sistema e que convirjam para o estado corrente, em face de falhas de até alguma taxa máxima.

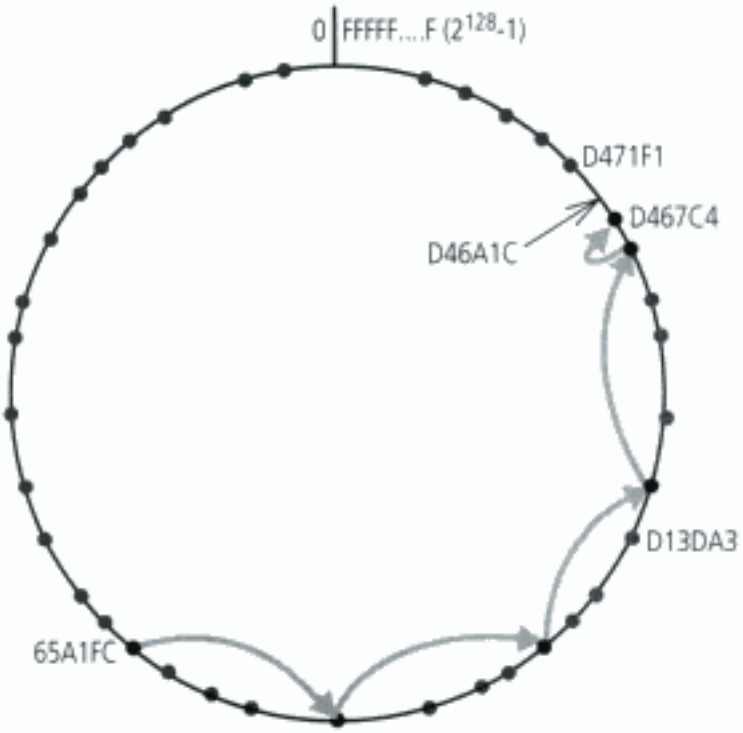
O espaço de GUID é tratado de forma circular: o vizinho inferior do GUID 0 é 2^{128}-1. A Figura 10.6 apresenta uma visão dos nós ativos distribuídos nesse espaço de endereço circular. Como todo conjunto de folhas inclui os GUIDs e os endereços IP dos vizinhos imediatos do nó corrente, um sistema Pastry com conjuntos de folhas corretos de tamanho de pelo menos 2 pode direcionar mensagens para qualquer GUID de maneira trivial, como segue.

Qualquer nó *A* que receba uma mensagem *M*, com endereço de destino *D*, direciona a mensagem comparando *D* com seu próprio GUID *A* e com cada um dos GUIDs em seu conjunto de folhas, a seguir, encaminha *M* para o nó dentre eles que esteja numericamente mais próximo a *D*. A Figura 10.6 ilustra isso para um sistema Pastry com $l = 4$. (Em instalações típicas reais do Pastry, $l = 8$.) Com base na definição de conjuntos de folhas, podemos concluir que, a cada etapa, *M* é encaminhada para um nó que está mais próximo a *D* do que o nó corrente, e que esse processo finalmente enviará *M* para o nó ativo mais próximo a *D*. Mas tal esquema de roteamento é claramente muito ineficiente, exigindo *~N/2l hops* para enviar uma mensagem em uma rede com *N* nós.

Estágio II. A segunda parte de nossa explicação descreve o algoritmo Pastry completo, e mostra como o roteamento eficiente é obtido com a ajuda de tabelas de roteamento.

Cada nó Pastry mantém uma tabela de roteamento estruturada em árvore, fornecendo GUIDs e endereços IP para um conjunto de nós espalhados pela gama inteira de 2^{128} valores de GUID possíveis, com densidade de cobertura maior para os GUIDs numericamente mais próximos de si próprios.

A Figura 10.7 mostra a estrutura da tabela de roteamento para um nó específico, e a Figura 10.8 ilustra as ações do algoritmo de roteamento. A tabela de roteamento é estruturada como segue: os GUIDs são vistos como valores hexadecimais e a tabela classifica os GUIDs com base em seus prefixos hexadecimais. A tabela tem tantas linhas quantos são os dígitos hexadecimais em um GUID; portanto, para o protótipo de sistema Pastry que estamos descrevendo, existem 128/4 = 32 linhas. Qualquer linha *n* contém 15 entradas – uma para cada valor possível do $n^{ésimo}$ dígito hexadecimal, excluindo o valor no GUID do nó local. Cada entrada na tabela aponta para um dos, potencialmente, muitos nós cujos GUIDs têm o prefixo relevante.



Os pontos representam nós ativos. O espaço é considerado circular: o nó 0 é adjacente ao nó (2^{128}-1). O diagrama ilustra o roteamento de uma mensagem do nó 65A1FC para D46A1C, usando apenas informações de conjunto de folhas, pressupondo conjuntos de folhas de tamanho 8 (l = 4). Esse é um tipo degenerado de roteamento que tem uma capacidade de escalabilidade muito baixa. Ele não é usado na prática.

Figura 10.6 O roteamento circular, puro, é correto, mas ineficiente.

O processo de roteamento, em qualquer nó A, usa as informações de sua tabela de roteamento R e o conjunto de folhas L para tratar cada requisição de uma aplicação e cada mensagem recebida de outro nó, de acordo com o algoritmo mostrado na Figura 10.9.

Podemos ter certeza de que o algoritmo terá êxito no envio de M para seu destino, pois as linhas 1, 2 e 7 executam as ações mencionadas no Estágio I de nossa descrição anterior e mostramos que esse é um algoritmo de roteamento completo, embora ineficiente. As etapas restantes são projetadas para usar a tabela de roteamento para melhorar o desempenho do algoritmo, reduzindo o número de *hops* exigidos.

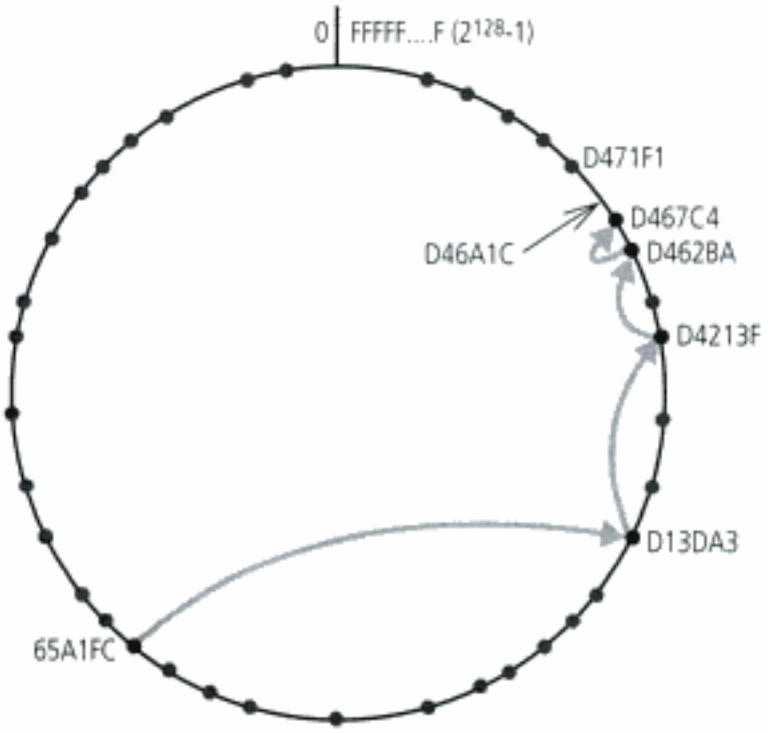
As linhas 4–5 entram em ação quando D não cai dentro do intervalo numérico do conjunto de folhas do nó corrente e as entradas da tabela de roteamento relevante estão disponíveis. A seleção de um destino para o próximo *hop* envolve a comparação dos dígitos hexadecimais de D com os de A (o GUID do nó corrente), da esquerda para a direita, para descobrir o comprimento p de seu prefixo comum mais longo. Esse comprimento é usado então como um deslocamento de linha, junto com o primeiro dígito não correspondente de D, como deslocamento de coluna, para acessar o elemento exigido da tabela de roteamento. A construção da tabela garante que esse elemento (se não estiver vazio) contém o endereço IP de um nó cujo GUID tem p+1 dígitos de prefixo em comum com D.

A linha 7 é usada quando D cai fora do intervalo numérico do conjunto de folhas e não existe uma entrada relevante da tabela de roteamento. Esse caso é raro; ele surge apenas quando os nós falharam recentemente e a tabela ainda não foi atualizada. O algoritmo de roteamento é capaz de prosseguir, varrendo o conjunto de folhas e a tabela de roteamento, e encaminhando M para outro nó cujo GUID tenha p dígitos de prefixo correspondentes, mas seja numericamente mais próximo a D. Se esse nó estiver em L, então estamos seguindo o procedimento do Estágio I, ilustrado na Figura 10.6. Se ele estiver em R, então deve ser mais próximo a D do que qualquer nó em L; portanto, estamos aperfeiçoando o Estágio I.

p =	*Prefixos de GUID e manipuladores de nós n correspondentes*															
0	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
	n	*n*	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*
1	*60*	*61*	*62*	*63*	*64*	*65*	*66*	*67*	*68*	*69*	*6A*	*6B*	*6C*	*6F*	*6E*	*6F*
	n	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*
2	*650*	*651*	*652*	*653*	*654*	*655*	*656*	*657*	*658*	*659*	*65A*	*65B*	*65C*	*65D*	*65E*	*65F*
	n	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*
3	*65A0*	*65A1*	*65A2*	*65A3*	*65A4*	*65A5*	*65A6*	*65A7*	*65A8*	*65A9*	*65AA*	*65AB*	*65AC*	*65AD*	*65AE*	*65AF*
	n		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*

A tabela de roteamento está localizada em um nó cujo GUID começa em 65A1. Os dígitos estão em hexadecimal. As letras *n* representam pares [GUID, endereço IP] especificando o próximo *hop* a ser dado pelas mensagens endereçadas aos GUIDs que correspondem a cada prefixo dado. As entradas sombreadas indicam que o prefixo corresponde ao GUID corrente até o valor dado de *p*: a próxima linha abaixo, ou o conjunto de folhas deve ser examinado para encontrar uma rota. Embora existam no máximo 128 linhas na tabela, apenas $log_{16}N$ linhas serão preenchidas, em média, em uma rede com *N* nós ativos.

Figura 10.7 **Primeiras quatro linhas de uma tabela de roteamento Pastry.**



Roteamento de uma mensagem do nó 65A1FC para D46A1C. Com a ajuda de uma tabela de roteamento adequadamente preenchida, a mensagem pode ser enviada em ~ $log_{16}(N)$ *hops*.

Figura 10.8 **Exemplo de roteamento do Pastry.**

Para manipular uma mensagem M endereçada para um nó D (onde $R[p,i]$ é o elemento na coluna i, linha p da tabela de roteamento):

```
1. Se (L_{-l} < D < L_l) { // o destino está dentro do conjunto de folhas ou é o nó corrente.
2.     Encaminha M para o elemento L_i do conjunto de folhas com GUID mais próximo a D ou
       o nó corrente A.
3. } senão { // usa a tabela de roteamento para enviar M para um nó com um GUID mais próximo
4.     localiza p, o comprimento do prefixo comum mais longo de D e A; e i, o (p+1)ésimo dígito
       hexadecimal de D.
5.     Se (R[p,i] ≠ null) encaminha M para R[p,i] // direciona M para um nó com um prefixo
       comum mais longo.
6.     senão { // não existe nenhuma entrada na tabela de roteamento
7.         Encaminha M para qualquer nó em L, ou R, com um prefixo comum de comprimento
           p, mas com um GUID numericamente mais próximo.
       }
   }
```

Figura 10.9 Algoritmo de roteamento do Pastry.

Integração de *hosts* ◊ Os nós novos usam um protocolo de associação para adquirir seus conteúdos de tabela de roteamento e conjunto de folhas, e para notificar outros nós das alterações que devem fazer em suas tabelas. Primeiramente, o novo nó calcula um GUID conveniente (normalmente, aplicando a função de resumo SHA-1 na chave pública do nó) e, então, estabelece contato com um nó Pastry vizinho. (Aqui, usamos o termo *vizinho* para nos referirmos à distância da rede; isto é, um pequeno número de *hops* de rede, ou atraso de transmissão baixo; veja o quadro intitulado *Algoritmo do vizinho mais próximo*, a seguir.)

Suponha que o GUID do novo nó seja X e que o nó mais próximo que ele contata tenha o GUID A. O nó X envia uma mensagem de requisição *join* (associação) para A, fornecendo X como destino. A envia a mensagem *join*, via Pastry, da maneira normal. O Pastry direcionará a mensagem *join* para o nó existente cujo GUID é numericamente mais próximo a X; vamos chamar esse nó de destino de Z.

A, Z e todos os nós (B, C,...) pelos quais a mensagem *join* é direcionada em seu caminho até Z, acrescentam uma etapa adicional no algoritmo normal de roteamento Pastry que resulta na transmissão do conteúdo da parte relevante de suas tabelas de roteamento e conjuntos de folhas para X. O nó X examina essas informações e constrói sua própria tabela de roteamento e conjunto de folhas a partir delas, solicitando, se necessário, informações adicionais de outros nós.

Para ver como X constrói sua tabela de roteamento, note que a primeira linha da tabela depende do valor do GUID de X e que, para minimizar as distâncias de roteamento, a tabela deve ser construída para direcionar as mensagens por meio de nós vizinhos, quando possível. Mas A é vizinho de X; portanto, a primeira linha da tabela de A é uma boa escolha inicial para a primeira linha da tabela de X, X_0. Por outro lado, a tabela de A provavelmente não é relevante para a segunda linha X_1, pois os GUIDs de X e de A podem não compartilhar o mesmo primeiro dígito hexadecimal. Mas o algoritmo de roteamento garante que os GUIDs de X e de B compartilham o mesmo primeiro dígito e isso significa que a

Algoritmo do vizinho mais próximo

O novo nó deve ter o endereço de pelo menos um nó Pastry existente, mas pode não ser um vizinho. Para garantir que nós mais próximos sejam conhecidos, o Pastry inclui um algoritmo do "vizinho mais próximo". Isso é feito medindo-se recursivamente o atraso da viagem de ida e volta de uma mensagem de teste (*probe*) enviada periodicamente para cada membro do conjunto de folhas do nó Pastry mais próximo conhecido, no momento.

segunda linha da tabela de roteamento de B, B_1, é um valor inicial conveniente para X_1. Analogamente, C_2 é conveniente para X_2, e assim por diante.

Além disso, recordando as propriedades dos conjuntos de folhas, note que, como o GUID de Z é numericamente mais próximo ao de X, o conjunto de folhas de X deve ser semelhante ao de Z. Na verdade, o conjunto de folhas ideal de X será diferente do de Z apenas por um membro. Portanto, o conjunto de folhas de Z é aceito como uma aproximação inicial adequada que finalmente será otimizada por meio da interação com seus vizinhos, conforme descrito sob o título Tolerância a falhas, a seguir.

Finalmente, uma vez que X tiver construído seu conjunto de folhas, e sua tabela de roteamento, da maneira descrita anteriormente, ele enviará seus conteúdos para todos os nós identificados no conjunto de folhas e na tabela de roteamento, e eles ajustarão suas próprias tabelas para incorporar o novo nó. A tarefa inteira de incorporação de um novo nó na infra-estrutura Pastry exige a transmissão de $O(\log N)$ mensagens.

Falha ou saída de *host* ◊ Os nós na infra-estrutura Pastry podem falhar, ou sair sem aviso. Um nó Pastry é considerado defeituoso quando seus vizinhos imediatos (no espaço de GUID) não podem mais se comunicar com ele. Quando isso ocorre, é necessário reparar os conjuntos de folhas que contêm o GUID do nó defeituoso.

Para reparar seu conjunto de folhas L, o nó que descobre a falha procura um nó ativo próximo ao nó defeituoso em L e solicita uma cópia do conjunto de folhas desse nó, L'. L' conterá uma seqüência de GUIDs que sobrepõem parcialmente os que estão em L, incluindo um que tenha um valor apropriado para substituir o nó defeituoso. Outros nós da vizinhança são então informados da falha e executam um procedimento semelhante. Esse procedimento de reparo garante que os conjuntos de folhas sejam reparados, a não ser que l nós de numeração adjacente falhem simultaneamente.

Os reparos nas tabelas de roteamento são feitos quando descobertos. O roteamento de mensagens pode prosseguir com algumas entradas da tabela de roteamento que não estão mais ativas – tentativas de roteamento malsucedidas resultam no uso de uma entrada diferente da mesma linha de uma tabela de roteamento.

Localidade ◊ A estrutura de roteamento Pastry é altamente redundante: existem muitas rotas entre cada par de nós. A construção das tabelas de roteamento tem como objetivo levar em conta essa redundância para reduzir os tempos de transmissão de mensagem reais, explorando as propriedades de localidade dos nós na rede de transporte subjacente (que normalmente é um subconjunto de nós na Internet).

Lembramos que cada linha em uma tabela de roteamento contém 16 entradas. As entradas da $i^{ésima}$ linha fornecem os endereços de 16 nós, com GUIDs com i-1 dígitos hexadecimais iniciais correspondentes ao GUID do nó corrente, e um $i^{ésimo}$ dígito que recebe cada um dos valores hexadecimais possíveis. Uma sobreposição Pastry preenchida adequadamente conterá muito mais nós do que podem caber em uma tabela de roteamento individual, e quando uma nova tabela de roteamento está sendo construída, é feita uma escolha para cada posição entre várias candidatas (extraídas das informações de roteamento fornecidas pelos outros nós), com base em um algoritmo *Proximity Neighbour Selection* (seleção de vizinho pela proximidade) [Gummadi *et al.* 2003]. É usada uma métrica de localidade (número de *hops* de IP ou latência medida) para comparar as candidatas e o nó mais próximo disponível é escolhido. Como as informações disponíveis não são abrangentes, esse mecanismo não pode produzir roteamentos globalmente perfeitos, mas simulações têm mostrado que ele resulta em rotas que, em média, são apenas cerca de 30–50% mais longas do que a ótima.

Tolerância a falhas ◊ Conforme descrito anteriormente, o algoritmo de roteamento Pastry presume que todas as entradas nas tabelas de roteamento e conjuntos de folhas se referem a nós ativos e funcionando corretamente. Todos os nós enviam mensagens de pulsação (isto é, mensagens enviadas em intervalos de tempo fixos para indicar que o remetente está vivo) para os nós vizinhos em seus conjuntos de folhas, mas as informações sobre nós defeituosos detectadas dessa maneira podem não ser disseminadas rápido o suficiente para eliminar erros de roteamento. Isso também não leva em conta nós mal-intencionados que podem tentar interferir no roteamento correto. Para superar esses problemas, espera-se que os clientes que dependam do envio confiável da mensagem empreguem um mecanismo de entrega pelo menos uma vez (veja a Seção 5.2.4), e a repitam várias vezes, na ausência de uma resposta. Isso fornecerá ao Pastry uma janela de tempo mais longa para detectar e reparar falhas de nó.

Para tratar com todas as falhas, ou nós mal-intencionados restantes, um pequeno grau de aleatoriedade é introduzido no algoritmo de seleção de rota descrito na Figura 10.9. Basicamente, a etapa da linha 5 da Figura 10.9 é modificada para uma pequena proporção de casos selecionados aleatoriamente para produzir um prefixo comum que seja menor do que o comprimento máximo. Isso resulta no uso de um roteamento extraído de uma linha anterior da tabela de roteamento, produzindo roteamento menos perfeito, mas diferente da versão padrão do algoritmo. Com essa variação aleatória no algoritmo de roteamento, as retransmissões do cliente devem finalmente ter sucesso, mesmo na presença de um pequeno número de nós mal-intencionados.

Dependabilidade ◊ Os autores do Pastry desenvolveram uma versão atualizada, chamada MSPastry [Castro *et al.* 2003], que usa o mesmo algoritmo de roteamento e métodos de gerenciamento de *host* semelhantes, mas que inclui algumas medidas de dependabilidade adicionais e algumas otimizações de desempenho nos algoritmos de gerenciamento de *host*.

As medidas de dependabilidade incluem o uso de mensagens de confirmação em cada *hop* no algoritmo de roteamento. Se o *host* que está fazendo o envio não receber uma mensagem de confirmação após um tempo limite especificado, ele seleciona uma rota alternativa e retransmite a mensagem. O nó que deixou de enviar a confirmação é então registrado como uma suspeita de falha.

Para detectar nós defeituosos, cada nó Pastry envia periodicamente uma mensagem de pulsação para seu vizinho imediato à esquerda (isto é, com um GUID menor) no conjunto de folhas. Cada nó também registra o tempo da última mensagem de pulsação recebida de seu vizinho imediato à direita (com um GUID maior). Se o intervalo de tempo desde a última pulsação ultrapassar o tempo limite, o nó de detecção começa um procedimento de reparo que envolve entrar em contato com os nós restantes do conjunto de folhas, com uma notificação sobre o nó defeituoso e uma requisição de substitutos sugeridos. Mesmo no caso de várias falhas simultâneas, esse procedimento termina com todos os nós no lado do nó defeituoso tendo conjuntos de folhas que contêm os l nós ativos com os GUIDs mais próximos.

Vimos que o algoritmo de roteamento pode funcionar corretamente usando apenas conjuntos de folhas; mas a manutenção das tabelas de roteamento é importante para o desempenho. Nós com suspeita de defeito nas tabelas de roteamento são examinados de maneira semelhante ao conjunto de folhas, e se eles deixam de responder, suas entradas na tabela de roteamento são substituídas por uma alternativa conveniente, obtida de um nó vizinho. Além disso, um protocolo de "fofoca" (*gossip*) simples (Seção 14.4.1) é usado para trocar informações da tabela de roteamento periodicamente entre os nós, visando reparar entradas defeituosas e evitar a lenta deterioração das propriedades da localidade. O protocolo de "fofoca" é executado aproximadamente a cada 20 minutos.

Trabalho de avaliação ◊ Castro e seus colegas realizaram uma exaustiva avaliação de desempenho do MSPastry, tendo como objetivo determinar o impacto sobre o desempenho e a confiança da taxa de associação/saída de *host* e dos mecanismos de dependabilidade associados [Castro *et al.* 2003].

A avaliação foi realizada com a execução do sistema MSPastry através de um simulador, funcionando em uma única máquina, que simulava uma grande rede de *hosts* com a passagem de mensagens substituída por atrasos de transmissão simulados. A simulação modelava de forma realista o comportamento de associação/saída de *hosts* e atrasos de transmissão de IP baseados em parâmetros de instalações reais.

Todos os mecanismos de dependabilidade do MSPastry foram incluídos, com intervalos realistas para mensagens de teste (*probe*) e pulsação. O trabalho de simulação foi validado pela comparação com medidas tiradas com o MSPastry executando uma carga de uma aplicação real em uma rede interna com 52 nós.

Resumimos aqui apenas os principais resultados.

Dependabilidade: com uma taxa de perda de mensagens de IP pressuposta de 0%, o MSPastry deixou de enviar 1,5 requisições em 100.000 (supostamente devido à indisponibilidade de *hosts* de destino) e todas as requisições que foram enviadas chegaram ao nó correto.

Com uma taxa de perda de mensagens de IP pressuposta de 5%, o MSPastry perdeu cerca de 3,3 requisições em 100.000 e 1,6 requisições em 100.000 foram enviadas para o nó errado. O uso de confirmações por *hop* no MSPastry garante que todas as mensagens perdidas, ou direcionadas para o lugar errado, sejam finalmente retransmitidas e cheguem ao nó correto.

Desempenho: a métrica usada para avaliar o desempenho do MSPastry é chamada de *penalidade de atraso relativa* (RDP – *Relative Delay Penalty*) [Chu *et al.* 2000] ou *distensão*. A RDP é uma medida direta do custo extra acarretado no emprego de uma camada de sobreposição de roteamento. É a relação entre o atraso médio no envio de uma requisição pela sobreposição de roteamento e no envio de uma mensagem semelhante entre os mesmos dois nós por meio de UDP/IP. Os valores de RDP observados para o MSPastry sob cargas simuladas variaram de ~1,8, com perda zero de mensagens de rede, a ~2,2, com 5% de perda de mensagens de rede.

Sobrecargas: a carga de rede extra gerada pelo *tráfego de controle* – mensagens envolvidas na manutenção de conjuntos de folhas e tabelas de roteamento – foi de menos de duas mensagens por minuto por nó. Tanto a RDP como o tráfego de controle aumentaram significativamente para comprimentos de sessão de menos de cerca de 60 minutos, devido às sobrecargas de configuração iniciais.

De modo geral, esses resultados mostram que podem ser construídas camadas de sobreposição de roteamento que obtêm bom desempenho e alta confiança com milhares de nós operando em ambientes realistas. Mesmo com comprimentos de sessão médios menores do que 60 minutos e altas taxas de erro de rede, o sistema degrada pouco, continuando a fornecer um serviço eficiente.

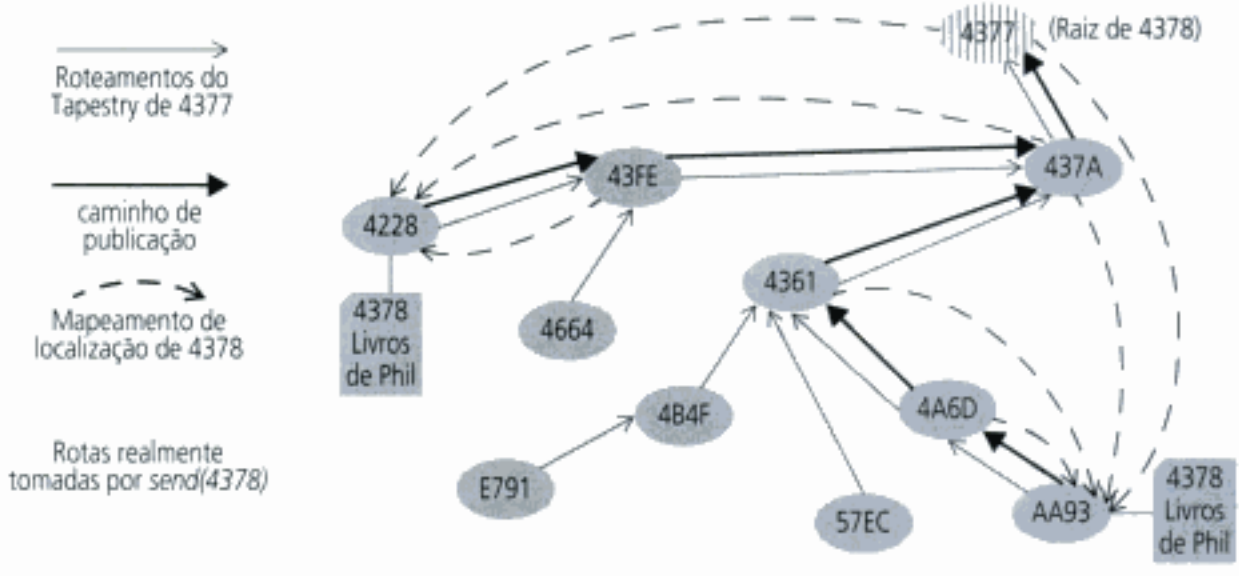
10.5.2 Tapestry

O Tapestry implementa uma tabela de resumo distribuída e direciona as mensagens para os nós com base nos GUIDs associados aos recursos, usando roteamento baseado em prefixo de maneira semelhante ao Pastry. Mas a API do Tapestry oculta das aplicações a tabela de resumo distribuída atrás da interface DOLR, como aquela que aparece na Figura 10.5. Os nós que contêm recursos usam a primitiva *publish(GUID)* para torná-los conhecidos do Tapestry; os proprietários de recursos continuam responsáveis por seu armazenamento. Os recursos replicados são publicados com o mesmo GUID em cada nó que contém uma réplica, resultando em várias entradas na estrutura de roteamento do Tapestry.

Isso proporciona às aplicações Tapestry uma flexibilidade adicional: elas podem colocar réplicas próximas (em distância da rede) aos usuários freqüentes dos recursos para reduzir as latências e minimizar as cargas de rede, ou para garantir tolerância de falhas de rede e *host*. Mas essa distinção entre o Pastry e o Tapestry não é fundamental: as aplicações Pastry podem obter flexibilidade semelhante fazendo os objetos associados aos GUIDs simplesmente agir como *proxies* de objetos em nível de aplicação mais complexos e o Tapestry pode ser usado para implementar uma tabela de resumo distribuída em termos de sua API DOLR [Dabek *et al.* 2003].

No Tapestry são usados identificadores de 160 bits para fazer referência tanto a objetos como aos nós que executam ações de roteamento. Os identificadores são *NodeIds*, que se referem aos computadores que executam operações de roteamento, ou *GUIDs*, que se referem aos objetos. Para qualquer recurso com GUID G existe um nó-raiz único com GUID R_G numericamente mais próximo a G. Os *hosts* H que contêm réplicas de G ativam *publish(G)* periodicamente para garantir que os *hosts* recém-chegados saibam da existência de G. A cada invocação de *publish(G)*, uma mensagem de publicação é direcionada do invocador para o nó R_G. Na recepção de uma mensagem de publicação, R_G insere (G, IP_H), o mapeamento entre G e o endereço IP do *host* que está fazendo o envio, em sua tabela de roteamento, e cada nó ao longo do caminho de publicação armazena o mesmo mapeamento na cache. Esse processo está ilustrado na Figura 10.10. Quando os nós contêm vários mapeamentos (G, IP) para o mesmo GUID, eles são ordenados pela distância da rede (tempo da viagem de ida e volta) até o endereço IP. Para objetos replicados isso implica na seleção da réplica do objeto mais próxima disponível como destino das mensagens subseqüentes enviadas ao objeto.

Zhao *et al.* [2004] fornecem detalhes completos dos algoritmos de roteamento e do gerenciamento de tabelas de roteamento do Tapestry em face da chegada e saída de nós. Esse artigo inclui amplos dados de avaliação de desempenho, baseados em simulação de redes Tapestry de larga escala, mostrando que seu desempenho é semelhante ao do Pastry. Na Seção 10.6.2, descreveremos o sistema de armazenamento de arquivos OceanStore, que foi construído e implantado sobre o Tapestry.



Réplicas do arquivo *Livros de Phil* (G=4378) são contidas nos nós 4228 e AA93. O nó 4377 é o nó raiz do objeto 4378. Os roteamentos do Tapestry mostrados são algumas das entradas nas tabelas de roteamento. Os caminhos de publicação mostram as rotas seguidas pelas mensagens de publicação armazenando mapeamentos de localização armazenados em cache para o objeto 4378. Os mapeamentos de localização são usados subseqüentemente para direcionar as mensagens enviadas para 4378.

Figura 10.10 **Roteamento do Tapestry. De [*zhoo et al. 2004*].**

10.6 Estudo de caso: Squirrel, OceanStore, Ivy

Os sistemas *peer-to-peer* de larga escala ainda não representam uma tecnologia consagrada. Sua implantação mais ampla tem sido em aplicações para *download* de arquivos feitos por usuários finais em sistemas como Napster, Freenet, Gnutella, Kazaa e BitTorrent. Mas esses sistemas não empregam camadas de sobreposição de roteamento separadas; portanto, suas avaliações de desempenho são difíceis de extrapolar para outras aplicações.

As camadas de sobreposição de roteamento descritas na seção anterior foram exploradas em várias experiências de aplicação e os aplicativos resultantes foram extensivamente avaliados. Escolhemos três deles para estudar melhor, o serviço de uso de cache web Squirrel, baseado no Pastry, e os sistemas de armazenamento de arquivo OceanStore e Ivy.

10.6.1 Cache web Squirrel

Os autores do Pastry desenvolveram o serviço *peer-to-peer* de cache web Squirrel para ser empregado em redes locais de computadores pessoais [Iyer *et al.* 2002]. Em redes locais médias e grandes, o uso de cache web normalmente é realizado com um computador servidor dedicado ou com um *cluster*. O sistema Squirrel executa a mesma tarefa, explorando armazenamento e recursos computacionais já disponíveis em computadores de *desktop* na rede local. Primeiramente, daremos uma breve descrição geral da operação de um serviço de uso de cache web e, em seguida, descreveremos em linhas gerais o projeto do Squirrel e examinaremos sua eficácia.

Uso de cache web ◊ Os navegadores web geram pedidos *GET* HTTP para objetos de Internet, como páginas HTML, imagens, etc. Esses objetos podem ser providos a partir da cache de um navegador na máquina cliente, *cache web proxy* – por um serviço sendo executado em outro computador na mesma rede local, ou em um nó vizinho na Internet, ou do servidor web *de origem* – o servidor cujo nome de domínio é incluído nos parâmetros da requisição GET – dependendo de qual contenha uma cópia atualizada do objeto. As caches local e de *proxy* contêm cada uma um conjunto de objetos recen-

temente recuperados, organizados para uma pesquisa rápida pelo URL. Alguns objetos não podem ser colocados na cache, pois são gerados dinamicamente pelo servidor em resposta a cada requisição.

Quando a cache de um navegador, ou a cache web do *proxy*, recebe uma requisição *GET*, existem três possibilidades: o objeto solicitado não pode ser atingido, há uma falta na cache ou o objeto é encontrado na cache. Nos dois primeiros casos, a requisição é encaminhada para o próximo nível, em direção ao servidor web de origem. Quando o objeto solicitado é encontrado em uma cache, o caráter atual da cópia colocada na cache deve ser testado.

Os objetos web são armazenados em servidores web e em servidores de cache com alguns valores adicionais de metadados, incluindo uma indicação de tempo fornecendo a *data da última modificação T* e possivelmente um *tempo de vida t* ou uma *eTag* (um código de resumo calculado a partir do conteúdo de uma página web). Esses itens de metadados são fornecidos pelo servidor de origem quando um objeto é retornado para um cliente.

Para objetos que têm um tempo de vida *t* associado, o objeto é considerado atual se *T+t* for posterior ao tempo corrente real. Para objetos sem tempo de vida, é usado um valor estimado para *t* (freqüentemente de apenas alguns segundos). Se o resultado dessa avaliação de atualidade for positivo, o objeto colocado na cache será retornado para o cliente, sem contato com o servidor web de origem. Caso contrário, uma requisição *GET* condicional (*cGET*) será emitida para validação no próximo nível. Existem dois tipos básicos de requisição *cGET*: uma requisição *If-Modified-Since*, contendo a indicação de tempo da última modificação conhecida, e uma requisição *If-None-Match*, contendo uma *eTag* representando o conteúdo do objeto. Essa requisição *cGET* pode ser servida por outra cache web ou pelo servidor de origem. Uma cache web que recebe uma requisição *cGET* e não tem uma cópia atualizada do objeto encaminha a requisição para o servidor web de origem. A resposta contém o objeto inteiro, ou uma mensagem *not-modified* se o objeto colocado na cache estiver intacto.

Quando um objeto recentemente modificado é recebido do servidor de origem, se for o caso, ele pode ser adicionado ao conjunto de objetos da cache local (removendo objetos mais antigos que ainda são válidos, se necessário), junto com uma indicação de tempo, um tempo de vida e uma *eTag*, se estiver disponível.

O esquema descrito anteriormente é a base da operação dos serviços de uso de cache web de *proxies* centralizados, implantados na maioria das redes locais que suportam grandes números de clientes web. Normalmente, as caches web de *proxy* são implementadas como um processo *multi-threaded*, executando em um único *host* dedicado, ou um conjunto de processos executando em um cluster de computadores, e, em ambos os casos, exigem uma quantidade substancial de recursos computacionais dedicados.

Squirrel ◊ O serviço de uso de cache web Squirrel executa as mesmas funções, usando uma pequena parte dos recursos de cada computador cliente em uma rede local. A função de resumo segura SHA-1 é aplicada ao URL de cada objeto armazenado na cache para produzir um GUID Pastry de 128 bits. Como o GUID não é usado para validar seu conteúdo, ele não precisa ser baseado no conteúdo do objeto inteiro, como acontece nos outros aplicativos Pastry. Os autores do Squirrel baseiam sua justificativa para isso no "argumento do princípio fim-a-fim" (Seção 2.2.1), afirmando que a autenticidade de uma página web pode ser comprometida em muitos pontos de sua jornada do *host* até o cliente; a autenticação de páginas armazenadas na cache acrescenta pouco em qualquer garantia global de autenticidade – o protocolo HTTPS (incorporando Transport Layer Security, Seção 7.6.3) deve ser usado para se obter uma garantia muito melhor para as interações que o exigem.

Na implementação mais simples do Squirrel – que se mostrou ser a mais eficiente –, o nó cujo GUID é numericamente mais próximo ao GUID de um objeto se torna o *nó de base* desse objeto, responsável por conter toda cópia do objeto colocada na cache.

Os nós clientes são configurados de forma a incluir um processo Squirrel local *proxy*, que assume a responsabilidade pelo armazenamento na cache local e remota de objetos web. Se uma cópia atualizada de um objeto solicitado não está na cache local, o Squirrel direciona uma requisição *Get*, ou uma requisição *cGet* (quando existe uma cópia antiga do objeto na cache local), por meio do Pastry, para o nó de base. Se o nó de base tiver uma cópia atualizada, ele responderá diretamente para o cliente com uma mensagem *not-modified*, ou com uma cópia atualizada, conforme for apropriado. Se o nó de base tiver uma cópia antiga, ou não tiver nenhuma cópia do objeto, ele emitirá uma requisição *cGet* ou *Get* para o servidor de origem, respectivamente. Se o servidor de origem puder, responderá com uma mensagem *not-modified*, ou com uma cópia do objeto. No primeiro caso, o nó de base revalida

sua entrada de cache e encaminha uma cópia do objeto para o cliente. No último caso, ele encaminha uma cópia do novo valor para o cliente e armazena uma cópia em sua cache local, caso o objeto possa ser colocado na cache.

Avaliação do Squirrel ◊ O Squirrel foi avaliado por meio de simulação, usando cargas modeladas derivadas de traços existentes da atividade de caches web de *proxies* centralizados, em dois ambientes de trabalho reais dentro da Microsoft, um com 105 clientes ativos (em Cambridge) e o outro com mais de 36.000 (em Redmond). A avaliação comparou o desempenho de uma cache web Squirrel com uma centralizada, sob três aspectos:

A redução na largura de banda externa total usada: a largura de banda externa total é inversamente relacionada à taxa de acertos, pois são apenas as faltas de cache que geram requisições para servidores web externos. As taxas de acerto observadas para servidores de cache web centralizados foram de 29% (para Redmond) e 38% (para Cambridge). Quando os mesmos registros de atividade foram usados para gerar uma carga simulada para a cache Squirrel, com cada cliente contribuindo com 100 MB de armazenamento em disco, foram obtidas taxas de acerto muito semelhantes, de 28% (Redmond) e 37% (Cambridge). Conclui-se que a largura de banda externa seria reduzida por uma proporção semelhante.

A latência percebida pelos usuários para acesso a objetos web: o uso de uma sobreposição de roteamento resultou em várias transferências de mensagem (*hops* de roteamento) na rede local para transmitir uma requisição de um cliente para o *host* responsável por armazenar o objeto relevante (o nó de base) na cache. Os números médios de *hops* de roteamento observados na simulação foram de 4,11 *hops* para enviar uma requisição GET, no caso de Redmond, e 1,8 hops no caso de Cambridge, enquanto apenas uma única transferência de mensagem é exigida para acessar um serviço de cache centralizado.

Mas as transferências locais levam apenas alguns milisegundos com um hardware Ethernet moderno, incluindo o tempo de configuração da conexão TCP, enquanto as transferências de mensagem TCP remotas na Internet exigem de 10–100 ms. Portanto, os autores do Squirrel argumentam que a latência para acesso a objetos encontrados em cache é superada pela latência muito maior do acesso a objetos não encontrados em cache, proporcionando uma experiência para o usuário semelhante àquela apresentada com uma cache centralizada.

A carga computacional e de armazenamento imposta sobre os nós clientes: em cada nó, o número médio de requisições de cache recebidas de outros nós durante o período inteiro da avaliação foi extremamente baixo, com apenas 0,31 por minuto (Redmond), indicando que a proporção global de recursos de sistema consumidos é extremamente baixa.

Com base nas medidas descritas anteriormente, os autores do Squirrel concluíram que seu desempenho é comparável ao de uma cache centralizada. O Squirrel obtém uma redução na latência observada para acesso à página web próxima daquela que pode ser obtida por um servidor de cache centralizado, com uma cache dedicada de tamanho semelhante. A carga adicional imposta sobre os nós clientes é baixa e provavelmente será imperceptível para os usuários. O sistema Squirrel foi subseqüentemente implantado como cache web principal em uma rede local com 52 máquinas clientes, usando o Squirrel, e os resultados confirmaram suas conclusões.

10.6.2 Sistema de armazenamento de arquivos OceanStore

Os desenvolvedores do Tapestry projetaram e construíram um protótipo de sistema de armazenamento de arquivos *peer-to-peer*. Ao contrário do Past, ele suporta o armazenamento de arquivos mutáveis. O projeto OceanStore [Kubiatowicz *et al.* 2000, Kubiatowicz 2003, Rhea *et al.* 2001, Rhea *et al.* 2003] tem como objetivo fornecer um recurso de armazenamento persistente de grande escala, escalável por incrementos, para objetos de dados mutáveis, com persistência a longo prazo e confiabilidade em um ambiente de recursos de rede e computacionais em constante mudança. O OceanStore se destina a ser usado em uma variedade de aplicações, incluindo a implementação de um serviço de arquivos do tipo do NFS, hospedagem de correio eletrônico, bancos de dados e outras aplicações envolvendo o compartilhamento e o armazenamento persistente de grandes números de objetos de dados.

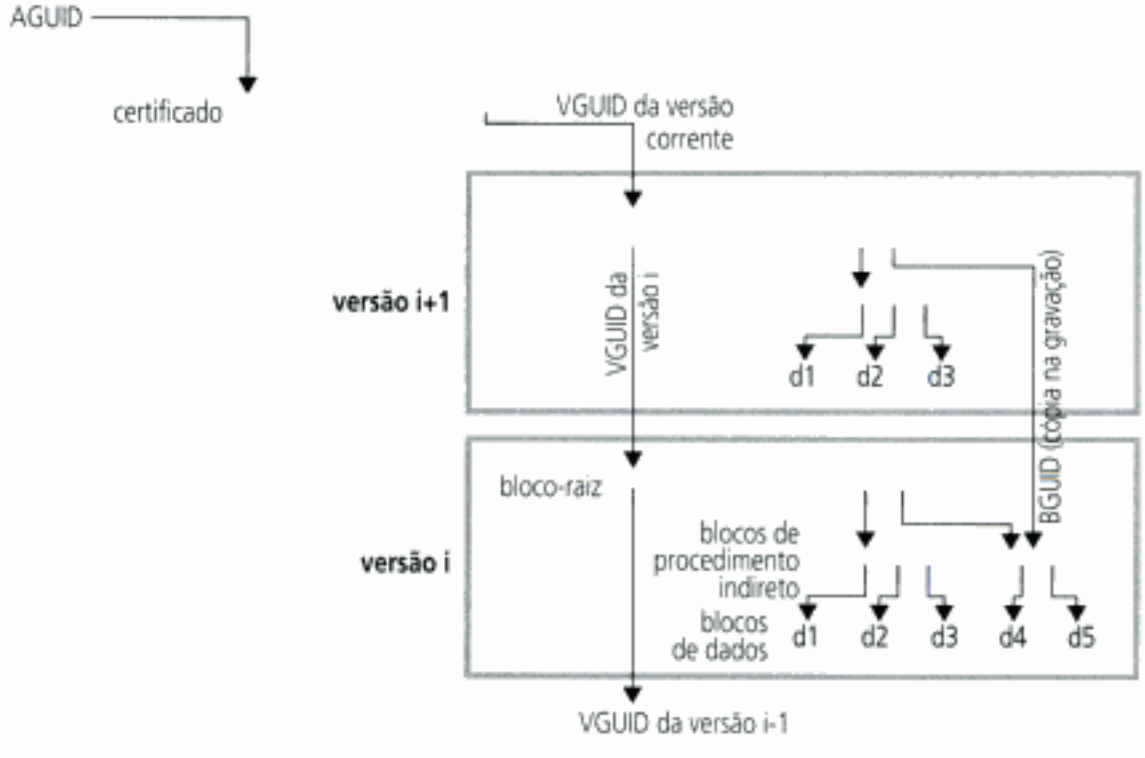
O projeto inclui preparativos para o armazenamento replicado de objetos de dados mutáveis e imutáveis. O mecanismo para manutenção da consistência entre as réplicas pode ser personalizado de acordo com as necessidades da aplicação, de uma maneira que foi inspirada pelo sistema Bayou (Seção 14.4.2). A privacidade e a integridade são obtidas por meio da criptografia dos dados e pelo uso de um protocolo de acordo bizantino (veja as Seções 11.5.3 e 11.5.4) para atualizações nos objetos replicados. Isso é necessário porque a confiabilidade dos *hosts* individuais não pode ser pressuposta.

Foi construído um protótipo do OceanStore, chamado Pond [Rhea *et al.* 2003]. Ele é suficientemente completo para suportar aplicativos e seu desempenho foi avaliado em uma variedade de comparativos para validar o projeto OceanStore e comparar seu desempenho com estratégias mais tradicionais. No restante desta seção, fornecemos um panorama do projeto OceanStore/Pond e resumimos os resultados da avaliação.

O Pond usa o mecanismo de sobreposição de roteamento Tapestry para colocar blocos de dados em nós distribuídos por toda a Internet e para enviar requisições a eles.

Organização do armazenamento ◊ Os objetos de dados OceanStore/Pond são análogos a arquivos, com seus dados armazenados em um conjunto de blocos. Mas cada objeto é representado como uma seqüência ordenada de versões imutáveis que são (em princípio) mantidas para sempre. Toda atualização em um objeto resulta na geração de uma nova versão. As versões compartilham os blocos inalterados, seguindo a técnica da cópia na gravação para criar e atualizar objetos, descrita na Seção 6.4.2. Assim, uma pequena diferença entre as versões exige apenas uma pequena quantidade de armazenamento adicional.

Os objetos são estruturados de uma maneira que lembra o sistema de armazenamento do Unix, com os blocos de dados organizados e acessados por meio de um bloco de metadados chamado de bloco-raiz e por blocos de procedimento indireto adicionais, se necessário (cf. *i-nodes* Unix). Outro nível de procedimento indireto é usado para associar um nome textual persistente, ou outro nome visível externamente (por exemplo, o nome de caminho de um arquivo), à seqüência de versões de um objeto de dados. A Figura 10.11 ilustra essa organização. Os GUIDs são associados ao objeto (um AGUID),



A versão i+1 foi atualizada nos blocos d1, d2 e d3. O certificado e os blocos-raízes incluem alguns metadados, não mostrados. Todas as setas não rotuladas são BGUIDs.

Figura 10.11 Organização do armazenamento de objetos do OceanStore.

ao bloco-raiz de cada versão do objeto (um VGUID), a blocos de procedimento indireto e a blocos de dados (BGUIDs). Várias réplicas de cada bloco são armazenadas nos pares de nós, selecionados de acordo com os critérios de localidade e disponibilidade de armazenamento, e seus GUIDs são publicados (usando a primitiva *publish()* da Figura 10.5) em cada um dos nós que contêm uma réplica para que o Tapestry possa ser usado pelos clientes para acessar os blocos.

São usados três tipos de GUIDs, conforme resumido na Figura 10.12. Os dois primeiros são GUIDs do tipo normalmente atribuído a objetos armazenados no Tapestry – eles são calculados a partir do conteúdo do bloco relevante, usando uma função de resumo segura para que eles possam ser usados posteriormente para autenticar e verificar a integridade do conteúdo. Os blocos a que eles fazem referência são necessariamente imutáveis, pois qualquer alteração no conteúdo de um bloco invalidaria o uso do GUID como token de autenticação.

O terceiro tipo de identificador usado é o AGUID. Ele se refere (indiretamente) ao fluxo inteiro de versões de um objeto, permitindo que os clientes acessem a versão corrente do objeto ou qualquer versão anterior. Como os objetos armazenados são mutáveis, os GUIDs usados para identificá-los não podem ser derivados de seus conteúdos, pois isso tornaria os GUIDs mantidos em índices, etc., obsoletos quando um objeto mudasse.

Em vez disso, quando um novo objeto de armazenamento é criado, um AGUID permanente é gerado pela aplicação de uma função de resumo segura em um nome específico da aplicação (por exemplo, um nome de arquivo) fornecido pelo cliente que está criando o objeto e de uma chave pública que representa o proprietário do objeto (veja a Seção 7.2.5). Em um aplicativo de sistema de armazenamento, um AGUID seria mantido nos diretórios para cada nome de arquivo.

A associação entre um AGUID e a seqüência de versões do objeto que ele identifica é gravada em um certificado assinado, que é armazenado e replicado por um esquema de replicação de cópia primária (também chamada de replicação passiva, veja a Seção 14.3.1). O certificado inclui o VGUID da versão corrente, e o bloco-raiz de cada versão contém o VGUID da versão anterior, de modo que existe um encadeamento de referências permitindo que os clientes que contenham um certificado percorram o encadeamento de versões inteiro (Figura 10.11). Um certificado assinado é necessário para garantir que a associação seja autêntica e que foi feita por um principal autorizado. É esperado que os clientes verifiquem isso. Quando é criada uma nova versão de um objeto, é gerado um novo certificado contendo o VGUID da nova versão, junto com uma indicação de tempo e um número de seqüência de versão.

O modelo de confiança dos sistemas *peer-to-peer* exige que a construção de cada novo certificado seja consentida (conforme descrito a seguir) entre um pequeno conjunto de *hosts* chamados de *anel interno*. Quando um novo objeto é armazenado no OceanStore, é selecionado um conjunto de *hosts* para atuar como anel interno desse objeto. Eles usam a primitiva *publish()* do Tapestry para tornar o AGUID do objeto conhecido para o Tapestry. Os clientes podem então usar o Tapestry para direcionar requisições do certificado do objeto para um dos nós do anel interno.

O novo certificado substitui a cópia primária antiga mantida em cada nó do anel interno e é disseminado para um número maior de cópias secundárias. Fica a cargo dos clientes determinar com que freqüência eles verificam a existência de uma nova versão (por exemplo, a maioria das instalações de NFS opera com uma janela de consistência de 30 segundos entre cliente e servidor, veja a Seção 8.3).

Como sempre acontece nos sistemas *peer-to-peer*, a confiança não pode ser depositada em nenhum *host* individual. A atualização de cópias primárias exige acordo de consenso entre os *hosts* do

Nome	*Significado*	*Descrição*
BGUID	GUID de bloco	Código de resumo seguro de um bloco de dados
VGUID	GUID de versão	BGUID do bloco-raiz de uma versão
AGUID	GUID ativo	Identifica exclusivamente todas as versões de um objeto

Figura 10.12 Tipos de identificador usados no OceanStore.

anel interno. Eles usam uma versão de uma máquina de estados, baseada no algoritmo de acordo bizantino, descrito por Castro e Liskov [2000], para atualizar o objeto e assinar o certificado. O uso de um protocolo de acordo bizantino garante que o certificado seja mantido corretamente, mesmo que alguns membros do anel interno falhem ou se comportem de maneira mal-intencionada. Como os custos computacionais e de comunicação do acordo bizantino sobem com o quadrado do número de *hosts* envolvidos, o número de *hosts* no anel interno é mantido pequeno, e o certificado resultante é replicado usando-se o esquema de cópia primária.

A realização de uma atualização também envolve a verificação dos direitos de acesso e a serialização de todas as outras gravações pendentes. Uma vez concluído o processo de atualização da cópia primária, os resultados são disseminados nas réplicas secundárias armazenadas nos *hosts* de fora do anel interno, usando uma árvore de roteamento *multicast* gerenciada pelo Tapestry.

Devido à sua natureza somente de leitura, os blocos de dados são replicados por um mecanismo diferente, de armazenamento mais eficiente. Ele é baseado na divisão de cada bloco em m fragmentos de igual tamanho, os quais são codificados usando *códigos de obliteração* [Weatherspoon e Kubiatowicz 2002] em n fragmentos, onde $n > m$. A principal propriedade do código de obliteração é que é possível reconstruir um bloco a partir de quaisquer m fragmentos. Em um sistema que usa código de obliteração, todos os objetos de dados permanecem disponíveis com a perda de até n-m *hosts*. Na implementação Pond, $m = 16$ e $n = 32$, de modo que, dobrando o custo de armazenamento, o sistema pode tolerar a falha de até 16 *hosts*, sem perda de dados. Os fragmentos são armazenados na rede usando o Tapestry para colocá-los e recuperá-los.

Esse alto nível de tolerância a falhas e disponibilidade de dados é obtido a certo custo na reconstrução de blocos a partir dos fragmentos do código de obliteração. Para minimizar o impacto disso, os blocos inteiros também são armazenados na rede usando o Tapestry. Como podem ser reconstruídos a partir de seus fragmentos, esses blocos são tratados como uma cache – eles não são tolerantes à falha e podem ser eliminados quando for exigido espaço de armazenamento.

Desempenho ◊ O Pond foi desenvolvido como protótipo para provar a exeqüibilidade de um serviço de arquivo *peer-to-peer* com capacidade de mudança de escala, em vez de uma implementação de produção. Ele é implementado em Java e inclui quase tudo do projeto descrito anteriormente. Ele foi avaliado em diversos comparativos destinados a isso e em uma simulação simples de cliente e servidor NFS em termos de objetos OceanStore. Os desenvolvedores testaram a simulação de NFS usando o *benchmark* do Andrew [Howard *et al.* 1988], que simula uma carga de trabalho de desenvolvimento de software. A tabela da Figura 10.13 mostra os resultados deste último. Eles foram obtidos usando-se um PC Pentium III de 1 GHz executando Linux. Os testes de rede local foram realizados usando uma Ethernet Gigabit e os resultados de rede de longa distância foram obtidos usando-se dois conjuntos de nós ligados pela Internet.

As conclusões tiradas pelos autores foram que o desempenho do OceanStore/Pond ao operar sobre uma rede de longa distância (isto é, a Internet) ultrapassa em muito o do NFS para leitura, e está dentro de um fator de três em relação ao NFS para atualização de arquivos e diretórios; os resultados em rede local são substancialmente piores. De modo geral, os resultados sugeriram que um serviço de arquivo *peer-to-peer* com escala de Internet, baseado no projeto OceanStore, seria uma solução eficiente para a distribuição de arquivos que não mudam com muita rapidez (como as cópias de páginas web armazenadas em cache). Seu potencial para ser empregado como uma alternativa para o NFS é questionável, mesmo para a rede de longa distância e é claramente não-competitivo para uso puramente local.

Esses resultados foram obtidos com blocos de dados armazenados sem fragmentação e replicação baseada em código de obliteração. O uso de chaves públicas contribui substancialmente para o custo computacional da operação do Pond. Os valores mostrados são para chaves de 512 bits, cuja segurança é boa, mas menos que ótima. Os resultados para chaves de 1024 bits foram substancialmente piores para as fases daqueles comparativos que envolviam atualizações de arquivo. Outros resultados obtidos com comparativos destinados para o propósito incluem a medida do impacto do processo de acordo bizantino sobre a latência das atualizações. Eles estão no intervalo de 100 ms a 10 segundos. Um teste de desempenho de saída de atualização atingiu um máximo de 100 atualizações/segundo.

	Rede local		Rede remota		
Fase	NFS Linux	Pond	NFS Linux	Pond	Operações predominantes no comparativo
1	0,0	1,9	0,9	2,8	Leitura e gravação
2	0,3	11,0	9,4	16,8	Leitura e gravação
3	1,1	1,8	8,3	1,8	Leitura
4	0,5	1,5	6,9	1,5	Leitura
5	2,6	21,0	21,5	32,0	Leitura e gravação
Total	4,5	37,2	47,0	54,9	

Os valores mostram tempos em segundos para executar as diferentes fases do benchmark do AFS. Ele tem cinco fases: (1) cria subdiretórios recursivamente; (2) copia uma árvore de diretórios; (3) examina o status de todos os arquivos da árvore sem examinar seus dados; (4) examina cada byte de dados em todos os arquivos; e (5) compila e vincula os arquivos.

Figura 10.13 Avaliação de desempenho do protótipo Pond simulando o NFS.

10.6.3 Sistema de arquivos Ivy

Assim como o OceanStore, o Ivy [Muthitacharoen *et al.* 2002] é um sistema de arquivos de leitura/gravação que suporta vários leitores e escritores implementados sobre uma camada de sobreposição de roteamento e um sistema de armazenamento de dados distribuído endereçado com código de resumo. Ao contrário do OceanStore, o sistema de arquivos Ivy simula um servidor Sun NFS. O Ivy armazena o estado dos arquivos como *logs* das requisições de atualização de arquivos emitidos por clientes Ivy e reconstrói os arquivos percorrendo os *logs*, quando não é capaz de atender uma requisição de acesso a partir de sua cache local. Os registros do *log* são mantidos no serviço de armazenamento distribuído endereçado com código de resumo DHash [Dabek *et al.* 2001]. (Os *logs* foram usados pela primeira vez para gravar atualizações de arquivo no sistema operacional distribuído Sprite [Rosenblum e Ooosterhout 1992], conforme brevemente descrito na Seção 8.5, mas eles foram usados simplesmente para otimizar o desempenho de atualização do sistema de arquivos.)

O projeto do Ivy resolve vários problemas não solucionados anteriormente, que surgem da necessidade de hospedar arquivos em máquinas parcialmente confiáveis, ou não confiáveis, incluindo:

- A manutenção de metadados de arquivo consistentes (cf. o conteúdo de *i-node* nos sistemas de arquivos Unix/NFS) com atualizações de arquivo potencialmente concorrentes em diferentes nós. Não são usadas travas (*locks*), pois a falha de nós, ou de conectividade da rede, poderia causar um bloqueio indefinido.
- Confiança parcial entre os participantes e a vulnerabilidade a ataques das máquinas dos participantes. A recuperação das falhas de integridade causadas por tais ataques é baseada na noção de modos de visualização do sistema de arquivos. Um modo de visualização é uma representação do estado, construído a partir de *logs*, das atualizações feitas por um conjunto de participantes. Participantes podem ser removidos e um modo de visualização pode ser recalculado sem suas atualizações. Assim, um sistema de arquivos compartilhado é visto como o resultado da integração de todas as atualizações realizadas por um conjunto de participantes (selecionados dinamicamente).
- Operação continuada durante particionamentos da rede os quais podem resultar em atualizações conflitantes nos arquivos compartilhados. As atualizações conflitantes são resolvidas usando-se métodos relacionados àqueles utilizados no sistema de arquivos Coda (Seção 15.4.3).

O Ivy implementa em cada nó cliente uma API baseada no protocolo do servidor NFS (semelhante ao conjunto de operações listadas na Seção 8.3, Figura 8.9). Os nós clientes incluem um processo servidor Ivy que usa DHash para armazenar e acessar registros de *log* em nós de toda uma rede local ou remota, com base em chaves (GUIDs), calculadas como o código de resumo do conteúdo do registro (veja a Figura 10.14). O DHash implementa uma interface de programação como a que aparece

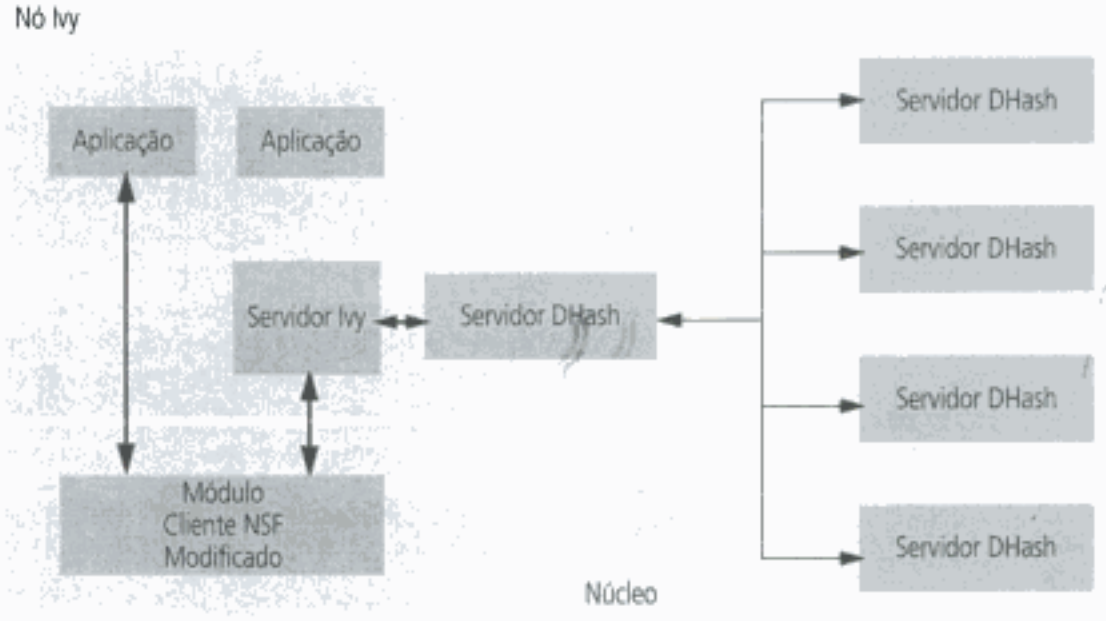


Figura 10.14 Arquitetura do sistema Ivy.

na Figura 10.4 e replica todas as entradas em vários nós para ter boa capacidade de recuperação e disponibilidade. Os autores do Ivy mencionam que, em princípio, o DHash poderia ser substituído por outro sistema de armazenamento distribuído endereçado com código de resumo, como o Pastry, o Tapestry ou o CAN.

Um sistema de armazenamento de arquivos Ivy consiste em um conjunto de *logs* de atualização, um por participante. Cada participante do Ivy anexa apenas em seu próprio *log*, mas pode ler todos os *logs* que compreendem o sistema de arquivos. As atualizações são armazenadas em *logs*, por participante separados, de modo que podem ser desfeitas no caso de brechas de segurança ou falhas de consistência.

Um *log* do Ivy é uma lista encadeada reversa, ordenada no tempo, de entradas de *log*. Cada entrada é um registro com indicação de tempo de uma requisição de cliente para alterar o conteúdo, ou metadados, de um arquivo ou diretório. O DHash usa o código de resumo SHA-1 de 160 bits de um registro como chave para colocar e recuperar o registro. Cada participante também mantém um bloco DHash mutável (chamado de cabeçalho de *log*), que aponta para o registro de *log* mais recente do participante. Os blocos mutáveis recebem um par de chaves públicas de criptografia de seus proprietários. O conteúdo do bloco é assinado com a chave privada e, portanto, pode ser autenticado com a chave pública correspondente. O Ivy usa vetores de versão (isto é, indicações de tempo vetoriais, veja a Seção 10.4) para impor uma ordem total nas entradas de *log*, ao ler vários *logs*.

O DHash armazena um registro de *log* usando como chave um código de resumo SHA-1 de seu conteúdo. Os registros de *log* são encadeados na ordem da indicação de tempo, usando a chave DHash como vínculo. O cabeçalho de *log* contém a chave da entrada de *log* mais recente. Para armazenar e recuperar cabeçalhos de *log*, um par de chaves públicas é calculado pelo proprietário do *log*. O valor da chave pública é usado como sua chave DHash, e a chave privada é usada pelo proprietário para assinar o cabeçalho de *log*. Qualquer participante que tenha a chave pública pode recuperar o cabeçalho de *log* e usá-lo para acessar todos os registros do *log*.

Supondo, por enquanto, um sistema de arquivos composto de um único *log*, o método de execução canônico de uma requisição para ler uma seqüência de bytes de um arquivo exige uma varredura seqüencial do *log* para localizar os registros que contêm atualizações da parte relevante do arquivo. Os *logs* têm comprimento ilimitado, mas a varredura termina quando é encontrado o primeiro registro (ou registros) que cobre a seqüência de bytes solicitada.

O algoritmo canônico para acessar um sistema de arquivos multiusuário, com múltiplos *logs*, envolve a comparação de indicações de tempo vetoriais nos registros de *log* e determinação da ordem das atualizações (pois um relógio global não pode ser pressuposto).

O tempo necessário para executar esse processo para uma operação simples, como a requisição de uma operação *read*, é potencialmente muito longo. Ele é reduzido para uma duração mais tolerável, e previsível, pelo uso de uma combinação de caches locais e *instantâneos*. Instantâneos são representações do sistema de arquivos calculadas e armazenadas de forma local em cada participante, como um subproduto de seu uso dos *logs*. Eles constituem uma representação temporária do sistema de arquivos, no sentido de que podem ser invalidados se um participante for ejetado do sistema.

A consistência de atualização é *fechar-para-abrir* (*close-to-open*), isto é, as atualizações realizadas em um arquivo por uma aplicação não são visíveis para outros processos até que o arquivo seja fechado. O uso de um modelo de consistência fechar-para-abrir permite que as operações *write* em um arquivo sejam salvas no nó cliente até que o arquivo seja fechado, então o conjunto inteiro de operações *write* é gravado como um único registro de *log* e um novo registro de cabeçalho de *log* é gerado e gravado (uma extensão do protocolo NFS permite que a ocorrência de uma operação *close* na aplicação seja notificada para o servidor Ivy).

Como existe um servidor Ivy separado em cada nó, e cada um armazena suas atualizações de forma autônoma em um *log* separado, sem coordenação com os outros servidores, a serialização das atualizações deve ser feita no momento em que os *logs* são lidos para construir o conteúdo dos arquivos. Os vetores de versão gravados nos registros de *log* podem ser usados para ordenar a maioria das atualizações, mas atualizações conflitantes são possíveis e elas devem ser resolvidas por métodos automáticos, ou manuais específicos da aplicação, como é feito no Coda (Seção 15.4.3).

A integridade dos dados é obtida por uma combinação dos mecanismos que já mencionamos: os registros de *log* são imutáveis e seus endereços são um código de resumo seguro de seu conteúdo; os cabeçalhos de *log* são conferidos pela verificação de uma assinatura de chave pública de seu conteúdo. Mas o modelo de confiança permite a possibilidade de que um participante mal-intencionado possa ter acesso a um sistema de arquivos. Por exemplo, ele poderia excluir arquivos que obteve direitos de acesso indevidamente. Quando isso é detectado, o participante mal-intencionado é ejetado do modo de visualização; seu *log* não é mais usado para calcular o conteúdo do sistema de arquivos, e os arquivos que já foram excluídos são novamente visíveis no novo modo de visualização.

Os autores do Ivy usaram um *benchmark* modificado do Andrew [Howard *et al.* 1988] para comparar o desempenho do Ivy com um servidor NFS padrão em ambientes de rede local e remota. Eles consideram (a) o Ivy usando servidores DHash locais comparados com um único servidor NFS local e (b) o Ivy usando servidores DHash localizados em vários sites remotos da Internet comparados com um único servidor NFS remoto. Eles também consideraram as características de desempenho como uma função dos números de participantes em um modo de visualização, o número de participantes gravando concorrentemente e o número de servidores DHash usados para armazenar os logs.

Eles descobriram que os tempos de execução do Ivy apresentava um fator duas vezes o tempo do NFS, para a maioria dos testes de *benchmark*, e não passavam um fator três vezes em nenhum dos casos restantes. Os tempos de execução para implantação em redes de longa distância ultrapassaram os do caso local por um fator de 10 ou mais, mas taxas semelhantes são obtidas para um servidor NFS remoto. Detalhes completos da avaliação de desempenho podem ser encontrados no artigo sobre o Ivy [Muthitacharoen *et al.* 2002]. Mas deve-se notar que o NFS não foi projetado para uso remoto; o Andrew File System e outros sistemas baseados em servidor desenvolvidos mais recentemente, como o xFS [Anderson *et al.* 1996], apresentam desempenho mais alto em implantação remota e poderiam ter constituído bases melhores para a comparação. A principal contribuição do Ivy está em sua nova estratégia para o gerenciamento da segurança e da integridade em um ambiente de confiança parcial – uma característica inevitável de sistemas distribuídos muito grandes, que abrangem muitas organizações e jurisdições.

10.7 Resumo

As arquiteturas *peer-to-peer* foram mostradas pela primeira vez, para suportar compartilhamento de dados em grande escala, com o uso do Napster e de seus derivados para compartilhamento de música digital. O fato de que grande parte de seu uso conflitava com as leis de direitos autorais não diminui sua importância técnica, embora tivessem inconvenientes que restringiam sua implantação em aplicações nas quais garantias da integridade dos dados e disponibilidade não eram importantes.

A pesquisa subseqüente resultou no desenvolvimento de plataformas de *middleware peer-to-peer* que enviam requisição para objetos de dados que estão localizados na Internet. Os objetos são endereçados usando-se GUIDs, os quais são nomes puros não contendo nenhuma informação de endereçamento IP. Os objetos são armazenados em nós de acordo com alguma função de mapeamento específica para cada sistema de *middleware*. O envio é realizado por sobreposição de roteamento no *middleware*, a qual mantém tabelas de roteamento e encaminha requisições ao longo de uma rota determinada pelo cálculo da distância, de acordo com a função de mapeamento escolhida.

As plataformas de *middleware* acrescentam garantias de integridade baseadas no uso de uma função de resumo segura para gerar os GUIDs, e garantias de disponibilidade baseadas na replicação de objetos em vários nós e nos algoritmos de roteamento tolerantes à falha.

As plataformas foram implantadas em várias aplicações piloto de larga escala, refinadas e avaliadas. Os resultados recentes da avaliação indicam que a tecnologia está pronta para implantação em aplicações que envolvem grandes números de usuários compartilhando muitos objetos de dados. As vantagens dos sistemas *peer-to-peer* incluem:

- sua capacidade de explorar recursos ociosos (armazenamento, processamento) nos computadores *host*;
- sua escalabilidade para suportar grandes números de clientes e *hosts* com excelente harmonização das cargas nos enlaces de rede e nos recursos computacionais do *host*;
- as propriedades de organização automática das plataformas de *middleware* resultam em custos de suporte amplamente independentes dos números de clientes e *hosts* implantados.

As deficiências e assuntos de pesquisa atual incluem:

- seu uso para o armazenamento de dados mutáveis é relativamente dispendioso, comparado a um serviço centralizado confiável;
- a principal promessa de que eles proporcionam anonimato de cliente e *host* ainda não resultou em fortes garantias de anonimato;

Exercícios

10.1 Os primeiros aplicativos de compartilhamento de arquivo, como o Napster, eram restritos em sua escalabilidade pela necessidade de manter um índice central de recursos e dos *hosts* que os continham. Quais outras soluções para o problema da indexação você pode identificar?

páginas 353–355, 359–360, Seção 15.4

10.2 O problema de manter índices de recursos disponíveis é dependente do aplicativo. Considere a conveniência de cada uma de suas resposta para o Exercício 10.1 para (a) compartilhamento de arquivos de música e mídia, (b) armazenamento a longo prazo de material arquivado, como conteúdo de periódico ou jornal, (c) armazenamento na rede de arquivos de leitura-gravação de propósito geral.

10.3 Quais são as principais garantias que os usuários esperam que os servidores convencionais (por exemplo, servidores web ou servidores de arquivos) ofereçam? *Seção 1.4.5*

10.4 As garantias oferecidas pelos servidores convencionais podem ser violadas como resultado de:

a) dano físico no *host*;

b) erros ou inconsistências dos administradores de sistema ou seus gerentes;

c) ataques bem-sucedidos contra a segurança do software de sistema;

d) erros de hardware ou software.

Cite dois exemplos de possíveis incidentes para cada tipo de violação. Quais deles poderiam ser descritos como uma brecha de confiança ou ação criminal? Elas seriam brechas de confiança se ocorressem em um computador pessoal que estivesse contribuindo com alguns recursos para um serviço *peer-to-peer*? Por que isso é relevante para os sistemas *peer-to-peer*? *Seção 7.1.1*

10.5 Normalmente, os sistemas *peer-to-peer* dependem de sistemas de computador *não confiáveis* e *voláteis* para a maioria de seus recursos. A confiança é um fenômeno social com conseqüências téc-

nicas. A volatilidade (isto é, disponibilidade imprevisível) também acontece freqüentemente devido às ações humanas. Elabore suas respostas para o Exercício 10.4, discutindo as maneiras possíveis pelas quais cada uma delas provavelmente vai diferir de acordo com os seguintes atributos dos computadores usados:

- posse
- localização geográfica
- conectividade da rede
- país ou jurisdição

O que isso sugere a respeito dos planos para a colocação de objetos de dados em um serviço de armazenamento *peer-to-peer*?

10.6 Avalie a disponibilidade e a confiabilidade dos computadores pessoais em seu ambiente. Você deve estimar:

Tempo de funcionamento: as horas por dia em que o computador está operando e conectado na Internet.

Consistência do software: o software é gerenciado por um técnico competente?

Segurança: o computador está totalmente protegido contra falsificação por parte de seus usuários ou outras pessoas?

Com base em sua avaliação, discuta a exeqüibilidade de executar um serviço de compartilhamento de dados no conjunto de computadores que você avaliou e descreva em linhas gerais os problemas que devem ser tratados em um serviço de compartilhamento de dados *peer-to-peer*. *páginas 355–357*

10.7 Explique como o uso do código de resumo seguro de um objeto, para identificar e direcionar mensagens para ele, garante que seja a prova de falsificação. Quais propriedades são exigidas da função de resumo? Como a integridade pode ser mantida, mesmo que uma proporção substancial dos nós pares seja destruída? *páginas 400, 423, Seção 7.4.3*

10.8 Freqüentemente é argumentado que os sistemas *peer-to-peer* podem oferecer anonimato para (a) clientes acessando recursos e (b) os *hosts* que dão acesso aos recursos. Discuta cada uma dessas proposições. Sugira uma maneira pela qual a resistência a ataques sobre o anonimato poderia ser melhorada. *página 353–355*

10.9 Os algoritmos de roteamento escolhem o próximo *hop* de acordo com uma estimativa da distância em algum espaço de endereçamento. Tanto o Pastry como o Tapestry usam espaços de endereço lineares e circulares, nos quais uma função baseada na diferença numérica aproximada entre GUIDs determina sua separação. O Kademlia usa a operação XOR dos GUIDs. Como isso ajuda na manutenção das tabelas de roteamento? A operação XOR fornece propriedades apropriadas para uma métrica de distância? *páginas 409, [Maymounkov e Mazieres 2002]*

10.10 Quando o serviço de uso de cache web *peer-to-peer* Squirrel foi avaliado pela simulação, 4,11 *hops* eram necessários, em média, para direcionar uma requisição de uma entrada de cache ao simular o tráfego de Redmond, enquanto apenas 1,8 foram exigidos para o tráfego de Cambridge. Explique esse fato e mostre que isso sustenta o desempenho teórico reivindicado para o Pastry. *páginas 360, 369–370*

Tempo e Estados Globais 11

Neste capítulo, apresentaremos alguns assuntos relacionados à questão de tempo em sistemas distribuídos. O tempo é um problema prático importante. Por exemplo, precisamos que os computadores espalhados pelo mundo mostrem a indicação de tempo das transações de comércio eletrônico consistentemente. O tempo também é uma construção teórica importante para se entender como as execuções distribuídas se desenrolam. Mas contar tempo é problemático nos sistemas distribuídos. Normalmente, cada computador tem seu próprio relógio físico e, havendo vários computadores, os relógios diferem entre si e é muito difícil sincronizá-los perfeitamente. Examinaremos algoritmos para a sincronização aproximada de relógios físicos e depois explicaremos os relógios lógicos, incluindo os relógios vetoriais, que são uma ferramenta para ordenar eventos sem saber precisamente quando eles ocorreram.

A ausência de um tempo físico global torna difícil descobrir o estado de nossos programas distribuídos quando eles são executados. Freqüentemente, precisamos saber qual é o estado do processo A, quando o processo B está em determinado ponto de sua execução, mas não podemos contar com os relógios físicos para saber o que é verdade ao mesmo tempo. A segunda metade do capítulo examinará algoritmos para determinar os estados globais de computações distribuídas, a despeito da falta de um tempo global.

11.1 Introdução

Este capítulo apresenta conceitos fundamentais e algoritmos relacionados para monitorar sistemas distribuídos à medida que sua execução se desenrola e para saber o momento exato em que eventos ocorrem.

O tempo é um problema importante e interessante nos sistemas distribuídos, por vários motivos. Primeiro, ele é uma quantidade que freqüentemente desejamos medir precisamente. Para saber em que hora do dia um evento específico ocorreu em um computador em particular é necessário sincronizar seu relógio com uma fonte de tempo externa confiável e aceita por todos. Por exemplo, uma transação de comércio eletrônico envolve eventos no computador do negociante e no computador de um banco. É importante, para propósitos de auditoria, que esses eventos tenham uma indicação de tempo precisa.

Segundo, foram desenvolvidos algoritmos que dependem da sincronização do relógio para vários problemas de distribuição [Liskov 1993]. Entre eles estão a manutenção da consistência dos dados distribuídos (o uso de indicações de tempo para dispor transações em série será discutido na Seção 13.6); a verificação da autenticidade de uma requisição enviada para um servidor (uma versão do protocolo de autenticação Kerberos, discutida no Capítulo 7, depende de relógios aproximadamente sincronizados); e a eliminação do processamento de atualizações replicadas (veja, por exemplo, Ladin *et al.* [1992]).

Einstein demonstrou, em sua Teoria da Relatividade Especial, as intrigantes conseqüências resultantes da observação de que a velocidade da luz é constante para todos os observadores, independentemente de sua velocidade relativa. A partir dessa suposição, ele provou, dentre outras coisas, que dois eventos considerados simultâneos em um ponto de referência não são necessariamente simultâneos de acordo com os observadores em outros pontos de referência que estão se movendo em relação a ele. Por exemplo, um observador na Terra e um observador viajando no espaço em uma nave espacial discordariam a respeito do intervalo de tempo entre os eventos, quanto mais suas velocidades relativas aumentassem.

Além disso, a ordem relativa de dois eventos pode ser até invertida para dois observadores diferentes. Mas isso não poderia acontecer se um evento tivesse causado a ocorrência do outro. Nesse caso, o efeito físico acompanha a causa física para todos os observadores, embora o tempo decorrido entre causa e efeito possa variar. Assim, a temporização de eventos físicos mostrou-se ser relativa ao observador e a noção de Newton de tempo físico absoluto foi desconsiderada. Não existe no universo nenhum relógio físico especial ao qual possamos recorrer quando queremos medir intervalos de tempo.

A noção de tempo físico também é problemática em um sistema distribuído. Isso não se deve aos efeitos da teoria da relatividade, que são desprezíveis, ou inexistentes, para computadores normais (a não ser que se considere computadores viajando em naves espaciais!). O problema é baseado em uma limitação semelhante em nossa capacidade de indicar o tempo de eventos, em diferentes nós, de modo suficientemente preciso, para saber a ordem em que quaisquer dois eventos ocorreram, ou se eles ocorreram simultaneamente. Não existe um tempo global absoluto ao qual possamos recorrer. Apesar disso, às vezes precisamos observar os sistemas distribuídos e determinar se certos estados ocorreram ao mesmo tempo. Por exemplo, nos sistemas orientados a objetos, precisamos estabelecer se as referências para um objeto em particular não existem mais – se o objeto se tornou lixo (no caso em que podemos liberar sua memória). Estabelecer isso exige observações dos estados de processos (para descobrir se eles contêm referências) e dos canais de comunicação entre processos (no caso em que mensagens contendo referências estejam em trânsito).

Na primeira metade deste capítulo, examinaremos métodos pelos quais os relógios de computador podem ser aproximadamente sincronizados, usando passagem de mensagens. Apresentaremos os relógios lógicos, incluindo os relógios vetoriais, os quais são usados para definir uma ordem de eventos, sem medir o tempo físico em que eles ocorreram.

Na segunda metade, descreveremos algoritmos cujo objetivo é capturar estados globais de sistemas distribuídos, à medida que eles executam.

11.2 Relógios, eventos e estados de processo

O Capítulo 2 apresentou um modelo introdutório de interação entre os processos dentro de um sistema distribuído. Vamos refinar esse modelo para nos ajudar a entender como caracterizamos a evolução do sistema quando ele executa e como indicamos o tempo dos eventos na execução de um sistema que interesse aos usuários. Começaremos considerando como ordenar e indicar o tempo dos eventos que ocorrem em um único processo.

Consideramos que um sistema distribuído consiste em um conjunto $\wp$ de N processos p_i, $i = 1, 2, \ldots N$. Cada processo é executado em um único processador e os processadores não compartilham memória (o Capítulo 18 considera o caso de processos que compartilham memória). Cada processo p_i em $\wp$ tem um estado s_i que, em geral, ele transforma ao ser executado. O estado do processo inclui os valores de todas as variáveis que estão dentro dele. Seu estado também pode incluir os valores de todos os objetos do seu ambiente de sistema operacional local que ele afeta como, entre outros, os arquivos. Presumimos que os processos não podem se comunicar de nenhuma maneira, exceto por meio do envio de mensagens pela rede. Assim, por exemplo, se existem processos que operam os braços de um robô, conectados em seus respectivos nós no sistema, então eles não podem se comunicar com um aperto de mãos!

Quando cada processo p_i executa, ele efetua uma série de ações, cada uma das quais sendo uma operação de envio ou recepção de mensagens, ou uma operação que transforma o estado de p_i – que altera um ou mais valores presentes em s_i. Na prática, podemos optar por usar uma descrição de alto nível das ações, de acordo com a aplicação. Por exemplo, se os processos em $\wp$ estão envolvidos em uma aplicação de comércio eletrônico, então as ações podem ser "o cliente enviou uma mensagem de pedido" ou "o servidor negociante gravou a transação no *log*".

Definimos um evento como a ocorrência de uma única ação que um processo realiza ao ser executado – uma ação de comunicação ou uma ação de transformação de estado. A seqüência de eventos dentro de um único processo p_i pode ser colocada em uma ordem total única, que denotaremos pela relação $\rightarrow_i$ entre os eventos. Isto é, $e \rightarrow_i e'$ se e somente se o evento e ocorre antes de e' em p_i. Essa ordem é bem definida, seja o processo *multi-threaded* ou não, pois supomos que o processo é executado em um único processador.

Agora, podemos definir o histórico do processo p_i como sendo a série de eventos que ocorrem dentro dele, ordenados conforme descrevemos pela relação $\rightarrow_i$:

$$histórico(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$$

Relógios ◊ Vimos como ordenar os eventos em um processo, mas não como indicar seu tempo – atribuir a eles uma data e uma hora do dia. Cada computador contém seu próprio relógio físico. Esses relógios são dispositivos eletrônicos que contam as oscilações que ocorrem em um cristal com uma freqüência de oscilação bem definida e que normalmente dividem essa contagem e armazenam o resultado em um registrador contador. Os dispositivos de relógio podem ser programados para gerar interrupções em intervalos regulares para que, por exemplo, possa ser implementada a fração de tempo de execução. Cada uma dessas interrupções é denominada de tiques do relógio (*clock ticks*). Contudo, não vamos nos preocupar com esse aspecto do funcionamento do relógio.

O sistema operacional lê o valor do relógio de hardware do nó $H_i(t)$, acerta sua escala e adiciona uma compensação para produzir um relógio de software $C_i(t) = \alpha H_i(t) + \beta$ que mede, aproximadamente, o tempo físico real t do processo p_i. Em outras palavras, quando o tempo real em um ponto de referência absoluto é t, $C_i(t)$ é a leitura no relógio de software. Por exemplo, $C_i(t)$ poderia ser o valor de 64 bits do número de nanossegundos decorridos no tempo t desde um tempo de referência conveniente. Em geral, o relógio não é completamente preciso e, assim, $C_i(t)$ irá diferir de t. Contudo, se C_i se comportar suficientemente bem (vamos examinar a noção de correção do relógio em breve), podemos usar seu valor para indicar o tempo de qualquer evento de p_i. Note que eventos sucessivos corresponderão a diferentes indicações de tempo somente se a *resolução do relógio* – o período entre as atualizações do valor do relógio – for menor do que o intervalo de tempo entre os eventos sucessivos. A taxa com que os eventos ocorrem depende de fatores como o comprimento do ciclo de instrução do processador.

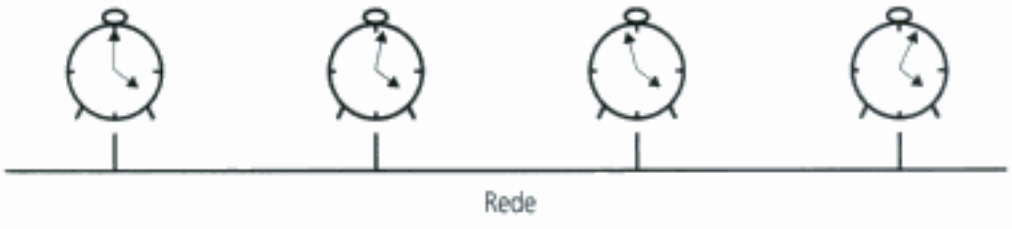


Figura 11.1 Distorção entre relógios de computador em um sistema distribuído.

Desvio de relógio e derivação de relógio ◊ Os relógios de computador, assim como os outros relógios, tendem a não estar em perfeito acordo (Figura 11.1). A diferença instantânea entre as leituras de quaisquer dois relógios é chamada de *distorção* (*skew*). Além disso, os relógios baseados em cristal usados nos computadores estão, assim como todos os outros relógios, sujeitos a *derivação de relógio* (*drift*), que significa que eles contam o tempo com diferentes velocidades e, portanto, divergem. Os osciladores de cristal estão sujeitos a variações físicas, e a conseqüência é que suas freqüências de oscilação diferem. Além disso, a freqüência de um mesmo relógio varia até com a temperatura. Existem projetos físicos que tentam compensar essa variação, mas eles não conseguem eliminá-la. A diferença no período de oscilação entre dois relógios pode ser extremamente pequena, mas a diferença acumulada em muitas oscilações leva a uma diferença perceptível nos contadores registrados pelos dois relógios, independentemente da precisão com que foram inicializados no mesmo valor. A *taxa de derivação* (*drift rate*) de um relógio é o deslocamento (diferença na leitura) entre o relógio local e um relógio de referência nominal perfeito, por unidade de tempo, medida pelo relógio de referência. Para relógios normais, baseados em um cristal de quartzo, isso dá cerca de 10^{-6} segundos de diferença em cada segundo – uma diferença de 1 segundo a cada 1.000.000 segundos, ou 11,6 dias. A taxa de derivação dos relógios de quartzo de alta precisão é de cerca de 10^{-7} ou 10^{-8}.

Tempo Universal Coordenado ◊ Os relógios de computador podem ser sincronizados com fontes externas de tempo altamente precisas. Os relógios físicos mais precisos usam osciladores atômicos, cuja taxa de derivação é de cerca de uma parte em 10^{13}. A saída desses relógios atômicos é usada como padrão para o tempo real decorrido, conhecido como *International Atomic Time* (tempo atômico internacional). Desde 1967, um segundo padrão foi definido como 9.192.631.770 períodos de transição entre duas camadas hiperfinas do estado fundamental do Césio-133 (Cs^{133}).

Os segundos, anos e outras unidades de tempo que usamos, têm suas raízes no tempo astronômico. Eles foram originalmente definidos em termos da rotação da Terra sobre seu eixo e sua translação em torno do Sol. Entretanto, o período de rotação da Terra sobre seu eixo está ficando gradualmente maior, principalmente devido ao atrito das marés. Os efeitos atmosféricos e as correntes de convecção dentro do núcleo da Terra também causam aumentos e diminuições no período de rotação da terra. Portanto, o tempo astronômico e o tempo atômico têm a tendência de entrar em descompasso.

O *Tempo Universal Coordenado* – abreviado como UTC (do seu equivalente em francês) – é um padrão internacional para contagem de tempo. Ele é baseado no tempo atômico, mas ocasionalmente é inserido – ou, mais raramente, excluído – o conhecido segundo bissexto, para manter a sincronização com o tempo astronômico. Os sinais UTC são sincronizados e transmitidos regularmente de estações de rádio terrestres e satélites cobrindo muitas partes do planeta. Por exemplo, nos EUA, a estação de rádio WWV transmite sinais de tempo em várias freqüências de ondas curtas. As fontes de satélite incluem o GPS (*Global Positioning System*).

Receptores estão disponíveis comercialmente. Comparados com o UTC perfeito, os sinais recebidos das estações terrestres têm uma precisão da ordem de 0,1–10 milissegundos, dependendo da estação usada. Os sinais recebidos do GPS têm precisão de cerca de 1 microssegundo. Computadores com receptores agregados podem sincronizar seus relógios com esses sinais de temporização. Os computadores também podem receber o tempo com uma precisão de poucos milissegundos, por meio de uma linha telefônica, de organizações como o National Institute for Standards and Technology dos EUA.

11.3 Sincronizando relógios físicos

Para saber em que hora do dia os eventos ocorrem nos processos de nosso sistema distribuído $\wp$ – por exemplo, para propósitos de contabilidade –, é necessário sincronizar os relógios C_i dos processos com uma fonte de tempo externa de referência. Esta é a *sincronização externa*. E se os relógios C_i são sincronizados com um grau de precisão conhecido, então podemos medir o intervalo entre dois eventos que ocorrem em diferentes computadores, recorrendo aos seus relógios locais – mesmo que eles não estejam necessariamente sincronizados com uma fonte de tempo externa. Esta é a *sincronização interna*. Definimos mais rigorosamente esses dois modos de sincronização sobre um intervalo de tempo real I, como segue:

> *Sincronização externa*: para um limite de sincronização $D > 0$ e para uma fonte S de tempo UTC, $|S(t) - C_i(t)| < D$, para $i = 1, 2, ...N$ e para todos os tempos reais t em I. Outra maneira de expressar isso é que os relógios C_i são *precisos* dentro do limite D.
>
> *Sincronização interna*: para um limite de sincronismo $D > 0$, $|C_i(t) - C_j(t)|$, para $i, j = 1, 2, ...N$ e para todos os tempos reais t em I. Outra maneira de expressar isso é que os relógios C_i concordam dentro do limite D.

Os relógios sincronizados internamente não são necessariamente sincronizados externamente, pois eles podem se desviar coletivamente de uma fonte de tempo externa, mesmo que concordem uns com os outros. Entretanto, a partir das definições conclui-se que, se o sistema $\wp$ for sincronizado externamente com um limite D, então o mesmo sistema será sincronizado internamente com um limite de $2D$.

Várias noções de *correção* para relógios foram sugeridas. É comum definir um relógio de hardware H como correto, se sua taxa de derivação cai dentro de um limite conhecido $\rho > 0$ (um valor derivado de outro fornecido pelo fabricante, como 10^{-6} segundos/segundo). Isso significa que o erro na medida do intervalo entre os tempos reais t e t' ($t' > t$) é limitado:

$$(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 - \rho)(t' - t)$$

Essa condição impossibilita saltos no valor de relógios de hardware (durante a operação normal). Às vezes, também exigimos que nossos relógios de software obedeçam à condição. Mas uma condição menos exigente de *monotonicidade* pode bastar. Monotonicidade é a condição de que um relógio C apenas sempre avance:

$$t' > t \Rightarrow C(t') > C(t)$$

Por exemplo, o recurso *make* do UNIX é uma ferramenta usada para compilar apenas os arquivos fontes que foram modificados desde a última vez em que foram compilados. As datas de modificação de cada par correspondente de arquivos fonte e objeto são comparadas para determinar essa condição. Se em um computador cujo relógio estivesse adiantado, fosse feita a compilação de um arquivo fonte e, em seguida, o relógio fosse atrasado para acertá-lo e, logo na seqüência, o arquivo fosse modificado, poderia parecer que o arquivo fonte foi modificado antes da compilação. Erroneamente, o recurso *make* não compilaria o arquivo fonte.

Podemos obter a monotonicidade, apesar do fato de um relógio se encontrar adiantando. Só precisamos mudar a taxa em que as atualizações são feitas para o tempo repassado às aplicações. Isso pode ser conseguido no software, sem mudar a taxa dos tiques do relógio de hardware físico – lembre-se de que $C_i(t) = \alpha H_i(t) + \beta$, onde estamos livres para escolher os valores de α e β.

Uma condição de correção mista, que às vezes é aplicada, é exigir que um relógio obedeça à condição de monotonicidade e que sua taxa de derivação seja limitada entre os pontos de sincronização, mas permitir que o valor do relógio salte para frente nesses pontos de sincronização.

Um relógio que não aplica as condições de correção é definido como *falho*. Diz-se que ocorre uma *falha de colapso* do relógio quando ele pára de tiquetaquear completamente; qualquer outra falha de relógio é uma *falha arbitrária*. Um exemplo de falha arbitrária é o de um relógio com o *bug* do ano 2000 (*bug* Y2K), que viola a condição de monotonicidade registrando a data, após 31 de dezembro de 1999, como 1º de janeiro de 1900, em vez de 2000; outro exemplo é um relógio com bateria fraca e cuja taxa de derivação repentinamente torna-se muito grande.

Note que, de acordo com as definições, os relógios não precisam ser precisos para estarem corretos. Como o objetivo pode ser a sincronização interna, em vez da externa, os critérios de correção se preocupam apenas com o funcionamento correto do mecanismo do relógio e não com sua configuração absoluta.

Vamos descrever agora os algoritmos de sincronização externa e interna.

11.3.1 Sincronização em um sistema síncrono

Começaremos considerando o caso mais simples possível: o da sincronização interna entre dois processos em um sistema síncrono distribuído. Em um sistema síncrono, são conhecidos os limites da taxa de derivação dos relógios, o atraso máximo de transmissão de mensagens e o tempo que leva para executar cada etapa de um processo (veja a Seção 2.3.1).

Um processo envia o tempo t de seu relógio local para o outro processo em uma mensagem m. Em princípio, o processo receptor poderia configurar seu relógio com o tempo $t + T_{trans}$, onde T_{trans} é o tempo que leva para transmissão de m entre eles. Então, os dois relógios concordariam (pois o objetivo é a sincronização interna; não importa se o relógio do processo remetente é preciso).

Infelizmente, T_{trans} está sujeito à variação e é desconhecido. Em geral, outros processos estão competindo pelos recursos com os processos a serem sincronizados, em seus respectivos nós, e outras mensagens competem com m pela rede. Contudo, sempre há um tempo de transmissão mínimo min que seria obtido se nenhum outro processo fosse executado e se não existisse nenhum outro tráfego na rede; min pode ser medido, ou estimado, de forma conservadora.

Por definição, em um sistema síncrono também existe um limite superior max para o tempo que leva para transmitir uma mensagem. Seja u a incerteza no tempo de transmissão da mensagem, tal que $u = (max - min)$. Se o receptor configurar seu relógio como $t + min$, então o desvio do relógio poderá ser no máximo u, pois, na verdade, a mensagem pode ter demorado o tempo max para chegar. Analogamente, se ele configurar seu relógio como $t + max$, o desvio poderá novamente ter o tamanho de u. Entretanto, se ele configurar seu relógio no ponto médio, $t + (max + min)/2$, então o desvio será no máximo $u/2$. Em geral, para um sistema síncrono, o limite ótimo que pode ser obtido para o desvio do relógio ao se sincronizar N relógios é $u(1 - 1/N)$ [Lundelius e Lynch 1984].

A maioria dos sistemas distribuídos encontrados na prática é assíncrona: os fatores que mais influenciam os atrasos de mensagem não estão ligados ao seu efeito e não existe nenhum limite superior max para os atrasos na transmissão de mensagens. Isso é particularmente verdadeiro no caso da Internet. Para um sistema assíncrono, podemos dizer somente que $T_{trans} = min + x$, onde $x \geq 0$. O valor de x não é conhecido em um caso em particular, embora uma distribuição de valores possa ser mensurável para uma determinada instalação.

11.3.2 Método de Cristian para sincronização de relógios

Cristian [1989] sugeriu o uso de um servidor de tempo, conectado a um dispositivo que recebe sinais de uma fonte UTC, para sincronizar computadores externamente. Ao receber uma requisição, o processo servidor S fornece o tempo, de acordo com seu relógio, como mostrado Figura 11.2. Cristian observou que, embora não haja um limite superior para os atrasos na transmissão de mensagens em um sistema assíncrono, freqüentemente os tempos de viagem de ida e volta para mensagens trocadas entre pares de

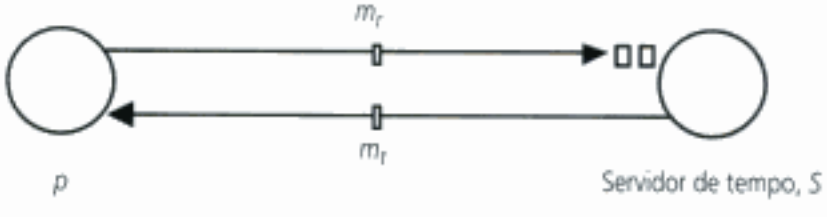


Figura 11.2 Sincronização de relógio usando um servidor de tempo.

processos são razoavelmente curtos – uma pequena fração de segundo. Ele descreve o algoritmo como *probabilístico*: o método só obtém sincronização se os tempos de viagem de ida e volta observados entre cliente e servidor forem suficientemente curtos, comparados com a precisão exigida.

Um processo p solicita o tempo em uma mensagem m_r e recebe o valor de tempo t em uma mensagem m_t (t é inserido em m_t no último ponto possível antes da transmissão do computador de S). O processo p registra o tempo de viagem de ida e volta total T_{viagem} que leva para enviar a requisição m_r e receber a resposta m_t. Ele pode medir esse tempo com precisão razoável, se sua taxa de derivação de relógio for pequena. Por exemplo, o tempo de viagem de ida e volta deve ser da ordem de 1–10 milissegundos em uma rede local, no qual um relógio com uma taxa de derivação de 10^{-6} segundos/segundo varia no máximo por 10^{-5} milissegundos.

Uma estimativa simples do tempo para o qual p deve configurar seu relógio é $t + T_{viagem}/2$, que presume que o tempo decorrido é dividido igualmente, antes e depois de S ter colocado t em m_t. Normalmente, essa é uma suposição razoavelmente precisa, a menos que as duas mensagens sejam transmitidas por redes diferentes. Se o valor do tempo de transmissão mínimo *min* for conhecido, ou puder ser estimado conservadoramente, então podemos determinar a precisão desse resultado, como segue.

O mais cedo que S poderia ter colocado o tempo em m_t seria *min* após p ter enviado m_r. O mais tarde que ele poderia ter feito isso seria *min* antes que m_t chegasse a p. Portanto, o tempo, de acordo com o relógio de S, para a chegada da mensagem de resposta está no intervalo $[t + min,\ t + T_{viagem} - min]$. A largura desse intervalo é $T_{viagem} - 2min$, de modo que a precisão é $\pm(T_{viagem}/2 - min)$.

A variabilidade pode ser tratada até certo ponto, fazendo-se várias requisições para S (espaçando-os para que congestionamentos transitórios na rede possam terminar) e pegando-se o valor mínimo de T_{viagem} para fazer a estimativa mais precisa. Quanto maior a precisão exigida, menor a probabilidade de obtê-la. Isso porque os resultados mais precisos são aqueles nos quais as duas mensagens são transmitidas em um tempo próximo a *min* – um evento improvável em uma rede ocupada.

Discussão sobre o algoritmo de Cristian ◊ Conforme foi descrito, o método de Cristian sofre do problema associado a todos os serviços implementados por um único servidor: o de que o único servidor de tempo pode falhar e, assim, tornar a sincronização temporariamente impossível. Por isso, Cristian sugeriu que o tempo fosse fornecido por um conjunto de servidores de tempo sincronizados, cada um com um receptor de sinais de tempo UTC. Por exemplo, um cliente poderia enviar sua requisição em *multicast* para todos os servidores e utilizar apenas a primeira resposta obtida.

Note que um servidor de tempo defeituoso que respondesse com valores de tempo espúrios, ou um servidor de tempo impostor que respondesse com tempos deliberadamente incorretos, poderia prejudicar um sistema de computação. Esses problemas estão fora dos objetivos do trabalho descrito por Cristian [1989], o qual presume que as fontes de sinais de tempo externas têm verificação automática. Cristian e Fetzer [1994] descrevem uma família de protocolos probabilísticos para sincronização interna do relógio, cada um dos quais tolerando certas falhas. Srikanth e Toueg [1987] descreveram pela primeira vez um algoritmo que é excelente com relação à precisão dos relógios sincronizados, embora tolere apenas algumas falhas. Dolev *et al.* [1986] mostraram que, se f é o número de relógios defeituosos de um total de N, então devemos ter $N > 3f$, se os outros relógios, corretos, ainda forem capazes de entrar em acordo. O problema do tratamento com relógios defeituosos é parcialmente resolvido pelo algoritmo Berkeley, que será descrito a seguir. O problema da interferência mal-intencionada na sincronização de relógios pode ser resolvido com técnicas de autenticação.

11.3.3 O algoritmo Berkeley

Gusella e Zatti [1989] descrevem um algoritmo de sincronização interna, que desenvolveram para conjuntos de computadores executando o UNIX Berkeley. Nele, um computador coordenador é escolhido para atuar como *mestre*. Ao contrário do protocolo de Cristian, esse computador faz periodicamente uma consulta seqüencial nos outros computadores cujos relógios devem ser sincronizados, chamados de *escravos*. Os escravos enviam de volta seus valores de relógio. O mestre faz uma estimativa dos tempos locais desses relógios, observando os tempos de viagem de ida e volta (semelhante à técnica de Cristian) e faz a média dos valores obtidos (incluindo a leitura de seu próprio relógio). O balanço das probabilidades é que essa média cancela as tendências dos relógios individuais de estarem

adiantados ou atrasados. A precisão do protocolo depende de um tempo máximo nominal de viagem de ida e volta entre o mestre e os escravos. O mestre elimina todas as leituras ocasionais associadas aos tempos maiores do que esse máximo.

Em vez de enviar para os outros computadores o tempo corrente atualizado – o que introduziria mais incerteza, devido ao tempo de transmissão da mensagem –, o mestre envia o valor pelo qual o relógio de cada escravo individual exige ajuste. Esse valor pode ser positivo ou negativo.

O algoritmo elimina as leituras de relógios defeituosos. Tais relógios teriam um efeito adverso significativo, caso fosse tirada uma média normal. O mestre tira uma *média tolerante à falha*, ou seja, é escolhido um subconjunto dos relógios que não diferem uns dos outros por mais do que um valor especificado e é feita a média das leituras apenas desses relógios.

Gusella e Zatti descrevem uma experiência envolvendo 15 computadores cujos relógios foram sincronizados dentro de cerca de 20-25 milissegundos, usando seu protocolo. A taxa de derivação dos relógios locais foi medida como menos de 2×10^{-5} e o tempo de viagem de ida e volta máximo obtido foi de 10 milissegundos.

Se o mestre falhar, então outro poderá ser escolhido para assumir o controle e funcionar exatamente como seu predecessor. A Seção 12.3 discutirá alguns algoritmos de eleição de propósito geral. Note que neles não há garantias da escolha de um novo mestre dentro de um limite de tempo – e, portanto, a diferença entre dois relógios poderia ser ilimitada, caso eles fossem usados.

11.3.4 O Network Time Protocol

O método de Cristian e o algoritmo Berkeley se destinam principalmente ao uso dentro de intranets. O NTP (*Network Time Protocol*) [Mills 1995] define uma arquitetura para um serviço de tempo e um protocolo para distribuir informações de tempo pela Internet.

Os principais objetivos de projeto e características do NTP são os seguintes.

Fornecer um serviço que permita aos clientes na Internet serem sincronizados precisamente com o UTC, a despeito dos atrasos de mensagem, grandes e variáveis, encontrados na comunicação via Internet. O NTP emprega técnicas estatísticas para a filtragem de dados de temporização e faz discriminação entre a qualidade dos dados de tempo de diferentes servidores.

Fornecer um serviço confiável que possa sobreviver a longas perdas de conectividade. Existem servidores redundantes e caminhos redundantes entre os servidores. Os servidores podem ser reconfigurados para que continuem a fornecer o serviço, caso um deles se torne inatingível.

Permitir que os clientes sejam sincronizados de forma suficientemente freqüente para compensar as taxas de derivação encontradas na maioria dos computadores. O serviço é projetado para suportar um grande número de clientes e servidores.

Fornecer proteção contra interferência no serviço de tempo, seja mal-intencionada ou acidental. O serviço de tempo usa técnicas de autenticação para verificar se os dados de temporização são originários das fontes confiáveis conhecidas. Ele também valida os endereços de retorno das mensagens que recebe.

O serviço NTP é fornecido por uma rede de servidores localizados na Internet. Os *servidores primários* são conectados diretamente a uma fonte de tempo, como um relógio de rádio recebendo UTC; os *servidores secundários* são sincronizados com os servidores principais. Os servidores são conectados em uma hierarquia lógica chamada *sub-rede de sincronização* (veja a Figura 11.3), cujos níveis são chamados de *strata*. Os servidores primários ocupam o *stratum* 1: eles estão na raiz. Os servidores do *stratum* 2 são secundários, sincronizados diretamente com os servidores primários; os servidores do *stratum* 3 são sincronizados com os servidores do *stratum* 2 e assim por diante. Os servidores de nível mais baixo (folha) são executados nas estações de trabalho dos usuários.

Os relógios pertencentes aos servidores com números de *stratum* mais altos estão sujeitos a serem menos precisos do que aqueles com números de *stratum* baixos, pois erros são introduzidos em cada nível de sincronização. O NTP também leva em conta, na avaliação da qualidade dos dados de temporização mantidos por um servidor em particular, os atrasos da viagem de ida e volta total da mensagem até a raiz.

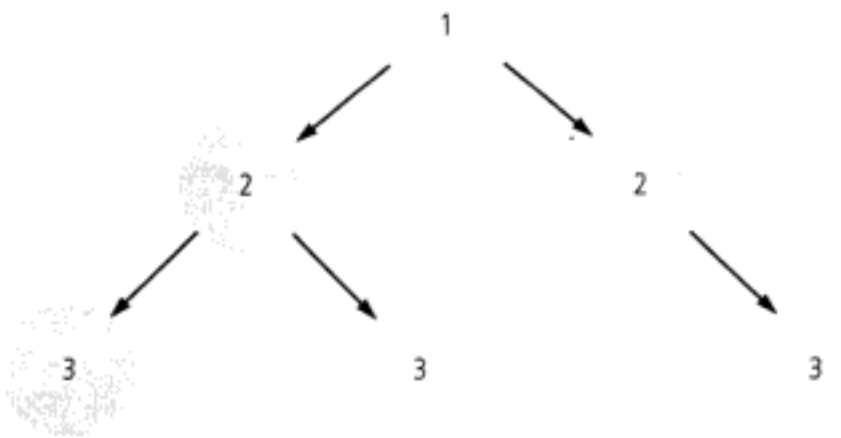


Figura 11.3 Um exemplo de sub-rede de sincronização em uma implementação de NTP.
Nota: as setas indicam controle de sincronização, os número fornecem o *stratum*.

A sub-rede de sincronização pode ser reconfigurada quando servidores se tornam inatingíveis ou quando ocorrem falhas. Se, por exemplo, a fonte de UTC de um servidor primário falha, então ele pode se tornar um servidor secundário do *stratum* 2. Se a fonte de sincronismo normal de um servidor secundário falha, ou se torna inatingível, então ele pode ser sincronizado com outro servidor.

Os servidores NTP são sincronizados entre si de três maneiras: *multicast*, chamada de procedimento e modo simétrico. O *modo multicast* se destina a ser usado em uma rede local de alta velocidade. Periodicamente, um ou mais servidores de tempo, enviam em *multicast* a informação de tempo para servidores que estão em execução em outros computadores conectados na rede local, os quais configuram seus relógios pressupondo um pequeno atraso. Esse modo pode obter apenas uma precisão relativamente baixa, mas que são consideradas suficientes para muitos propósitos.

O *modo de chamada de procedimento* é semelhante ao funcionamento do algoritmo de Cristian, descrito anteriormente. Nesse modo, um servidor aceita requisições de outros computadores, os quais ele processa respondendo com sua indicação de tempo (leitura corrente do relógio). Esse modo é conveniente onde é exigida uma precisão melhor do que a que pode ser obtida com o *multicast* – ou onde *multicast* não é suportado no hardware. Por exemplo, servidores de arquivos na mesma rede local, ou em uma rede local vizinha, que precisem manter informações precisas do tempo de acesso a arquivos, poderiam entrar em contato com um servidor local no modo de chamada de procedimento.

Finalmente, o *modo simétrico* que se destina a ser usado pelos servidores que fornecem informações de tempo em redes locais e pelos níveis hierarquicamente mais altos (número de *stratum* mais baixos) da sub-rede de sincronização, onde uma maior precisão precisa ser obtida. Dois servidores operando no modo simétrico trocam mensagens com informações de temporização. Esses dados são armazenados como parte de uma associação entre os servidores que são mantidos para melhorar a precisão de sua sincronização com o passar do tempo.

Em todos os modos, as mensagens são enviadas de maneira não confiável, usando o protocolo de transporte UDP. No modo de chamada de procedimento e no modo simétrico, os processos trocam pares de mensagens. Cada mensagem carrega indicações de tempo de eventos de mensagem recentes: os tempos locais de quando a mensagem NTP anterior entre o par foi enviada e recebida e o tempo local de quando a mensagem corrente foi transmitida. O destinatário da mensagem NTP anota o tempo local ao receber a mensagem. Os quatro tempos, T_{i-3}, T_{i-2}, T_{i-1},e T_i aparecem na Figura 11.4 para as mensagens m e m' enviadas entre os servidores A e B. Note que, no modo simétrico, ao contrário do algoritmo de Cristian descrito anteriormente, pode haver um atraso não desprezível entre a chegada de uma mensagem e o envio da próxima. Além disso, mensagens podem ser perdidas, mas as três indicações de tempo transportadas em cada mensagem são válidas.

Para cada par de mensagens enviadas entre dois servidores, o NTP calcula uma compensação o_i, que é uma estimativa da compensação real entre os dois relógios, e um atraso d_i, que é o tempo

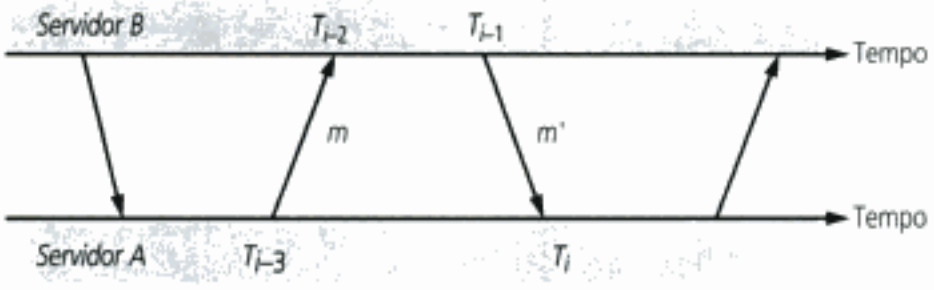


Figura 11.4 Mensagens trocadas entre dois pares NTP.

de transmissão total das duas mensagens. Se o valor real da compensação do relógio em B, relativa à de A, for o e, se os tempos de transmissão reais de m e m' forem t e t' respectivamente, então temos:

$$T_{i-2} = T_{i-3} + t + o \text{ e } T_i = T_{i-1} + t' - o$$

Isso leva a:

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

Além disso

$$o = o_i + (t' - t)/2, \text{ onde } o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$$

Usando o fato de que $t, t' \geq 0$, pode ser mostrado que $o_i - d_i/2 \leq o \leq o_i + d_i/2$. Assim, o_i é uma estimativa da compensação e d_i é uma medida da precisão dessa estimativa.

Os servidores NTP aplicam um algoritmo de filtragem de dados em sucessivos pares $<o_i, d_i>$, para fazer uma estimativa da compensação o e calcular a qualidade dessa estimativa com base em uma quantidade estatística chamada *filtro de dispersão*. Uma dispersão relativamente alta representa dados relativamente não confiáveis. Os oito pares $<o_i, d_i>$ mais recentes são mantidos. Assim como acontece com o algoritmo de Cristian, o valor de o_j que corresponde ao valor mínimo d_j é escolhido para fazer a estimativa de o.

Entretanto, o valor da compensação derivado da comunicação com uma única fonte não é necessariamente usado sozinho para controlar o relógio local. Em geral, um servidor NTP se envolve nas trocas de mensagem com vários de seus pares. Além da filtragem de dados aplicada nas trocas com cada par, o NTP aplica um algoritmo de seleção de par. Esse algoritmo examina os valores obtidos das trocas com cada um dos vários pares, procurando valores relativamente não confiáveis. A saída desse algoritmo pode fazer um servidor mudar o par que normalmente utiliza para obter sincronismo.

Os pares com números de *stratum* mais baixos são mais favorecidos do que aqueles que possuem números de *stratum* mais altos, pois estão mais próximos das fontes de tempo principais. Além disso, aqueles com a *dispersão de sincronização* mais baixa são relativamente favorecidos. Essa é a soma das dispersões de filtro medidas entre o servidor e a raiz da sub-rede de sincronização. (Os pares trocam dispersões de sincronização nas mensagens, permitindo que esse total seja calculado.)

O NTP emprega um modelo de laço com bloqueio de fase [Mills 1995], o qual modifica a freqüência de atualização do relógio local de acordo com observações de sua taxa de derivação. Para dar um exemplo simples, se é descoberto um relógio que sempre avança no tempo na taxa de, digamos, quatro segundos por hora, então sua freqüência pode ser reduzida ligeiramente (no software ou no hardware) para compensar isso. Assim, a derivação do relógio entre intervalos de sincronização é reduzida.

Mills cita precisão de sincronização na ordem de dezenas de milissegundos na Internet e de um milissegundo em redes locais.

11.4 Tempo lógico e relógios lógicos

Do ponto de vista de um único processo, os eventos são ordenados exclusivamente pelos tempos dados pelo relógio local. Entretanto, conforme Lamport [1978] mostrou, como não podemos sincronizar perfeitamente os relógios em um sistema distribuído, em geral não podemos usar o tempo físico para descobrir a ordem de qualquer par de eventos arbitrários que ocorram dentro dele.

Em geral, podemos usar um esquema semelhante à causalidade física, mas aplicada aos sistemas distribuídos, para ordenar alguns dos eventos que ocorrem em diferentes processos. Essa ordenação é baseada em dois pontos simples e intuitivamente óbvios:

- Se dois eventos ocorreram no mesmo processo p_i ($i = 1, 2, ..., N$), então eles ocorreram na ordem em que p_i os observou – essa é a ordem $\rightarrow_i$ que definimos anteriormente.
- Quando uma mensagem é enviada entre processos, o evento de envio da mensagem ocorreu antes do evento de recepção da mensagem.

Lamport chamou a ordenação parcial obtida pela generalização desses dois relacionamentos de relação *antes do acontecido* (*happened-before*). Às vezes, ela também é conhecida como relação de *ordenação causal* (*causal ordering*) ou *ordenação causal potencial* (*potential causal ordering*).

Podemos definir a relação antes do acontecido (AA), denotada por $\rightarrow$, como segue:

AA1: Se $\exists$ processa p_i: $e \rightarrow_i e'$, então $e \rightarrow e'$.

AA2: Para qualquer mensagem m, $send(m) \rightarrow receive(m)$
– onde $send(m)$ é o evento de envio da mensagem e $receive(m)$ é o evento de sua recepção.

AA3: Se e, e' e e'' são eventos tais que $e \rightarrow e'$ e $e' \rightarrow e''$, então $e \rightarrow e''$.

Assim, se e e e' são eventos e se $e \rightarrow e'$, então, podemos encontrar uma série de eventos e_1, e_2,..., e_n que ocorrem em um ou mais processos, tal que $e = e_1$ e $e' = e_n$ e, para $i = 1, 2, ..., N - 1$, AA1 ou AA2 se aplica entre e_i e e_{i+1}. Isto é, ou eles ocorrem sucessivamente no mesmo processo ou existe uma mensagem m tal que $e_i = send(m)$ e $e_{i+1} = receive(m)$. A seqüência de eventos e_1, e_2,..., e_n não precisa ser única.

A relação $\rightarrow$ está ilustrada na Figura 11.5 para o caso de três processos p_1, p_2 e p_3. Pode ser visto que $a \rightarrow b$, pois os eventos ocorrem nessa ordem no processo p_1 ($a \rightarrow_1 b$) e, semelhantemente, $c \rightarrow d$. Além disso, $b \rightarrow c$, pois esses eventos são o envio e a recepção da mensagem m_1 e, semelhantemente, $d \rightarrow f$. Combinando essas relações, também podemos dizer que, por exemplo, $a \rightarrow f$.

Na Figura 11.5, também pode ser visto que nem todos os eventos estão relacionados pela relação $\rightarrow$. Por exemplo, $a \not\rightarrow e$ e $e \not\rightarrow a$, pois eles ocorrem em diferentes processos e não existe nenhum encadeamento de mensagens intervenientes entre eles. Dizemos que eventos como a e e, que não são ordenados por $\rightarrow$, são concorrentes, e escrevemos isso como $a \parallel e$.

A relação $\rightarrow$ captura um fluxo de dados interveniente entre dois eventos. Note, entretanto, que, em princípio, os dados podem fluir de maneiras diferentes da passagem de mensagens. Por exemplo, se

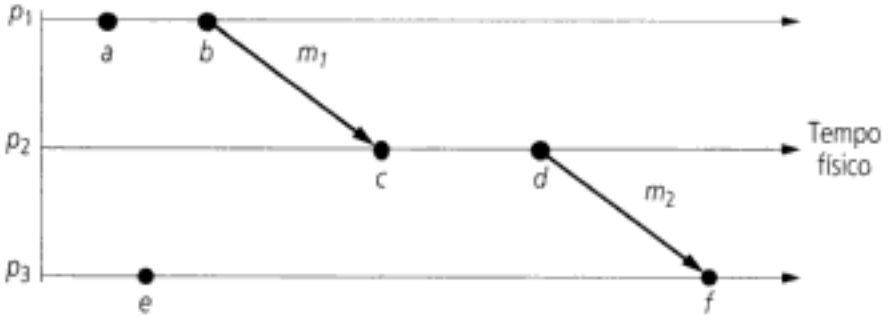


Figura 11.5 Eventos ocorrendo em três processos.

Smith insere um comando em seu processo para enviar uma mensagem e então telefona para Jones, que faz seu processo enviar outra mensagem, então o envio da primeira mensagem claramente *aconteceu antes* da segunda. Infelizmente, como para essa coordenação, nenhuma mensagem de rede foi enviada entre os processos emitentes, não podemos modelar esse tipo de relacionamento em nosso sistema.

Outro ponto a notar é que, se a relação antes do acontecido vale entre dois eventos, então o primeiro poderia ou não ter causado o segundo. Por exemplo, se um servidor recebe uma mensagem de requisição e, subseqüentemente, envia uma resposta, então claramente a transmissão da resposta é causada pela transmissão da requisição. Entretanto, a relação $\rightarrow$ captura apenas a causalidade em potencial e dois eventos podem estar relacionados por $\rightarrow$, mesmo que não exista nenhuma conexão real entre eles. Um processo poderia, por exemplo, receber uma mensagem e, subseqüentemente, enviar outra mensagem, mas que ele envia a cada cinco minutos, sem nenhuma relação específica com a primeira mensagem. Nenhuma causalidade real foi envolvida, mas a relação $\rightarrow$ ordenaria esses eventos.

Relógios lógicos ◊ Lamport inventou um mecanismo simples por meio do qual a ordenação antes do acontecido pode ser capturada numericamente, chamado de *relógio lógico*. O relógio lógico de Lamport é um contador de software que aumenta a contagem monotonicamente, cujo valor não precisa ter nenhum relacionamento em particular com qualquer relógio físico. Cada processo p_i mantém seu próprio relógio lógico, L_i, que utiliza para aplicar as conhecidas *indicações de tempo de Lamport* nos eventos. Denotamos a indicação de tempo do evento e em p_i como $L_i(e)$ e com $L(e)$, a indicação de tempo do evento e no processo em que ela ocorreu.

Para capturar a relação antes do acontecido $\rightarrow$, os processos atualizam seus relógios lógicos e transmitem os valores de seus relógios lógicos em mensagens, como segue:

RL1: L_i é incrementado antes da ocorrência de um evento no processo p_i:
$L_i := L_i + 1$

RL2: (a) Quando um processo p_i envia uma mensagem m, m leva 'de carona' (*piggybacking*) o valor $t = L_i$.

(b) Na recepção (m, t), um processo p_j calcula L_j: $max(L_j, t)$ e, então, aplica RL1 antes de indicar o tempo do evento *receive*(m).

Embora incrementemos os relógios por 1, poderíamos ter escolhido qualquer valor positivo. Isso pode ser facilmente mostrado, por indução, em qualquer seqüência de eventos relacionando dois eventos e e e', que $e \rightarrow e' \Rightarrow L(e) < L(e')$.

Note que o inverso não é verdadeiro. Se $L(e) < L(e')$, então, não podemos inferir que $e \rightarrow e'$. Na Figura 11.6, ilustramos o uso de relógios lógicos para o exemplo dado na Figura 11.5. Cada um dos processos p_1, p_2 e p_3 tem seu relógio lógico inicializado em 0. Os valores de relógio dados são aqueles imediatamente após o evento ao qual eles são adjacentes. Note que, por exemplo, $L(b) > L(e)$, mas $b \parallel e$.

Relógios lógicos totalmente ordenados ◊ Alguns pares de eventos distintos, gerados por diferentes processos, têm indicações de tempo de Lamport numericamente idênticas. Entretanto, podemos criar uma ordem total nos eventos – isto é, uma ordem para a qual todos os pares de eventos distintos

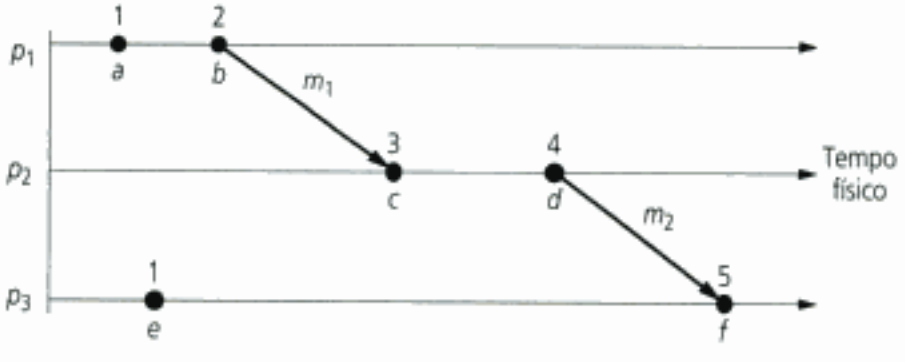


Figura 11.6 Indicações de tempo de Lamport para os eventos mostrados na Figura 11.5.

são ordenados levando em conta os identificadores dos processos em que os eventos ocorrem. Se e é um evento ocorrendo em p_i com indicação de tempo local T_i e e' é um evento ocorrendo em p_j com indicação de tempo local T_j, definimos as indicações de tempo globais lógicas para esses eventos como (T_i, i) e (T_j, j) respectivamente. E definimos $(T_i, i) < (T_j, j)$, se e somente se $T_i < T_j$ ou $T_i = T_j$ e $i < j$. Essa ordenação não tem nenhum significado físico geral (pois os identificadores de processo são arbitrários), mas às vezes ela é útil. Lamport a utilizou, por exemplo, para ordenar a entrada de processos em uma seção crítica.

Relógios vetoriais ◊ Mattern [1989] e Fidge [1991] desenvolveram relógios vetoriais para superar a deficiência dos relógios de Lamport: o fato de que, a partir de $L(e) < L(e')$, não podemos concluir que $e \rightarrow e'$. Um relógio vetorial para um sistema de N processos é um vetor de N inteiros. Cada processo mantém seu próprio relógio vetorial V_i, o qual utiliza para indicar o tempo dos eventos locais. Assim como nas indicações de tempo de Lamport, os processos levam 'de carona' as indicações de tempo vetoriais nas mensagens que trocam entre si e existem regras simples para atualizar os relógios, como segue:

RV1: Inicialmente, $V_i[j] = 0$, para $i, j = 1, 2..., N$.

RV2: Imediatamente antes de p_i indicar o tempo de um evento, ele configura $V_i[i] := V_i[i] + 1$.

RV3: p_i inclui o valor $t = V_i$ em cada mensagem que envia.

RV4: Quando p_i recebe uma indicação de tempo t em uma mensagem, ele configura $V_i[j] := max(V_i[j], t[j])$, para $j = 1, 2..., N$. Considerar o máximo de duas componentes de indicações de tempo vetoriais dessa maneira, é conhecido como operação de *merge* (integração).

Para um relógio vetorial V_i, $V_i[i]$ é o número de eventos em que p_i indicou o tempo e $Vi[j]$ $(j \neq i)$ é o número de eventos ocorridos em p_j nos quais p_i potencialmente foi afetado. (O processo p_j pode ter indicado o tempo de mais eventos nesse ponto, mas ainda nenhuma informação sobre eles fluiu para p_i nas mensagens.)

Podemos comparar indicações de tempo vetoriais, como segue:

$V = V'$ *sse* $V[j] = V'[j]$ para $j = 1, 2 ..., N$

$V \leq V'$ *sse* $V[j] \leq V'[j]$ para $j = 1, 2 ..., N$

$V < V'$ *sse* $V \leq V' \wedge V \neq V'$

Seja $V(e)$ a indicação de tempo vetorial aplicada pelo processo em que e ocorre. É simples mostrar, por indução, no comprimento de qualquer seqüência de eventos relacionados a dois eventos e e e', que $e \rightarrow e' \Rightarrow V(e) < V(e')$. O Exercício 10.13 leva o leitor a mostrar o inverso: se $V(e) < V(e')$, então, $e \rightarrow e'$.

A Figura 11.7 mostra as indicações de tempo vetoriais dos eventos da Figura 11.5. Pode ser visto, por exemplo, que $V(a) < V(f)$, que reflete o fato de que $a \rightarrow f$. Analogamente, podemos identificar quando dois eventos são concorrentes comparando suas indicações de tempo. Por exemplo, que $c \parallel e$ pode ser visto dos fatos de que nem $V(c) \leq V(e)$ nem $V(e) \leq V(c)$.

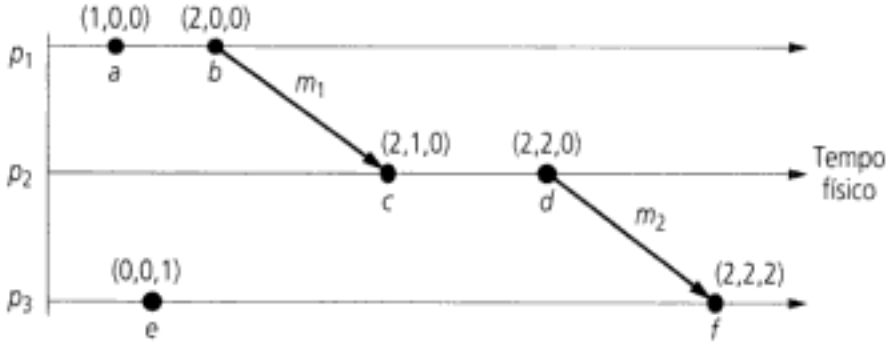


Figura 11.7 Indicações de tempo vetoriais para os eventos mostrados na Figura 11.5.

As indicações de tempo vetoriais têm a desvantagem, comparadas com as indicações de tempo de Lamport, de ocupar espaço de armazenamento e carga útil de mensagem proporcional a N, o número de processos. Charron-Bost [1991] mostrou que se somos capazes de identificar se dois eventos são concorrentes ou não, inspecionando suas indicações de tempo, então a dimensão N é inevitável. Entretanto, existem técnicas para armazenar e transmitir volumes de dados menores, à custa do processamento necessário para reconstruir as versões completas dos vetores. Raynal e Singhal [1996] apresentam uma narrativa sobre algumas dessas técnicas. Eles também descrevem a noção de *relógios de matriz*, na qual os processos mantêm estimativas dos tempos vetoriais de outros processos, assim como de seus próprios tempos.

11.5 Estados globais

Nesta e na próxima seção, examinaremos o problema de descobrir se uma propriedade em particular é verdadeira em um sistema distribuído, quando ele a executa. Começaremos dando os exemplos da coleta de lixo distribuída, da detecção de impasses, da detecção de término e da depuração.

Coleta de lixo distribuída: um objeto é considerado lixo se não existem mais referências a ele em nenhuma parte do sistema distribuído. A memória ocupada por esse objeto pode ser reivindicada, quando ele for reconhecido como lixo. Para verificar se um objeto é lixo, devemos ver se não existem referências a ele em nenhuma parte do sistema. Na Figura 11.8a, o processo p_1 tem dois objetos, ambos com referências – um deles é referenciado localmente pelo próprio p_1 e o outro é referenciado remotamente por p_2. O processo p_2 tem um objeto que é lixo, sem referências a ele em nenhuma parte do sistema. Ele também tem um objeto para o qual nem p_1, nem p_2, tem uma referência, mas existe uma referência a ele em uma mensagem que está em trânsito entre os processos. Isso mostra que, quando consideramos as propriedades de um sistema, devemos incluir o estado dos canais de comunicação, assim como o estado dos processos.

Detecção de impasses distribuída: um impasse (*deadlock*) distribuído ocorre quando cada processo de uma coleção de processos espera que outro envie uma mensagem para ele, e há um ciclo no grafo desse relacionamento "espera por". A Figura 11.8b mostra que cada um dos processos p_1 e p_2 espera por uma mensagem do outro; portanto, esse sistema nunca fará progresso.

Detecção de término distribuída: o problema aqui é detectar se um algoritmo distribuído terminou. Detectar o término é um problema que parece enganosamente fácil de resolver: à primeira vista, parece que é necessário apenas testar se cada processo parou. Para ver que isso não é assim, considere um algoritmo distribuído executado por dois processos p_1 e p_2, cada um dos quais podendo solicitar valores um do outro. Instantaneamente, podemos verificar que um processo ou está ativo ou está passivo – um processo passivo não está envolvido em nenhuma atividade propriamente dita, mas está preparado para responder com um valor solicitado pelo outro. Suponha que tenhamos descoberto que p_1 e p_2 são passivos (Figura 11.8c). Para ver que não podemos concluir que o algoritmo terminou, considere o seguinte cenário: quando testamos o estado de p_1 (passivo, no caso), havia uma mensagem a caminho para ele emitida por p_2, o qual se tornou passivo imediatamente após enviá-la. Ao receber a mensagem, p_1 se tornou novamente ativo – após descobrirmos que ele era passivo. O algoritmo não tinha terminado.

Os fenômenos do término e do impasse são semelhantes sob alguns aspectos, mas trata-se de problemas diferentes. Primeiro, um impasse pode afetar apenas um subconjunto dos processos em um sistema, enquanto o término considera todos os processos. Segundo, o estado ativo/passivo de um processo não é o mesmo que esperar em um ciclo de impasse: um processo em impasse está tentando executar mais uma ação, pela qual outro processo espera; um processo passivo não está envolvido em nenhuma atividade.

Depuração distribuída: os sistemas distribuídos são complicados de depurar [Bonnaire *et al.* 1995] e é preciso cuidado ao se estabelecer o que ocorreu durante a execução. Por exemplo, Smith escreveu uma aplicação na qual cada processo p_i contém uma variável x_i ($i = 1, 2 \ldots, N$). As variáveis mudam à medida que o programa executa, mas são obrigadas a estar sempre dentro de um valor δ

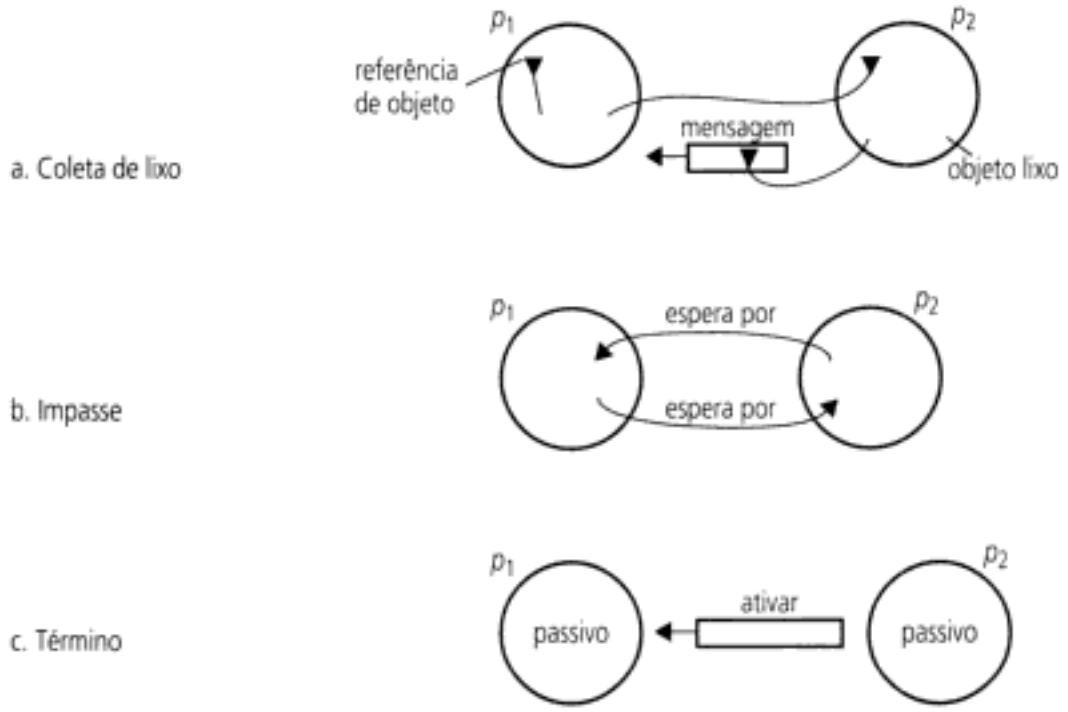


Figura 11.8 **Detectando propriedades globais.**

uma da outra. Infelizmente, há um erro no programa e ela suspeita que sob certas circunstâncias $|x_i - x_j| > \delta$ para alguns i e j, violando suas restrições de consistência. Seu problema é que esse relacionamento deve ser avaliado quanto aos valores das variáveis que ocorrem ao mesmo tempo.

Cada um dos problemas anteriores tem soluções específicas, adequadas a eles, mas todos ilustram a necessidade de observar um estado global e, portanto, motivam uma estratégia geral.

11.5.1 Estados globais e cortes consistentes

Em princípio, é possível observar a sucessão de estados de um processo individual, mas a questão de como verificar um estado global do sistema – o estado do conjunto de processos – é muito mais difícil de resolver.

O problema básico é a ausência de um tempo global. Se todos os processos tivessem relógios perfeitamente sincronizados, poderíamos concordar com um tempo no qual cada processo registraria seu estado – o resultado seria um estado global real do sistema. A partir do conjunto de estados de processo, poderíamos identificar, por exemplo, se os processos estariam em um impasse. Mas não podemos obter uma sincronização perfeita do relógio; portanto, esse método não está disponível para nós.

Assim, poderíamos perguntar se é possível reunir um estado global significativo a partir dos estados locais gravados em diferentes tempos reais. A resposta é um categórico sim, mas para ver isso, primeiro apresentaremos algumas definições.

Voltemos ao nosso sistema geral $\wp$ de N processos p_i (i = 1, 2,..., N), cuja execução queremos estudar. Dissemos anteriormente que uma série de eventos ocorre em cada processo e que podemos caracterizar a execução de cada processo por seu histórico:

$$histórico(p_i) = h_i = < e_i^0\ e_i^1\ e_i^2 ... >$$

Analogamente, podemos considerar qualquer prefixo finito do histórico do processo:

$$h_i^k = < e_i^0\ e_i^1 ...\ e_i^k >$$

Cada evento é uma ação interna do processo (por exemplo, a atualização de uma de suas variáveis) ou o envio e a recepção de mensagens pelos canais de comunicação que ligam os processos.

Hidden page

eventos aconteceram simultaneamente, ainda assim podemos considerar que eles ocorreram em uma ordem definida – digamos, ordenados de acordo com os identificadores de processo. (Os eventos que ocorrem simultaneamente devem ser concorrentes: nenhum aconteceu antes do outro.) Um sistema evolui dessa maneira pelos estados globais consistentes.

Uma *série* (*run*) é a ordenação total de todos os eventos em um histórico global que é consistente com cada ordem do histórico local, $\rightarrow_i (i = 1, 2, ..., N)$. Uma *linearização* ou *série consistente* é uma ordenação dos eventos em um histórico global que é consistente com essa relação antes do acontecido $\rightarrow$ em H. Note que uma linearização também é uma série.

Nem todas as séries passam pelos estados globais consistentes, mas todas as linearizações passam apenas pelos estados globais consistentes. Dizemos que um estado S' *pode ser atingido* a partir de um estado S, se há uma linearização que passa por S e depois por S'.

Às vezes, podemos alterar a ordem dos eventos concorrentes dentro de uma linearização e derivar uma série que ainda passa somente pelos estados globais consistentes. Por exemplo, se dois eventos sucessivos em uma linearização são a recepção de mensagens por dois processos, então podemos trocar a ordem desses dois eventos.

11.5.2 Predicados de estado global, estabilidade, segurança e subsistência

Detectar uma condição como um impasse ou término significa avaliar o *predicado de um estado global*. O predicado de um estado global é uma função que faz o mapeamento do conjunto de estados globais de processos no sistema $\wp$ para {*Verdadeiro*, *Falso*}. Uma das características úteis dos predicados, associada ao estado de um objeto ser lixo, do sistema estar em um impasse, ou do sistema ter terminado, é que todos são estáveis: quando o sistema entra em um estado no qual o predicado é *Verdadeiro*, ele permanece *Verdadeiro* em todos os estados futuros que podem ser atingidos a partir desse estado. Em contraste, quando monitoramos, ou depuramos, uma aplicação, estamos freqüentemente interessados nos predicados não-estáveis, como o nosso exemplo de variáveis cuja diferença supostamente é limitada. Mesmo que a aplicação atinja um estado no qual o limite seja obtido, ele não precisa permanecer nesse estado.

Também notamos aqui mais duas noções relevantes aos predicados do estado global: segurança (*safety*) e subsistência (*liveness*). Suponha que exista uma propriedade indesejável α, que é um predicado do estado global do sistema – por exemplo, α poderia ser a propriedade de estar em um impasse. Seja S_0 o estado original do sistema. A *segurança* com relação à α é a afirmação de que α é avaliado como *Falso* para todos os estados S que podem ser atingidos a partir de S_0. Inversamente, seja β uma propriedade desejável do estado global de um sistema – por exemplo, a propriedade de atingir o término. A *subsistência* com relação à β é a propriedade de que, para qualquer linearização L começando no estado S_0, β é avaliado como *Verdadeiro* para algum estado S_L que pode ser atingido a partir de S_0.

11.5.3 O algoritmo do instantâneo de Chandy e Lamport

Chandy e Lamport [1985] descrevem um algoritmo do instantâneo (*snapshot*) para determinar os estados globais de sistemas distribuídos, o qual apresentaremos agora. O objetivo do algoritmo é gravar um conjunto de estados de processo e do canal (um "instantâneo") para um conjunto de processos p_i $(i = 1, 2, ..., N)$ tal que, mesmo que a combinação dos estados gravados nunca possa ter ocorrido ao mesmo tempo, o estado global gravado é consistente.

Veremos que o estado gravado pelo algoritmo do instantâneo tem propriedades convenientes para avaliar predicados globais estáveis.

O algoritmo grava o estado de forma local nos processos; ele não fornece um método para agrupar o estado global em um site. Um método óbvio de agrupar é fazer todos os processos enviarem o estado que gravaram para um processo coletor designado, mas não vamos tratar desse problema aqui.

O algoritmo presume que:

- nem os canais, nem os processos falham; a comunicação é confiável, de modo que toda mensagem enviada é recebida intacta, exatamente uma vez;
- os canais são unidirecionais e fornecem entrega de mensagens com ordenamento FIFO;

Hidden page

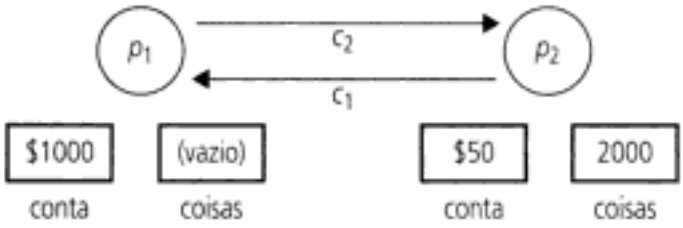


Figura 11.11 Dois processos e seus estados iniciais.

Ilustramos o algoritmo para um sistema de dois processos, p_1 e p_2, conectados por dois canais unidirecionais, c_1 e c_2. Os dois processos negociam "coisas". O processo p_1 envia requisição de "uma coisa" para p_2, por c_2, contendo um pagamento na razão de $10 por "coisa". Algum tempo depois, o processo p_2 envia "coisas" para p_1 pelo canal c_1. Os processos têm os estados iniciais mostrados na Figura 11.11. O processo p_2 já recebeu uma requisição de cinco "coisas", as quais serão rapidamente enviadas para p_1.

A Figura 11.12 mostra uma execução do sistema enquanto o estado é gravado. O processo p_1 grava seu estado no estado global real S_0, quando o estado de p_1 é <$1000, 0>. Seguindo a regra de envio de marcador, o processo p_1 emite então uma mensagem de marcador por seu canal de saída c_2, antes de enviar a próxima mensagem em nível de aplicação: (Pedido 10, $100) pelo canal c_2. O sistema entra no estado global S_1.

Antes que p_2 receba o marcador, ele emite uma mensagem de aplicação (cinco "coisas") por c_1, em resposta ao pedido anterior de p_1, gerando um novo estado global S_2.

Agora, o processo p_1 recebe a mensagem de p_2 (cinco "coisas") e p_2 recebe o marcador. Seguindo a regra de recepção de marcador, p_2 grava seu estado como <$50, 1995> e o do canal c_2 como a seqüência vazia. Seguindo a regra de envio de marcador, ele envia uma mensagem de marcador por c_1.

Quando o processo p_1 recebe a mensagem de marcador de p_2, ele grava o estado do canal c_1 como a única mensagem (cinco "coisas") que recebeu após ter gravado seu estado pela primeira vez. O estado global final é S_3.

O estado gravado final é p_1: <$1000, 0>; p_2: <$50, 1995>; c_1: <(cinco "coisas")>; c_2: < >. Note que esse estado difere de todos os estados globais pelos quais o sistema realmente passou.

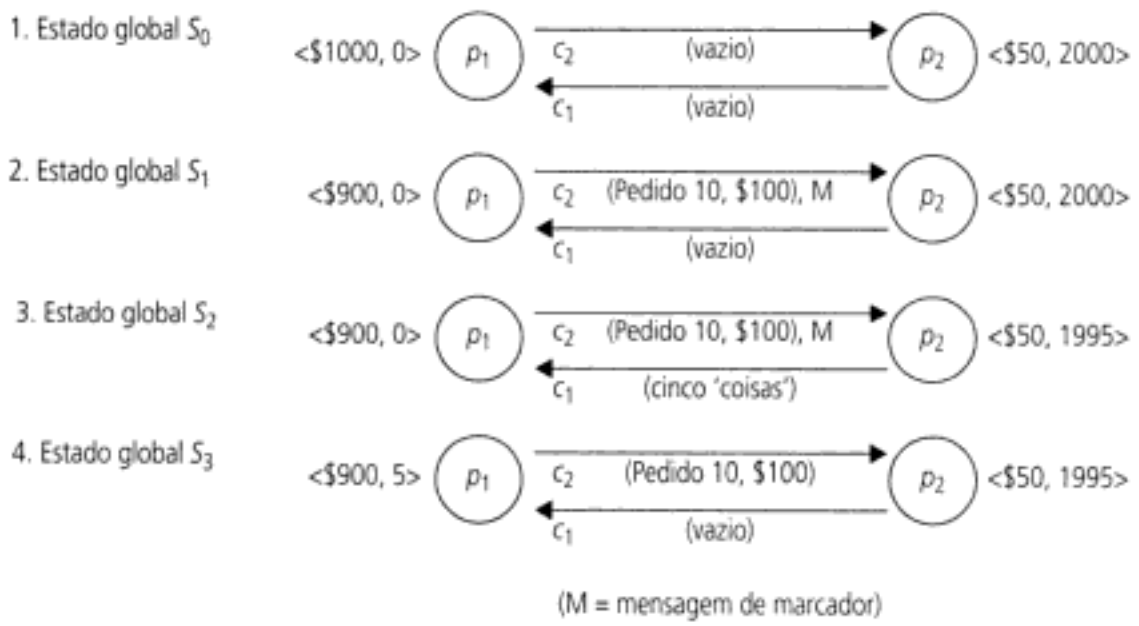


Figura 11.12 A execução dos processos da Figura 11.11.

Término do algoritmo do instantâneo ◊ Suponhamos que um processo que recebeu uma mensagem de marcador grava seu estado dentro de um tempo finito e envie mensagens de marcador através de cada canal de saída dentro de um tempo finito (mesmo quando não precisa mais enviar mensagens de aplicação por esses canais). Se houver um caminho de canais de comunicação de um processo p_i para um processo p_j ($j \neq i$), então fica claro, a partir dessas suposições, que p_j gravará seu estado em um tempo finito após p_i ter gravado seu estado. Como estamos supondo que o grafo de processos e canais é fortemente conectado, segue-se que todos os processos terão gravado seus estados e os estados dos canais de entrada em um tempo finito após, inicialmente, um processo gravar seu estado.

Caracterizando o estado observado ◊ O algoritmo do instantâneo seleciona um corte a partir do histórico da execução. O corte e, portanto, o estado gravado por esse algoritmo, é consistente. Para ver isso, sejam e_i e e_j eventos que ocorrem em p_i e p_j respectivamente, tal que $e_i \rightarrow e_j$. Afirmamos que, se e_j está no corte, então e_i está no corte. Isto é, se e_j ocorreu antes de p_j ter gravado seu estado, então e_i deve ter ocorrido antes que p_i gravasse seu estado. Isso é evidente se os dois processos são o mesmo; portanto, vamos supor que $j \neq i$. Suponha, por enquanto, o oposto do que queremos provar: que p_i gravou seu estado antes que e_i ocorresse. Considere a seqüência de mensagens de H, $m_1, m_2, ..., m_H (H \geq 1)$, dando origem à relação $e_i \rightarrow e_j$. Pela ordem FIFO nos canais que essas mensagens atravessam, e pelas regras de envio e recepção de marcador, uma mensagem de marcador teria chegado a p_j na frente de $m_1, m_2, ..., m_H$. Pela regra de recepção de marcador, p_j teria, portanto, gravado seu estado antes do evento e_j. Isso contradiz nossa suposição de que e_j está no corte, e terminamos.

Podemos estabelecer ainda uma relação de alcance entre o estado global observado e os estados globais iniciais e finais, quando o algoritmo é executado. Seja $Sys = e_0, e_1, ...$ a linearização do sistema ao ser executado (onde dois eventos ocorreram exatamente ao mesmo tempo, os ordenamos de acordo com os identificadores de processo). Seja S_{inic} o estado global imediatamente antes que o primeiro processo tenha gravado seu estado; seja S_{final} o estado global de quando o algoritmo do instantâneo termina, imediatamente após a ação de gravação do último estado; e seja S_{inst} o estado global gravado.

Encontraremos uma permutação de Sys, $Sys' = e'_0, e'_1, e'_2, ...$ tal que todos os três estados S_{inic}, S_{inst} e S_{final} ocorrem em Sys', S_{inst} pode ser atingido a partir de S_{inic} em Sys' e S_{final} pode ser atingido a partir de S_{inst} em Sys'. A Figura 11.13 mostra essa situação, na qual a linearização superior é Sys e a linearização inferior é Sys'.

Derivamos Sys' de Sys, primeiro classificando todos os eventos em Sys como eventos *anteriores ao instantâneo* ou eventos *posteriores ao instantâneo*. Um evento anterior ao instantâneo no processo p_i é aquele que ocorreu em p_i antes que ele gravasse seu estado; todos os outros eventos são posteriores ao instantâneo. É importante entender que um evento posterior ao instantâneo pode ocorrer antes de um evento anterior ao instantâneo em Sys, caso os eventos ocorram em diferentes processos. (É claro que nenhum evento posterior ao instantâneo pode ocorrer antes de um evento anterior ao instantâneo no mesmo processo.)

Mostraremos como podemos ordenar todos os eventos anteriores ao instantâneo antes dos eventos posteriores ao instantâneo, para obtermos Sys'. Suponha que e_j seja um evento posterior ao instantâneo em um processo, e que e_{j+1} seja um evento anterior ao instantâneo em um processo diferente. Não pode ser que $e_j \rightarrow e_{j+1}$. Para isso, esses dois eventos seriam o envio e a recepção de uma mensagem, respectivamente. Uma mensagem de marcador teria de ter precedido a mensagem, tornando a recepção

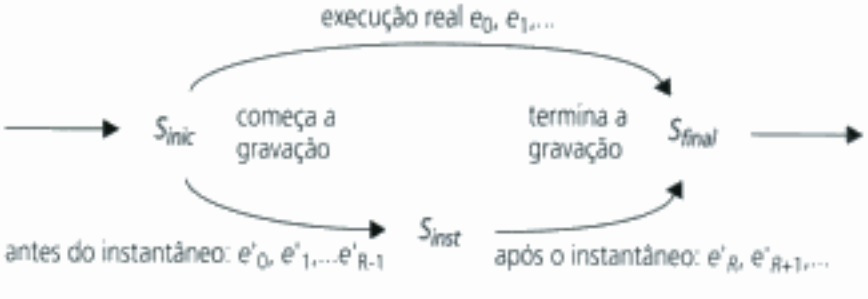


Figura 11.13 Alcançabilidade de estados no algoritmo do instantâneo.

da mensagem um evento posterior ao instantâneo, mas, por suposição, e_{j+1} é um evento anterior ao instantâneo. Portanto, podemos trocar os dois eventos sem violar a relação antes do acontecido (isto é, a seqüência resultante de eventos continua sendo uma linearização). A troca não introduz novos estados de processo, pois não alteramos a ordem em que os eventos ocorrem em nenhum processo individual.

Continuamos a trocar pares de eventos adjacentes dessa maneira, conforme for necessário, até termos ordenado todos os eventos anteriores ao instantâneo $e'_0, e'_1, e'_2, \ldots, e'_{R-1}$, antes de todos os eventos posteriores ao instantâneo $e'_{R-1}, e'_R, e'_{R+2}, \ldots$, com Sys' sendo a execução resultante. Para cada processo, o conjunto de eventos em $e'_0, e'_1, e'_2, \ldots, e'_{R-1}$ que ocorreram nele é exatamente o conjunto de eventos que ele experimentou antes de ter gravado seu estado. Portanto, o estado de cada processo nesse ponto e o estado dos canais de comunicação é o do estado global S_{inst} gravado pelo algoritmo. Não perturbamos nenhum dos dois estados S_{inic} ou S_{final} com os quais a linearização começa e termina. Portanto, estabelecemos a relação de alcançabilidade.

Estabilidade e alcançabilidade do estado observado ◊ A propriedade de alcançabilidade do algoritmo do instantâneo é útil para detectar predicados estáveis. Em geral, todo predicado não-estável que estabelecemos como sendo *Verdadeiro* no estado S_{inst} pode ou não ter sido *Verdadeiro* na execução real cujo estado global gravamos. Entretanto, se um *predicado estável* é *Verdadeiro* no estado S_{inst}, então podemos concluir que o predicado é *Verdadeiro* no estado S_{final}, pois, por definição, um predicado estável que é *Verdadeiro* em um estado S, também é *Verdadeiro* em qualquer estado que possa ser atingido a partir de S. Analogamente, se o predicado for avaliado como *Falso* para S_{inst}, então ele também deverá ser *Falso* para S_{inic}.

11.6 Depuração distribuída

Examinaremos agora o problema da gravação do estado global de um sistema, para que possamos fazer mais declarações úteis a respeito do fato de um estado transitório – em oposição a um estado estável – ter ocorrido em uma execução real. Em geral, é isso que exigimos na depuração de um sistema distribuído. Demos anteriormente um exemplo no qual cada processo em um conjunto de processos p_i tem uma variável x_i. A condição de segurança exigida nesse exemplo é $|x_i - x_j| \leq \delta$ $(i, j = 1, 2, \ldots N)$; essa restrição deve ser satisfeita mesmo que um processo possa alterar o valor de sua variável a qualquer momento. Outro exemplo é um sistema distribuído controlando um sistema de dutos em uma usina, onde estamos interessados em saber se todas as válvulas (controladas por diferentes processos) foram abertas em algum momento. Nesses exemplos, em geral, não podemos observar os valores das variáveis, ou os estados das válvulas, simultaneamente. O desafio é monitorar a execução do sistema com o passar do tempo – para capturar informações de rastreamento, em vez de um único instantâneo –, de modo que possamos estabelecer *post hoc* se a condição de segurança exigida foi, ou pode ter sido, violada.

O algoritmo do instantâneo de Chandy e Lamport reúne o estado de maneira distribuída e mostramos como os processos no sistema poderiam enviar o estado que agrupam para um processo monitor coletá-las. O algoritmo que vamos descrever (atribuído a Marzullo e Neiger [1991]) é centralizado. Os processos observados enviam seus estados para um processo chamado monitor, o qual monta estados globalmente consistentes a partir do que recebe. Consideramos que o monitor reside fora do sistema, observando sua execução.

Nosso objetivo é determinar os casos em que determinado predicado do estado global ϕ era definitivamente *Verdadeiro* em algum ponto na execução que observamos, e os casos em que ele era possivelmente *Verdadeiro*. A noção de possivelmente surge como um conceito natural, pois podemos extrair um estado global consistente S a partir de um sistema em execução, e verificar se $\phi(S)$ é *Verdadeiro*. Nenhuma observação única de um estado global consistente nos permite concluir se um predicado não-estável foi avaliado como *Verdadeiro* na execução real. Contudo, podemos estar interessados em saber se eles *poderiam* ter ocorrido, na medida do que podemos identificar observando a execução.

A noção de definitivamente se aplica à execução real e não a uma série que extrapolamos a partir dela. Pode parecer paradoxal considerarmos o que aconteceu em uma execução real. Entretanto, é possível avaliar se ϕ era definitivamente *Verdadeiro*, considerando todas as linearizações dos eventos observados.

Agora, definiremos as noções de *possivelmente* ϕ e *definitivamente* ϕ para um predicado ϕ, em termos de linearizações de H, o histórico da execução do sistema.

possivelmente ϕ A declaração *possivelmente* ϕ significa que existe um estado global consistente S pelo qual passa uma linearização de H, tal que $\phi(S)$ é *Verdadeiro*.

definitivamente ϕ A declaração *definitivamente* ϕ significa que para todas as linearizações L de H, existe um estado global consistente S pelo qual L passa, tal que $\phi(S)$ é *Verdadeiro*.

Quando usamos o algoritmo do instantâneo de Chandy e Lamport e obtemos o estado global S_{inst}, podemos declarar *possivelmente* ϕ, se $\phi(S_{inst})$ for *Verdadeiro*. Mas, em geral, avaliar *possivelmente* ϕ impõe uma busca por todos os estados globais consistentes derivados da execução observada. Somente se $\phi(S)$ for avaliado como *Falso* para todos os estados globais consistentes S não será o caso de *possivelmente* ϕ. Note também que, embora possamos concluir *definitivamente* $(\neg\phi)$ a partir de $\neg$*possivelmente* ϕ, não podemos concluir $\neg$*possivelmente* ϕ a partir de *definitivamente* $(\neg\phi)$. Esta última é a afirmação de que $\neg\phi$ vale em algum estado em cada linearização: ϕ pode valer para outros estados.

Agora, descreveremos:

- como os estados de processo são coletados;
- como o monitor extrai estados globais consistentes;
- como o monitor avalia *possivelmente* ϕ e *definitivamente* ϕ em sistemas assíncronos e síncronos.

Coletando o estado ◊ Os processos observados p_i $(i = 1, 2, ..., N)$ inicialmente enviam seus estados iniciais para o processo monitor e, depois disso, periodicamente, em mensagens de estado. O processo monitor grava as mensagens de estado do processo p_i em uma fila separada Q_i, para cada $i = 1, 2, ..., N$.

A atividade de preparar e enviar mensagens de estado pode atrasar a execução normal dos processos observados, mas não interfere nela de outra forma. Não há necessidade de enviar o estado, exceto inicialmente e quando ele muda. Existem duas otimizações para reduzir o tráfego de mensagens de estado no monitor. Primeiro, o predicado do estado global pode depender apenas de certas partes dos estados dos processos. Por exemplo, ele pode depender apenas dos estados de variáveis específicas. Portanto, os processos observados só precisam enviar o estado relevante para o processo monitor. Segundo, eles só precisam enviar seus estados nos momentos em que o predicado ϕ pode se tornar *Verdadeiro*, ou deixar de ser *Verdadeiro*. Não há motivos para enviar alterações no estado que não afetem o valor do predicado.

Por exemplo, no exemplo de sistema de processos p_i que devem obedecer à restrição $|x_i - x_j| \leq \delta$ $(i = 1, 2, ..., N)$, os processos só precisam notificar o monitor quando os valores de suas próprias variáveis x_i mudam. Quando enviam seus estados, eles fornecem o valor de x_i, mas não precisam enviar nenhuma outra variável.

11.6.1 Observando estados globais consistentes

O monitor deve montar estados globais consistentes nos quais avaliará ϕ. Lembre-se de que um corte C é consistente se e somente se para todos os eventos e no corte C, $f \rightarrow e \Rightarrow \in C$.

Por exemplo, a Figura 11.14 mostra dois processos p_1 e p_2 com variáveis x_1 e x_2, respectivamente. Os eventos mostrados nas linhas do tempo (com indicações de tempo vetoriais) são ajustes nos valores das duas variáveis. Inicialmente, $x_1 = x_2 = 0$. O requisito é $|x_1 - x_2| \leq 50$. Os processos fazem ajustes em suas variáveis, mas ajustes "grandes" fazem uma mensagem, contendo o novo valor, ser enviada para o outro processo. Quando um dos dois processos recebe uma mensagem de ajuste do outro, ele configura sua variável com um valor igual ao contido na mensagem.

Quando um dos processos p_1 ou p_2 ajusta o valor de sua variável (seja um ajuste "pequeno" ou "grande"), ele envia o valor em uma mensagem de estado para o processo monitor. Este último mantém as mensagens de estado, em filas por processo, para efeitos de análise. Se os processos monitores usassem valores do corte inconsistente C_1 da Figura 11.14, então descobriria que $x_1 = 1$, $x_2 = 100$, violando a restrição $|x_1 - x_2| \leq 50$. Mas esse estado nunca ocorreu. Por outro lado, valores do corte consistente C_2 mostram $x_1 = 105$, $x_2 = 90$.

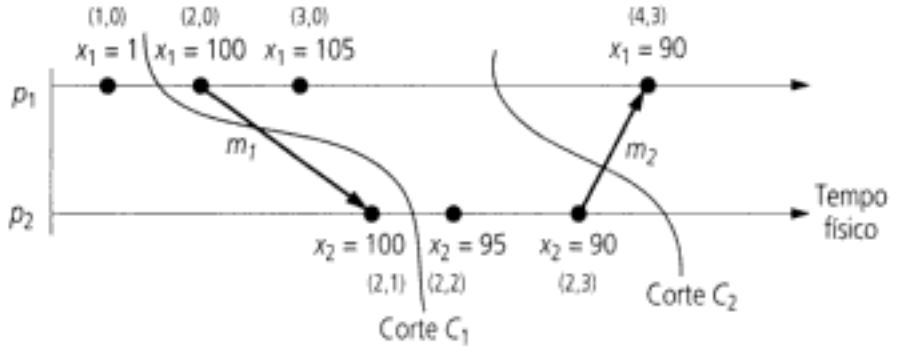


Figura 11.14 Indicações de tempo vetoriais e valores de variável para a execução da Figura 11.9.

Para que o monitor possa distinguir estados globais consistentes de estados globais inconsistentes, os processos observados incluem seus valores de relógio vetorial com suas mensagens de estado. Cada fila Q_i é mantida na ordem de envio, a qual pode ser imediatamente estabelecida examinando-se o i-ésimo componente das indicações de tempo vetoriais. É claro que o processo monitor não pode deduzir nada sobre a ordem dos estados enviados por processos diferentes a partir de sua ordem de chegada, devido às latências de mensagem variáveis. Em vez disso, ele precisa examinar as indicações de tempo vetoriais das mensagens de estado.

Seja $S = (s_1, s_2, \ldots, s_N)$ um estado global extraído das mensagens de estado recebidas pelo processo monitor. Seja $V(S_i)$ a indicação de tempo vetorial do estado s_i recebida de p_i. Então, pode ser mostrado que S é um estado global consistente (EGC) se e somente se:

$$V(s_i)[i] \geq V(s_j)[i] \text{ para } i, j = 1, 2, \ldots, N - \text{(Condição } EGC\text{)}$$

Isso diz que o número de eventos de p_i, conhecidos em p_j, ao enviar s_j não é maior do que o número de eventos que tinham ocorrido em p_i quando ele enviou s_i. Em outras palavras, se o estado de um processo depende de outro (de acordo com a ordenação antes do acontecido), então o estado global também abrange o estado do qual ele depende.

Resumindo, agora temos um método pelo qual o processo monitor pode estabelecer se determinado estado global é consistente, usando as indicações de tempo vetoriais mantidas pelos processos observados e levados de carona nas mensagens de estado enviadas a ele.

A Figura 11.15 mostra a treliça de estados globais consistentes correspondente à execução dos dois processos da Figura 11.14. Essa estrutura captura a relação de alcançabilidade entre estados globais consistentes. Os nós denotam estados globais e os arcos denotam as possíveis transições entre esses estados. O estado global S_{00} tem os dois processos em seu estado inicial; S_{10} tem p_2 ainda em seu

Figura 11.15 Treliça de estados globais para a execução da Figura 11.14.

estado inicial e p_1, no próximo estado em seu histórico local. O estado S_{01} não é consistente, devido à mensagem m_1 enviada de p_1 para p_2; portanto, ele não aparece na treliça.

A treliça é organizada em níveis com, por exemplo, S_{00} no nível 0 e S_{10} no nível 1. Generalizando, S_{ij} está no nível $(i + j)$. Uma linearização percorre a treliça de qualquer estado global para qualquer estado global que pode ser atingido a partir dele no próximo nível – isto é, em cada etapa, algum processo experimenta um evento. Por exemplo, S_{22} pode ser atingido a partir de S_{20}, mas S_{22} não pode ser atingido a partir de S_{30}.

A treliça nos mostra todas as linearizações correspondentes a um histórico. Em princípio, agora está claro como um processo monitor deve avaliar *possivelmente* ϕ e *definitivamente* ϕ. Para avaliar *possivelmente* ϕ, o processo monitor começa no estado inicial e percorre todos os estados consistentes que podem ser atingidos a partir desse ponto, avaliando ϕ em cada estágio. Ele pára quando ϕ é avaliado como *Verdadeiro*. Para avaliar *definitivamente* ϕ, o processo monitor precisa tentar descobrir um conjunto de estados através dos quais todas as linearizações devem passar, e em cada um dos quais ϕ é avaliado como *Verdadeiro*. Por exemplo, se na Figura 11.15 $\phi(S_{30})$ e $\phi(S_{21})$ são *Verdadeiros*, então, como todas as linearizações passam por esses estados, *definitivamente* ϕ vale.

11.6.2 Avaliando possivelmente ϕ

Para avaliar *possivelmente* ϕ, o processo monitor precisa percorrer a treliça dos estados que podem ser atingidos, partindo do estado inicial $(s_1^0, s_2^0 \ldots s_N^0)$. O algoritmo aparece na Figura 11.16. O algoritmo presume que a execução é infinita. Ele pode ser facilmente adaptado para uma execução finita.

O processo monitor pode descobrir o conjunto de estados consistente no nível $L + 1$ que podem ser atingidos a partir de determinado estado consistente no nível L, seguindo-se o método. Seja $S = (s_1, s_2, \ldots, s_N)$ um estado consistente. Então, um estado consistente no próximo nível, que pode ser atingido a partir de S, tem a forma $S' = (s_1, s_2, \ldots s_i', \ldots s_N)$, que difere de S somente por conter o próximo estado (após um único evento) de algum processo p_i. O monitor pode encontrar todos esses estados percorrendo as filas de mensagens de estado Q_i (1, 2,..., N). O estado S' pode ser atingido a partir de S se e somente se:

para $j = 1, 2, \ldots, N, j \neq i$: $V(s_j)[j] \geq V(s_i')[j]$

1. *Avaliando possivelmente ϕ para o histórico global H de N processos*
   $L := 0$;
   Estados := { $(s_1^0, s_2^0 \ldots s_N^0)$ };
   while ($\phi(S)$ = *Falso* para todo $S \in$ *Estados*
   $L := L + 1$;
   Alcançável := { S': S' pode ser atingido em H a partir de um $S \in$ *Estados* $\wedge$ *nível*(S') = L };
   Estados := *Alcançável*
   end while
   output "*possivelmente* ϕ";

2. *Avaliando definitivamente ϕ para o histórico global H de N processos*
   $L := 0$;
   if ($\Phi(s_1^0, s_2^0, \ldots, s_N^0)$) *then Estados* := {} *else Estados* := { $(s_1^0, s_2^0 \ldots s_N^0)$ };
   while (*Estados* $\neq$ { })
   $L := L + 1$;
   Alcançavel := {S': S' pode ser atingido em H a partir de algum $S \in$ *Estados* $\wedge$ *nível*(S') = L };
   Estados := { $S \in$ *Alcançável*: $\phi(S)$ = *Falso*}
   end while
   output "*definitivamente* ϕ";

Figura 11.16 Algoritmos para avaliar *possivelmente* ϕ e *definitivamente* ϕ.

Hidden page

Em um sistema síncrono, suponha que os processos mantenham seus relógios físicos internamente sincronizados dentro de um limite conhecido, e que os processos observados forneçam indicações de tempo físicas, assim como indicações de tempo vetoriais, em suas mensagens de estado. Então, o processo monitor precisa considerar apenas os estados globais consistentes cujos estados locais poderiam ter existido simultaneamente, dado o sincronismo aproximado dos relógios. Com um sincronismo suficientemente bom do relógio, eles serão em muito menor número do que todos os estados globalmente consistentes.

Agora, mostraremos um algoritmo para explorar relógios sincronizados dessa maneira. Supomos que cada processo observado $p_i (i = 1, 2, ..., N)$ e o processo monitor, que chamaremos de p_0, mantêm um relógio físico $C_i (i = 0, 1, ..., N)$. Eles são sincronizados dentro de um limite conhecido $D > 0$; isto é, no mesmo tempo real:

$|C_i(t) - C_j(t)| < D$ para $i, j = 0, 1, ..., N$

Os dois processos observados enviam seu tempo vetorial e seu tempo físico com suas mensagens de estado para o processo monitor. Agora, o processo monitor aplica uma condição que não apenas testa a consistência de um estado global $S = (s_1, s_2, ..., s_N)$, mas também testa se cada par de estados poderia ter acontecido no mesmo tempo real, dados os valores de relógio físico. Em outras palavras, para $i, j = 1, 2,..., N$:

$V(s_i)[i] \geq V(s_j)[i]$ e s_i e s_j poderiam ter ocorrido no mesmo tempo real.

A primeira cláusula é a condição que usamos anteriormente. Para a segunda cláusula, note que p_i está no estado s_i a partir do momento em que notifica pela primeira vez o processo monitor, $C_i(s_i)$, até algum tempo local posterior $L_i(s_i)$, digamos, quando ocorre a próxima transição de estado em p_i. Para s_i e s_j terem obtido o mesmo tempo real, temos então, considerando o limite de sincronização do relógio:

$C_i(s_i) - D \leq C_j(s_j) \leq L_i(s_i) + D$ – ou *vice-versa* (trocando e e j).

O processo monitor deve calcular um valor para $L_i(s_i)$, o qual é medido no relógio de p_i. Se o processo monitor tiver recebido uma mensagem de estado do próximo estado de s'_i de p_i, então $L_i(s_i)$ é $C_i(s'_i)$. Caso contrário, o processo monitor estima $L_i(s_i)$ como $C_0 - max + D(s_i)$, onde C_0 é o valor corrente do relógio local do monitor e max é o tempo de transmissão máximo para uma mensagem de estado.

11.7 Resumo

Este capítulo começou descrevendo a importância da indicação precisa do tempo para sistemas distribuídos. Em seguida, descreveu algoritmos para sincronizar relógios apesar da derivação entre eles e da variabilidade dos atrasos de mensagem entre computadores.

O grau da precisão da sincronização que pode realmente ser obtido atende a muitos requisitos, mas não é suficiente para determinar a ordem de dois eventos arbitrários ocorrendo em diferentes computadores. A relação antes do acontecido é uma ordem parcial dos eventos, que reflete um fluxo de informações entre eles – dentro de um processo, ou por meio de mensagens entre processos. Alguns algoritmos exigem que os eventos sejam colocados na ordem antes do acontecido; por exemplo, sucessivas atualizações feitas em cópias separadas dos dados. Os relógios de Lamport são contadores atualizados de acordo com o relacionamento antes do acontecido entre eventos. Os relógios vetoriais são um aprimoramento dos relógios de Lamport, pois é possível determinar, examinando suas indicações de tempo vetoriais, se dois eventos estão ordenados pelo relacionamento antes do acontecido ou se são concorrentes.

Apresentamos os conceitos de eventos, históricos locais e globais, cortes, estados locais e globais, séries, estados consistentes, linearizações (séries consistentes) e alcance. Um estado, ou série consistente, é aquele que está de acordo com a relação antes do acontecido.

Passamos a considerar o problema da gravação de um estado global consistente pela observação da execução de um sistema. Nosso objetivo foi avaliar um predicado nesse estado. Uma classe importante de predicados são os predicados estáveis. Descrevemos o algoritmo do instantâneo de Chandy

e Lamport, o qual captura um estado global consistente e nos permite fazer afirmações sobre se um predicado estável vale na execução real. Mostramos o algoritmo de Marzullo e Neiger para inferir afirmações sobre se um predicado vale ou pode ter valido no curso real. O algoritmo emprega um processo monitor para agrupar estados. O monitor examina indicações de tempo vetoriais para extrair estados globais consistentes, e constrói e examina a treliça de todos os estados globais consistentes. Esse algoritmo envolve muita complexidade computacional, mas é valioso entendê-lo e o mesmo pode trazer algum benefício prático em sistemas reais, onde relativamente poucos eventos mudam o valor do predicado global. O algoritmo tem uma variante mais eficiente nos sistemas síncronos, onde os relógios podem ser sincronizados.

Exercícios

11.1 Por que a sincronização do relógio do computador é necessária? Descreva os requisitos de projeto de um sistema para sincronizar os relógios em um sistema distribuído. *página 380*

11.2 Um relógio está mostrando 10:27:54.0 (hr:min:seg), quando se descobre que ele está adiantado 4 segundos. Explique por que não é desejável configurá-lo com a hora certa nesse ponto e mostre (numericamente) como ele deve ser ajustado de modo a estar correto após terem decorrido 8 segundos. *página 383–384*

11.3 Um esquema para implementar envio de mensagens confiáveis no máximo uma vez usa relógios sincronizados para rejeitar mensagens duplicadas. Os processos colocam o valor de seus relógios locais (uma "indicação de tempo") nas mensagens que enviam. Cada receptor mantém uma tabela fornecendo, para cada processo remetente, a maior indicação de tempo de mensagem que observou. Suponha que os relógios estejam sincronizados dentro de 100 ms e que as mensagens podem chegar no máximo 50 ms após a transmissão.

(i) Quando um processo pode ignorar uma mensagem contendo uma indicação de tempo T, se tiver registrado a última mensagem recebida desse processo como tendo a indicação de tempo T'?

(ii) Quando um receptor pode remover de sua tabela uma indicação de tempo de 175.000 (ms)? (Dica: use o valor do relógio local do receptor.)

(iii) Os relógios devem ser sincronizados internamente ou externamente? *página 384*

11.4 Um cliente tenta sincronizar com um servidor de tempo. Ele grava os tempos de viagem de ida e volta e as indicações de tempo retornadas pelo servidor na tabela a seguir.

Quais desses tempos ele deve usar para configurar seu relógio? Com que tempo ele deve ser configurado? Faça uma estimativa da precisão da configuração com relação ao relógio do servidor. Se for conhecido que o tempo entre o envio e o recebimento de uma mensagem no sistema envolvido é de pelo menos 8 ms, suas respostas mudarão?

Viagem de ida e volta (ms)	*Tempo (hr:min:seg)*
22	*10:54:23.674*
25	*10:54:25.450*
20	*10:54:28.342*

página 384

11.5 No sistema do Exercício 11.4, é exigido manter a sincronização do relógio de um servidor de arquivos dentro de ±1 milissegundo. Discuta isso em relação ao algoritmo de Cristian. *página 384*

11.6 Quais reconfigurações você esperaria que ocorressem na sub-rede de sincronização NTP? *página 386–387*

11.7 Um servidor NTP B recebe uma mensagem do servidor A às 16:34:23.480, contendo uma indicação de tempo 16:34:13.430, e dá uma resposta. O servidor A recebe a mensagem às 16:34:15.725, contendo a indicação de tempo de B 16:34:25.7. Faça uma estimativa da compensação de B e A e a sua precisão. *página 387–388*

11.8 Discuta os fatores a serem levados em conta ao se decidir com qual servidor NTP um cliente deve sincronizar seu relógio. *página 387–388*

11.9 Discuta como é possível compensar a derivação de relógio entre pontos de sincronização observando a taxa de derivação com o passar do tempo. Discuta as limitações de seu método. *página 389*

11.10 Considerando um encadeamento de zero ou mais mensagens conectando os eventos e e e', e usando indução, mostre que $e \rightarrow e' \Rightarrow L(e) < L(e')$. *página 389–390*

11.11 Mostre que $V_j[i] \leq V_j[i]$. *página 390–391*

11.12 De maneira semelhante ao Exercício 11.10, mostre que $e \rightarrow e' \Rightarrow V(e) < V(e')$. *página 391–392*

11.13 Usando o resultado do Exercício 11.11, mostre que, se os eventos e e e' são concorrentes, então nem $V(e) \leq V(e')$, nem $V(e') \leq V(e)$. Assim, mostre que, se $V(e) < V(e')$, então $e \rightarrow e'$. *página 391–392*

11.14 Dois processos P e Q estão conectados em anel, usando dois canais, e constantemente fazem o rodízio de uma mensagem m. Em qualquer dado momento, existe apenas uma cópia de m no sistema. O estado de cada processo consiste no número de vezes que ele recebeu m, e P envia m primeiro. Em certo ponto, P tem a mensagem e seu estado é 101. Imediatamente após enviar m, P inicia o algoritmo do instantâneo. Explique o funcionamento do algoritmo nesse caso, fornecendo o(s) estado(s) global(is) possível(is) relatado(s) por ele. *página 395–396*

11.15 A figura acima mostra eventos ocorrendo para cada um de dois processos, p_1 e p_2. As setas entre os processos denotam transmissão de mensagens.

Desenhe e rotule a treliça de estados consistentes (estado de p_1, estado de p_2), começando com o estado inicial (0,0). *página 401–402*

11.16 Jones está executando um conjunto de processos $p_1, p_2, \ldots, p_N$. Cada processo p_i contém uma variável v_i. Ela deseja determinar se todas as variáveis $v_1, v_2, \ldots, v_N$ já foram iguais no curso da execução.

(i) Os processos de Jones são executados em um sistema síncrono. Ela usa um processo monitor para determinar se as variáveis já foram iguais. Quando os processos de aplicação devem se comunicar com o processo monitor e o que suas mensagens devem conter?

(ii) Explique a declaração *possivelmente* $(v_1 = v_2 = \ldots = v_N)$. Como Jones pode determinar se essa declaração é verdadeira para sua execução? *página 402–403*

12 Coordenação e Acordo

Neste capítulo, apresentaremos alguns tópicos e algoritmos relacionados ao problema de como os processos coordenam suas ações e entram em acordo sobre os valores compartilhados em sistemas distribuídos, a despeito das falhas. O capítulo começa com algoritmos para obter exclusão mútua dentre um conjunto de processos, de modo a coordenar seus acessos aos recursos compartilhados. Em seguida, ele examina como uma eleição pode ser implementada em um sistema distribuído. Isto é, ele descreve como um grupo de processos pode concordar sobre um novo coordenador para suas atividades, após o coordenador anterior ter falhado.

A segunda metade examina os problemas relacionados com a comunicação *multicast*, do consenso, do acordo bizantino e da consistência interativa. Na comunicação *multicast*, o problema é como entrar em acordo sobre questões como a ordem em que as mensagens devem ser enviadas. O consenso e os outros problemas são generalizados a partir deste: como um conjunto de processos pode concordar com algum valor, independentemente do domínio dos valores em questão? Encontramos um resultado fundamental na teoria dos sistemas distribuídos: que, sob certas condições – incluindo condições de falha surpreendentemente benignas – é impossível garantir que os processos cheguem a um consenso.

12.1 Introdução

Este capítulo apresenta um conjunto de algoritmos cujos objetivos variam, mas os quais compartilham uma meta fundamental em sistemas distribuídos: a de que um conjunto de processos coordene suas ações ou concorde com um ou mais valores. Por exemplo, no caso de um mecanismo complexo, como uma nave espacial, é fundamental que os computadores que a estão controlando concordem com condições como o fato de a missão da nave espacial estar prosseguindo ou ter sido cancelada. Além disso, os computadores precisam coordenar suas ações corretamente, com relação aos recursos compartilhados (os sensores e controladores da nave espacial). Os computadores devem ser capazes de fazer isso mesmo onde não haja nenhum relacionamento mestre-escravo fixo entre os componentes (o que tornaria a coordenação particularmente simples). O motivo para se evitar relacionamentos mestre-escravo fixos é que, freqüentemente, exigimos que nossos sistemas continuem funcionando corretamente, mesmo que ocorram falhas; portanto, precisamos evitar pontos de falha únicos, como os mestres fixos.

Uma distinção importante para nós, como no Capítulo 11, será se o sistema distribuído sob estudo é assíncrono ou síncrono. Em um sistema assíncrono, não podemos fazer nenhuma suposição quanto à temporização. Em um sistema síncrono, devemos supor que existem limites para o atraso máximo na transmissão das mensagens, para o tempo que leva para executar cada etapa de um processo e para as taxas de derivação de relógio. As suposições síncronas nos permitem usar tempos limites para detectar falhas de processo.

Outro objetivo importante do capítulo, ao discutirmos os algoritmos, é considerar as falhas e como tratar com elas ao projetar algoritmos. A Seção 2.3.2 apresentou um modelo de falha, o qual usaremos neste capítulo. Suportar falhas é uma atividade sutil; portanto, começaremos considerando alguns algoritmos que não toleram falhas e passaremos para as falhas benignas, até considerarmos como fazemos para tolerar falhas arbitrárias. Encontramos um resultado fundamental na teoria dos sistemas distribuídos. Mesmo sob condições de falhas benignas, em um sistema assíncrono é impossível garantir que um conjunto de processos possa concordar com um valor compartilhado – por exemplo, que todos os processos de controle de uma nave espacial concordem com o prosseguimento da missão ou com o cancelamento da missão.

A Seção 12.2 examinará o problema da exclusão mútua distribuída. Trata-se da ampliação para os sistemas distribuídos do familiar problema de evitar condições de disputa nos núcleos e em aplicativos *multi-threadeds*. Como grande parte do que ocorre nos sistemas distribuídos é o compartilhamento de recursos, esse é um problema importante a ser resolvido. Em seguida, a Seção 12.3 apresentará um problema relacionado, porém mais geral, de como "eleger" um processo a partir de um conjunto de processos, para desempenhar uma função especial. Por exemplo, no Capítulo 11, vimos como os processos sincronizavam seus relógios com um servidor de tempo designado. Se esse servidor falhar e vários servidores sobreviventes puderem desempenhar essa função, então, por causa da consistência, é necessário escolher apenas um servidor para assumir o comando.

A comunicação *multicast* é o assunto da Seção 12.4. Conforme a Seção 4.5.1 explicou, o *multicast* é um paradigma de comunicação muito útil, com aplicações desde a localização de recursos até a coordenação das atualizações em dados replicados. A Seção 12.4 examinará a confiabilidade do *multicast* e a semântica da ordenação, e fornecerá algoritmos para obter as variações. A entrega de mensagens baseada em *multicast* é basicamente um problema de acordo entre processos: os destinatários concordam com quais mensagens receberão e em que ordem as receberão. A Seção 12.5 discutirá o problema do acordo de maneira mais geral, principalmente nas formas conhecidas como consenso e acordo bizantino.

O tratamento dado neste capítulo envolve a declaração das suposições e dos objetivos a serem atingidos e dar uma explicação informal sobre o motivo pelo qual os algoritmos apresentados estão corretos. Não há espaço suficiente para oferecer uma abordagem mais rigorosa. Para isso, recomendamos ao leitor um texto que forneça uma explicação completa sobre os algoritmos distribuídos, como Attiya e Welch [1998] e Lynch [1996].

Antes de apresentarmos os problemas e algoritmos, discutiremos as suposições de falha e a questão prática da detecção de falhas em sistemas distribuídos.

12.1.1 Suposições de falhas e detectores de falhas

Por simplicidade, este capítulo presume que cada par de processos está conectado por canais confiáveis. Isto é, embora os componentes de rede subjacentes possam falhar, os processos usam um protocolo de comunicação confiável que mascara essas falhas – por exemplo, retransmitindo mensagens perdidas ou corrompidas. Também por simplicidade, presumimos que nenhuma falha de processo implica em uma ameaça à capacidade dos outros processos se comunicarem. Isso significa que nenhum dos processos depende de outro para encaminhar mensagens.

Note que um canal confiável finalmente termina por enviar uma mensagem para o buffer de entrada do destinatário. Em um sistema síncrono, supomos que existe redundância de hardware onde é necessária, de modo que um canal confiável não apenas finalmente entrega cada mensagem, a despeito das falhas subjacentes, mas faz isso dentro de um limite de tempo especificado.

Em qualquer intervalo de tempo específico, a comunicação entre alguns processos pode acontecer, enquanto a comunicação entre outros é retardada. Por exemplo, a falha de um roteador entre duas redes pode significar que um conjunto de quatro processos seja dividido em dois pares, de modo que a comunicação dentro do par é possível pelas suas respectivas redes; mas a comunicação entre pares não é possível enquanto o roteador estiver fora de operação. Isso é conhecido como *particionamento da rede* (Figura 12.1). Em uma rede ponto a ponto, como a Internet, topologias complexas e escolhas de roteamento independentes significam que a conectividade pode ser *assimétrica*: a comunicação é possível do processo p para o processo q, mas não *vice-versa*. A conectividade também pode ser *intransitiva*: a comunicação é possível de p para q e de q para r, mas p não pode se comunicar diretamente com r. Assim, nossa suposição de confiabilidade impõe que, finalmente, qualquer enlace ou roteador defeituoso será reparado ou evitado. Contudo, nem todos os processos podem ser capazes de se comunicar ao mesmo tempo.

O capítulo presume, a não ser que declarado de outra forma, que os processos só falham por causa de defeitos – uma suposição suficientemente boa para muitos sistemas. Na Seção 12.5, consideraremos como fazer para tratar dos casos onde os processos têm falhas arbitrárias (bizantinas). Qualquer que seja o tipo de falha, um processo *correto* é aquele que não apresenta falha alguma em qualquer ponto na execução sob consideração. Note que a correção se aplica à execução inteira e não apenas a uma parte dela. Portanto, um processo que sofre uma falha por colapso era "não defeituoso" antes desse ponto e "não correto" depois dele.

Um dos problemas no projeto de algoritmos que podem superar falhas de processo é o de decidir quando um processo falhou. Um *detector de falha* [Chandra e Toueg 1996, Stelling *et al.* 1998] é um serviço que os processos consultam para saber se um processo em particular falhou. Freqüentemente, ele é implementado por um objeto local para cada processo (no mesmo computador) que executa um algoritmo de detecção de falha, em conjunto com seus correlatos em outros processos. O objeto local de cada processo é chamado de *detector de falha local*. Em breve, descreveremos em linhas gerais como fazer para implementar detectores de falha, mas primeiro nos concentraremos em algumas de suas propriedades.

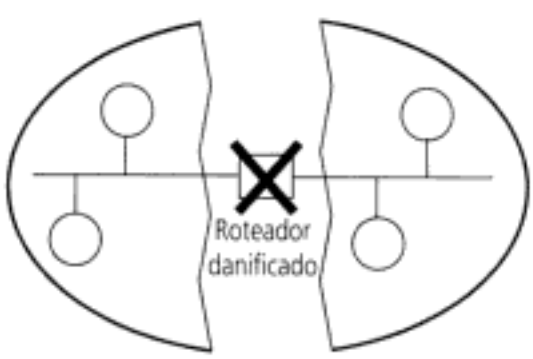


Figura 12.1 Um particionamento de rede.

Um detector de falha não é necessariamente preciso. A maioria cai na categoria dos *detectores de falha não confiáveis*. Um detector de falha não confiável pode produzir um de dois valores, dada a identidade de um processo: *não suspeito* ou *suspeito*. Esses dois resultados são sugestões, que podem, ou não, refletir precisamente se o processo realmente falhou. Um resultado *não suspeito* significa que recentemente o detector recebeu uma evidência sugerindo que o processo não falhou; por exemplo, uma mensagem foi recebida recentemente dele. Mas é claro que o processo pode ter falhado desde então. Um resultado *suspeito* significa que o detector de falha tem alguma indicação de que o processo pode ter falhado. Por exemplo, pode ser que nenhuma mensagem tenha sido recebida do processo por mais do que um período máximo nominal de silêncio (mesmo em um sistema assíncrono, limites superiores práticos podem ser usados como sugestões). A suspeita pode ser depositada em lugar errado: por exemplo, o processo poderia estar funcionando corretamente, mas no outro lado de uma partição de rede; ou poderia estar executando mais lentamente do que o esperado.

Um *detector de falha confiável* é aquele que é sempre preciso na detecção da falha de um processo. Ele responde às consultas dos processos indicando *não suspeito* – o que, como antes, pode ser apenas uma sugestão – ou *falho*. Um resultado *falho* significa que o detector determinou que o processo está em colapso. Lembre-se de que um processo em colapso permanece desse jeito, pois, por definição, um processo nunca dá outro passo, uma vez que tenha falhado.

É importante perceber que, embora estejamos falando de um detector de falha atuando para um conjunto de processos, a resposta dada por ele para um processo é apenas tão boa quanto à informação disponível nesse processo. Às vezes, um detector de falha pode dar diferentes respostas para diferentes processos, pois as condições de comunicação variam de um processo para outro.

Podemos implementar um detector de falha não confiável usando o algoritmo a seguir. Cada processo p envia uma mensagem "*p está aqui*" para cada outro processo e faz isso a cada T segundos. O detector de falha usa uma estimativa do tempo máximo de transmissão da mensagem, de D segundos. Se o detector de falha local no processo q não receber uma mensagem "*p está aqui*" dentro de $T + D$ segundos da última, então relatará para q que p é *suspeito*. Entretanto, se subseqüentemente ele receber uma mensagem "*p está aqui*", então relatará para q que p está OK.

Em um sistema distribuído real, existem limites práticos para os tempos de transmissão da mensagem. Até os sistemas de e-mail desistem após alguns dias, pois é provável que os enlaces de comunicação e roteadores tenham sido reparados nesse tempo. Se escolhermos valores pequenos para T e D (de modo que eles totalizem, digamos, 0,1 segundo), então o detector de falha provavelmente suspeitará de processos não falhos muitas vezes e muita largura de banda será ocupada com mensagens "*p está aqui*". Se escolhermos um valor grande para o tempo limite total (digamos, uma semana), os processos falhos serão freqüentemente relatados como *Não suspeitos*.

Uma solução prática para esse problema é usar valores de tempo limite que reflitam as condições de atraso observadas na rede. Se um detector de falha local recebesse uma mensagem "*p está aqui*" em 20 segundos, em vez do máximo esperado de 10 segundos, ele poderia reconfigurar seu valor de tempo limite para p de acordo com isso. O detector de falha permanece não confiável e suas respostas às consultas ainda são apenas sugestões, mas a probabilidade de sua precisão aumenta.

Em um sistema síncrono, nosso detector de falha pode se tornar confiável. Podemos escolher D de modo que não seja uma estimativa, mas um limite absoluto para os tempos de transmissão da mensagem; a ausência de uma mensagem "*p está aqui*" dentro de $T + D$ segundos autoriza o detector de falha local a concluir que p falhou.

O leitor pode estar se perguntando se os detectores de falha têm algum uso prático. Os detectores de falha não confiáveis podem suspeitar de um processo que não falhou (eles podem ser *imprecisos*); e eles podem não suspeitar de um processo que, na realidade, falhou (eles podem ser *incompletos*). Por outro lado, os detectores de falha confiáveis exigem que o sistema seja síncrono (e alguns sistemas práticos são).

Apresentamos os detectores de falha porque eles nos ajudam a pensar sobre a natureza das falhas em um sistema distribuído. E qualquer sistema prático projetado para suportar falhas deve detectá-las – ainda que imperfeitamente. Mas verifica-se que mesmo os detectores de falha não confiáveis com certas propriedades bem definidas podem nos ajudar a providenciar soluções práticas para o problema da coordenação de processos na presença de falhas. Voltaremos a esse assunto na Seção 12.5.

12.2 Exclusão mútua distribuída

Os processos distribuídos freqüentemente precisam coordenar suas atividades. Se um conjunto de processos compartilha um recurso, ou uma coleção de recursos, então, freqüentemente, a exclusão mútua é exigida para evitar interferência e garantir a consistência ao acessar esses recursos. Esse é o problema da *seção crítica*, familiar no domínio dos sistemas operacionais. Em um sistema distribuído, entretanto, nem as variáveis compartilhadas nem os recursos fornecidos por um único núcleo local podem ser usados para resolvê-lo, em geral. Precisamos de uma solução para a *exclusão mútua distribuída*: uma que seja baseada unicamente na passagem de mensagens.

Em alguns casos, os recursos compartilhados são gerenciados por servidores que também fornecem mecanismos de exclusão mútua. O Capítulo 13 descreve como alguns servidores sincronizam os acessos de cliente aos recursos. Mas, em alguns casos práticos, é exigido um mecanismo de exclusão mútua separado.

Considere usuários que atualizam um arquivo de texto. Uma maneira simples de garantir que suas atualizações sejam consistentes é permitir que apenas um usuário por vez o acesse, exigindo que o editor bloqueie o arquivo antes que as atualizações possam ser feitas. Os servidores de arquivo NFS, descritos no Capítulo 8, são projetados para serem sem estado e, portanto, não suportam bloqueio de arquivo. Por isso, os sistemas UNIX fornecem um serviço de bloqueio de arquivo separado, implementado pelo daemon *lockd*, para tratar dos pedidos de bloqueio dos clientes.

Um exemplo particularmente interessante é aquele onde não há nenhum servidor e um conjunto de processos pares precisa coordenar seus acessos aos recursos compartilhados entre eles mesmos. Isso ocorre rotineiramente em redes como as Ethernet e em redes IEEE 802.11 sem fio no modo *ad hoc*, onde as interfaces de rede cooperam como pares para que apenas um nó por vez transmita no meio compartilhado. Considere também um sistema monitorando o número de vagas em um estacionamento, com um processo em cada entrada e saída, controlando o número de veículos que entram e que saem. Cada processo mantém uma contagem do número total de veículos dentro do estacionamento e mostra se ele está lotado ou não. Os processos devem atualizar consistentemente a contagem compartilhada do número de veículos. Existem várias maneiras de conseguir isso, mas seria conveniente que esses processos pudessem obter exclusão mútua unicamente se comunicando entre si, eliminando a necessidade de um servidor separado.

É útil ter um mecanismo genérico de exclusão mútua distribuída à nossa disposição, que seja independente do esquema de gerenciamento de recursos específico em questão. Examinaremos agora alguns algoritmos para conseguir isso.

12.2.1 Algoritmos de exclusão mútua

Consideramos um sistema de N processos p_i, $i = 1, 2, \ldots, N$ que não compartilham variáveis. Os processos acessam recursos comuns, mas fazem isso em uma seção crítica. Por simplicidade, supomos que existe apenas uma seção crítica. É fácil estender os algoritmos que apresentaremos para mais de uma seção crítica.

Supomos que o sistema seja assíncrono, que os processos não falham, e que o envio de mensagens é confiável, de modo que toda mensagem enviada finalmente é entregue intacta, exatamente uma vez.

O protocolo em nível de aplicativo para executar uma seção crítica é o seguinte:

```
enter()              // entra na seção crítica – bloqueia, se necessário
resourceAccesses()   // acessa recursos compartilhados na seção crítica
exit()               // sai da seção crítica – outros processos podem entrar agora
```

Nossos requisitos básicos de exclusão mútua são os seguintes:

EM1: (segurança) No máximo um processo por vez pode ser executado na seção crítica (SC).

EM2: (subsistência) Os pedidos para entrar e sair da seção crítica têm sucesso.

A condição EM2 implica em independência de impasse e inanição. Um impasse envolveria dois ou mais processos travando indefinidamente, enquanto tentam entrar ou sair da seção crítica, devido

Hidden page

trar na seção crítica. Se nenhum outro processo tiver o *token* no momento do pedido, então o servidor responderá imediatamente, concedendo o *token*. Se o *token* estiver de posse de outro processo, então o servidor não responderá, mas enfileirará o pedido. Na saída da seção crítica, uma mensagem é enviada para o servidor, devolvendo o *token* a ele.

Se a fila de processos em espera não estiver vazia, o servidor escolherá a entrada mais antiga, a removerá e responderá para o processo correspondente. Então, o processo escolhido terá a posse do *token*. Na figura, mostramos uma situação em que o pedido de p_2 foi anexado na fila, a qual já continha o pedido de p_4. Quando p_3 sai da seção crítica, o servidor remove a entrada de p_4 e concede a permissão para entrar na seção crítica enviando uma resposta a ele. Correntemente, o processo p_1 não solicita entrada na seção crítica.

Dada nossa suposição de que não ocorrem falhas, é fácil ver que as condições de segurança e subsistência são satisfeitas por esse algoritmo. Entretanto, o leitor deve verificar que o algoritmo não satisfaz a propriedade EM3.

Avaliaremos agora o desempenho desse algoritmo. A entrada na seção crítica – mesmo quando nenhum processo a ocupa correntemente – exige duas mensagens (um *pedido*, seguido de uma *concessão*) e atrasa o processo de solicitação pelo tempo dessa viagem de ida e volta. A saída da seção crítica exige uma única mensagem de *liberação*. Supondo a passagem de mensagens assíncrona, isso não atrasa o processo de saída.

O servidor pode se tornar um gargalo de desempenho para o sistema como um todo. O atraso de sincronização é o tempo exigido para uma viagem de ida e volta: uma mensagem de *liberação* para o servidor, seguida de uma mensagem de *concessão* para o próximo processo a entrar na seção crítica.

Um algoritmo baseado em anel ◊ Uma das maneiras mais simples de constituir a exclusão mútua entre os N processos, sem exigir um processo adicional, é organizá-los em um anel lógico. Isso exige apenas que cada processo p_i tenha um canal de comunicação com o processo seguinte no anel, $P_{(i+1) \bmod N}$. A idéia é que a exclusão seja concedida pela obtenção de um *token*, na forma de uma mensagem passada de processo para processo, em uma única direção – digamos, no sentido horário – em torno do anel. A topologia em anel pode não estar relacionada com as interconexões físicas entre os computadores subjacentes.

Se um processo não pede para entrar na seção crítica ao receber o *token*, então ele encaminha imediatamente o *token* para seu vizinho. Um processo que solicite o *token* espera até recebê-lo, mas o mantém. Para sair da seção crítica, o processo envia o *token* para seu vizinho.

A organização dos processos aparece na Figura 12.3. É fácil verificar que as condições EM1 e EM2 são satisfeitas por esse algoritmo, mas que o *token* não é necessariamente obtido na ordem antes do acontecido. (Lembre-se de que os processos podem trocar mensagens independentemente da rotação do *token*.)

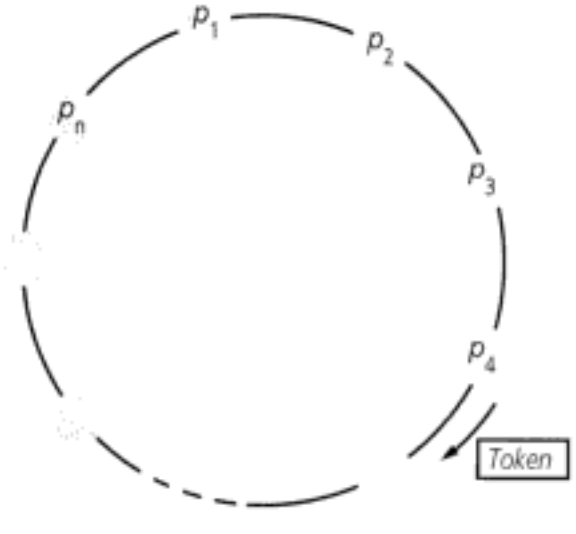


Figura 12.3 Um anel de processos transferindo um *token* de exclusão mútua.

Esse algoritmo consome largura de banda de rede continuamente (exceto quando um processo está dentro da seção crítica): os processos enviam mensagens em torno do anel, mesmo quando nenhum processo solicita entrada na seção crítica. O atraso experimentado por um processo que esteja solicitando a entrada na seção crítica está entre 0 (quando ele acabou de receber o *token*) e N mensagens (quando ele acabou de passar o *token*). Sair da seção crítica exige apenas uma mensagem. O atraso de sincronização entre a saída de um processo da seção crítica e a entrada do processo seguinte está entre 1 e N transmissões de mensagem.

Um algoritmo usando multicast e relógios lógicos ◊ Ricart e Agrawala [1981] desenvolveram um algoritmo para implementar exclusão mútua entre N processos pares, baseado em *multicast*. A idéia básica é que os processos que solicitam a entrada em uma seção crítica difundem seletivamente (*multicast*) uma mensagem de pedido e só podem entrar nela quando todos os outros processos tiverem respondido a essa mensagem. As condições sob as quais um processo responde a um pedido são projetadas de forma a garantir que as condições EM1–EM3 sejam satisfeitas.

Os processos $p_1, p_2, ..., p_N$ apresentam identificadores numéricos distintos. Presume-se que eles possuam canais de comunicação de um para o outro e que cada processo p_i mantenha um relógio de Lamport, atualizado de acordo com as regras RL1 e RL2 da Seção 11.4. As mensagens que solicitam entrada são da forma $<T, p_i>$, onde T é a indicação de tempo do remetente e p_i é o identificador do remetente.

Cada processo registra seu estado de estar fora da seção crítica (RELEASED), querendo entrar (WANTED) ou estar na seção crítica (HELD), em uma variável *state*. O protocolo aparece na Figura 12.4.

Se um processo solicita entrada, e o estado dos outros processos é *RELEASED*, então todos responderão imediatamente ao pedido e o solicitante obterá a entrada. Se algum processo estiver no estado HELD, então esse processo não responderá aos pedidos até que tenha terminado com a seção crítica; portanto, o solicitante não poderá entrar nesse meio tempo. Se dois ou mais processos solicitam a entrada ao mesmo tempo, o pedido do processo que apresentar a indicação de tempo mais baixa será o primeiro a coletar $N - 1$ respostas, garantindo a próxima entrada. Se os pedidos apresentarem indicações de tempo de Lamport iguais, serão ordenados de acordo com os identificadores correspon-

```
On initialization
    state := RELEASED;

To enter the section
    state := WANTED;                                   \
    Multicast request to all processes;                 | Processamento do pedido
    T := request's timestamp;                           | referido aqui
    Wait until (number of replies received = (N – 1));  |
    state := HELD;                                     /

On receipt of a request <Ti, pi> at pj (i ≠ j)
    if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
        queue request from pi without replying;
    else
        reply immediately to pi;
    end if

To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

Figura 12.4 Algoritmo de Ricart e Agrawala.

dentes dos processos. Note que, quando um processo solicita a entrada, ele retarda o processamento dos pedidos de outros processos até que seu próprio pedido tenha sido enviado e ele tiver gravado a indicação de tempo T do pedido. Isso é assim para que os processos tomem decisões consistentes ao processar pedidos.

Esse algoritmo tem a propriedade de segurança EM1. Se fosse possível dois processos, p_i e p_j ($i \neq j$), entrarem na seção crítica ao mesmo tempo, eles teriam que ter respondido um ao outro. Mas, como os pares $<T_i, p_i>$ são totalmente ordenados, isso é impossível. Deixamos para o leitor verificar que o algoritmo também atende aos requisitos EM2 e EM3.

Para ilustrar o algoritmo, considere uma situação envolvendo três processos, p_1, p_2 e p_3, apresentada na Figura 12.5. Vamos supor que p_3 não esteja interessado em entrar na seção crítica e que p_1 e p_2 solicitam a entrada concorrentemente. A indicação de tempo do pedido de p_1 é 41 e a de p_2 é 34. Quando p_3 recebe seus pedidos, responde imediatamente. Quando p_2 recebe o pedido de p_1, verifica que seu próprio pedido tem a indicação de tempo mais baixa e, portanto, não responde, detendo p_1. Entretanto, p_1 verifica que o pedido de p_2 tem uma indicação de tempo mais baixa do que à de seu próprio pedido e, portanto, responde imediatamente. Ao receber essa segunda resposta, p_2 pode entrar na seção crítica. Quando p_2 sair da seção crítica, responderá ao pedido de p_1 e, portanto, garantirá sua entrada.

A obtenção da entrada exige $2(N - 1)$ mensagens nesse algoritmo: $N - 1$ para difundir o pedido por *multicast*, seguido de $N - 1$ respostas. Ou então, se houver suporte de hardware para *multicast*, apenas uma mensagem será exigida para o pedido; assim, o total será de N mensagens. Portanto, esse é um algoritmo mais dispendioso, em termos de consumo de largura de banda, do que os algoritmos que acabamos de descrever. Entretanto, o atraso do cliente no pedido de entrada é, novamente, o tempo de uma viagem de ida e volta (ignorando qualquer atraso acarretado no envio *multicast* da mensagem de pedido).

A vantagem desse algoritmo é que seu atraso de sincronização é apenas o tempo da transmissão de uma mensagem. Os dois algoritmos anteriores acarretavam um atraso de sincronização de uma viagem de ida e volta.

O desempenho do algoritmo pode ser melhorado. Primeiramente, note que o processo que entrou por último na seção crítica, e que não recebeu nenhum outro pedido de entrada de outros processos, ainda passa pelo protocolo, conforme descrito, mesmo que pudesse simplesmente decidir realocá-lo para si mesmo de forma local. Segundo, Ricart e Agrawala refinaram esse protocolo de modo que ele exigisse N mensagens para obter a entrada no pior (e comum) caso, sem suporte de hardware para *multicast*. Isso está descrito em Raynal [1988].

Algoritmo de votação de Maekawa ◊ Maekawa [1985] observou que, para um processo entrar em uma seção crítica, não é necessário que todos os seus pares concedam o acesso. Os processos só precisam obter permissão de *subconjuntos* de seus pares para entrar, desde que os subconjuntos usa-

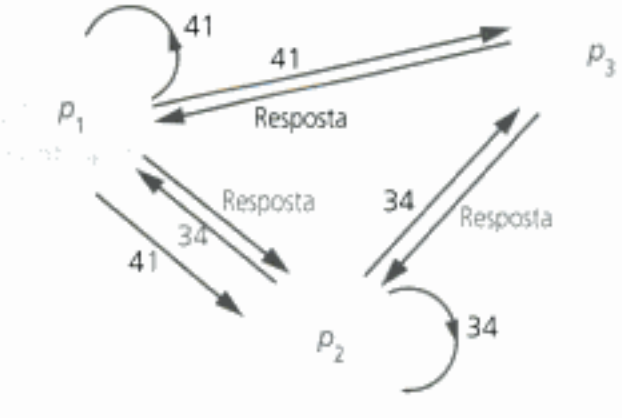


Figura 12.5 Sincronização por *multicast*.

dos por quaisquer dois processos se sobreponham. Podemos considerar os processos como votando um no outro para entrar na seção crítica. Um processo "candidato" deve reunir votos suficientes para entrar. Os processos na intersecção de dois conjuntos de votantes garantem a propriedade de segurança EM1, de que no máximo um processo pode entrar na seção crítica, depositando seus votos para apenas um candidato.

Maekawa associou um conjunto de votação V_i a cada processo p_i ($i = 1, 2 ..., N$), onde $V_i \subseteq \{p_1, p_1 ..., p_N\}$. Os conjuntos V_i são escolhidos de modo que, para todo $i, j = 1, 2 ..., N$:

- $p_i \in V_i$
- $V_i \cap V_j \neq \varnothing$ – existe pelo menos um membro em comum de quaisquer dois conjuntos votantes
- $|V_i| = K$ – para ser imparcial, cada processo tem um conjunto votante de mesmo tamanho
- Cada processo p_j está contido em M dos conjuntos votantes V_j.

Maekawa mostrou que a solução ótima, que minimiza K e permite que os processos obtenham exclusão mútua, tem $K \sim \sqrt{N}$ e $M = K$ (de modo que cada processo está em tantos conjuntos de votação quantos são os elementos em cada um desses conjuntos). Não é simples calcular os conjuntos ótimos R_i. Como uma aproximação, uma maneira simples de derivar conjuntos R_i tais que $|R_i| = 2\sqrt{N}$, é colocar os processos em uma matriz de $\sqrt{N}$ por $\sqrt{N}$ e deixar V_i ser a união da linha e coluna que contêm p_i.

O algoritmo de Maekawa aparece na Figura 12.6. Para obter a entrada na seção crítica, um processo p_i envia mensagens de *pedido* para todos os K membros de V_i (incluindo ele mesmo). p_i não pode entrar na seção crítica até que tenha recebido todas as K mensagens de *resposta*. Quando um processo p_j em V_j recebe a mensagem de *pedido de* p_i, ele envia uma mensagem de *resposta* imediatamente, a não ser que seu estado seja HELD, ou que já tenha respondido ("votado") desde a última vez que recebeu uma mensagem de *liberação*. Caso contrário, ele enfileira a mensagem de pedido (na ordem de sua chegada), mas não responde ainda. Quando um processo recebe uma mensagem de *liberação*, ele remove o nodo cabeça de sua fila de pedidos pendentes (se a fila não estiver vazia) e envia uma mensagem de *resposta* (um "voto") em retorno a ela. Para sair da seção crítica, p_i envia mensagens de *liberação* para todos os K membros de V_i (incluindo ele mesmo).

Esse algoritmo obtém a propriedade da segurança EM1. Se fosse possível que dois processos p_i e p_j entrassem na seção crítica ao mesmo tempo, então os processos em $V_i \cap V_j \neq \varnothing$ teriam que ter votado em ambos. Mas o algoritmo permite que um processo deposite no máximo um voto entre sucessivos recebimentos de uma mensagem de *liberação* – portanto, essa situação é impossível.

Infelizmente, o algoritmo é propenso a impasses. Considere três processos p_1, p_2 e p_3 com $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$ e $V_3 = \{p_3, p_1\}$. Se os três processos solicitam a entrada na seção crítica, então é possível que p_1 responda para si mesmo e detenha p_1, que p_2 responda para si mesmo e detenha p_3 e que p_3 responda para si mesmo e detenha p_1. Cada processo recebeu uma de duas respostas e nenhum pode prosseguir.

O algoritmo pode ser adaptado [Saunders 1987] de modo que se torne livre de impasses. No protocolo adaptado, os processos enfileiram os pedidos pendentes na ordem antes do acontecido, de modo que o requisito EM3 também é satisfeito.

A utilização de largura de banda do algoritmo é de $2\sqrt{N}$ mensagens por entrada na seção crítica e de $\sqrt{N}$ mensagens por saída (supondo que não exista nenhum recurso de hardware para *multicast*). O total de $3\sqrt{N}$ é superior às $2(N - 1)$ mensagens exigidas pelo algoritmo de Ricart e Agrawala, se $N > 4$. O atraso de cliente é o mesmo do algoritmo de Ricart e Agrawala, mas o atraso de sincronização é pior: um tempo de viagem de ida e volta, em vez de um único tempo de transmissão de mensagem.

Tolerância a falhas ◊ Os principais pontos a considerar na avaliação dos algoritmos anteriores, com relação à tolerância a falhas, são:

- O que acontece quando mensagens são perdidas?
- O que acontece quando um processo falha?

Nenhum dos algoritmos que descrevemos toleraria a perda de mensagens, caso os canais fossem não confiáveis. O algoritmo baseado em anel não pode tolerar uma falha por colapso de um único

```
Na inicialização
    state := RELEASED;
    voted := FALSE;

Para p_i entrar na seção crítica
    state := WANTED;
    Envia o pedido por multicast para todos os processos em V_i;
    Espera até (número de respostas recebidas = K);
    state := HELD;

No recebimento de um pedido de p_i em p_j
    if (state = HELD or voted = TRUE)
    then
            enfileira o pedido de p_i sem responder;

    else

            envia resposta para p_i;
            voted := TRUE;

    end if

Para p_i sair da seção crítica
    state := RELEASED;
    Envia a liberação via multicast para todos os processos em V_i;

No recebimento de uma liberação de p_i em p_j
    if (fila de pedidos não estiver vazia)
    then

            remove cabeça da fila – digamos, p_k;
            envia resposta para p_k;
            voted := TRUE;

    else

            voted := FALSE;

    end if
```

Figura 12.6 Algoritmo de Maekawa.

processo. Como se vê, o algoritmo de Maekawa pode tolerar algumas falhas de processo por colapso: se um processo falho não estiver em um conjunto votante que seja exigido, então sua falha não afetará os outros processos. O algoritmo do servidor central pode tolerar a falha por colapso de um processo cliente que não contenha, nem tenha solicitado, o *token*. O algoritmo de Ricart e Agrawala, conforme o descrevemos, pode ser adaptado para tolerar a falha por colapso de tal processo, fazendo-o conceder todos os pedidos implicitamente.

Convidamos o leitor a considerar como adaptar os algoritmos para tolerar falhas, supondo que um detector de falha confiável esteja disponível. Mesmo com um detector de falha confiável, é necessário cuidado para permitir a existência de falhas em qualquer ponto (inclusive durante um procedimento de recuperação) e para reconstruir o estado dos processos após uma falha ter sido detectada. Por exemplo, no algoritmo do servidor central, se o servidor falhar, deverá ser estabelecido se ele manteve o *token*, ou se um dos processos clientes o mantém.

Examinaremos o problema geral de como os processos devem coordenar suas ações na presença de falhas, na Seção 12.5.

12.3 Eleições

Um algoritmo para escolher um único processo para desempenhar uma função em particular é chamado de *algoritmo de eleição*. Por exemplo, em uma variante de nosso algoritmo do servidor central para exclusão mútua, o servidor é escolhido dentre os processos p_i, $i = 1, 2, \ldots, N$ que precisam usar a seção crítica. Um algoritmo de eleição é necessário para escolher qual dos processos desempenhará a função de servidor. É fundamental que todos os processos concordem com a escolha. Depois disso, se o processo que desempenha a função de servidor quiser se retirar, outra eleição será exigida para escolher um substituto.

Dizemos que um processo *convoca a eleição* se ele faz uma ação que inicia uma execução em particular do algoritmo de eleição. Um processo individual não convoca mais do que uma eleição por vez, mas, em princípio, os N processos poderiam convocar N eleições concorrentes. A qualquer momento, um processo p_i *é participante* – significando que ele está envolvido em alguma execução do algoritmo de eleição – ou *não participante* – significando que correntemente ele não está envolvido em nenhuma eleição.

Um requisito importante é que a escolha do processo eleito seja única, mesmo que vários processos convoquem eleições concorrentemente. Por exemplo, dois processos poderiam decidir, independentemente, que um processo coordenador falhou e ambos convocam eleições.

Sem perda de generalidade, exigimos que o processo eleito seja escolhido como aquele com o maior identificador. O identificador pode ser qualquer valor útil, desde que os identificadores sejam exclusivos e totalmente ordenados. Por exemplo, poderíamos eleger o processo com a menor carga computacional, fazendo cada processo usar <1/*carga*, *i*> como identificador, onde *carga* > 0, e o índice de processo *i* seria usado para ordenar identificadores com a mesma carga.

Cada processo p_i ($i = 1, 2\ldots, N$) tem uma variável $elected_i$, que conterá o identificador do processo eleito. Quando o processo se torna participante de uma eleição pela primeira vez, ele configura essa variável com o valor especial '$\perp$', para denotar que ela ainda não está definida.

Nossos requisitos são que, durante qualquer execução em particular do algoritmo:

E1: (segurança) Um processo participante p_i tem $elected_i = \perp$ ou $elected_i = P$, onde P é escolhido como o processo não defeituoso com o maior identificador no final da execução.

E2: (subsistência) Todos os processos p_i participam e configuram $elected_i \neq \perp$ – ou falham.

Note que podem existir processos p_i que ainda não sejam participantes, os quais registram em $elected_i$ o identificador do processo eleito anterior.

Medimos o desempenho de um algoritmo de eleição por sua utilização de largura de banda de rede total (que é proporcional ao número total de mensagens enviadas) e pelo *tempo do ciclo de rotação* do algoritmo: o número de tempos de transmissão de mensagem dispostos em série, entre o início e o término de uma execução.

Um algoritmo de eleição baseado em anel ◊ Fornecemos o algoritmo de Chang e Roberts [1979], que é conveniente para um conjunto de processos organizados em um anel lógico. Cada processo p_i tem um canal de comunicação para o processo seguinte no anel, $p_{(i+1) \bmod N}$ e todas as mensagens são enviadas no sentido horário em torno do anel. Supomos que não ocorrem falhas e que o sistema é assíncrono. O objetivo desse algoritmo é eleger um único processo, chamado de *coordenador*, que é aquele com o maior identificador.

Inicialmente, cada processo é marcado como *não participante* de uma eleição. Qualquer processo pode iniciar uma eleição. Ele prossegue marcando a si mesmo como *participante*, colocando seu identificador em uma mensagem de *eleição* e enviando-a para seu vizinho no sentido horário.

Quando um processo recebe uma mensagem de *eleição*, ele compara o identificador presente na mensagem com o seu próprio. Se o identificador que chegou é maior, então ele encaminha a mensagem para seu vizinho. Se o identificador que chegou é menor e o receptor não é *participante*, então ele substitui por seu próprio identificador na mensagem e a encaminha; mas não encaminha a mensagem se já for *participante*. Em qualquer caso, no encaminhamento de uma mensagem de *eleição*, o processo marca a si mesmo como *participante*.

Entretanto, se o identificador recebido for o do próprio receptor, então o identificador desse processo deve ser o maior e se tornará o coordenador. Mais uma vez, o coordenador marca a si mesmo como *não participante* e envia uma mensagem *elected* para seu vizinho, anunciando sua eleição e incluindo sua identidade.

Quando um processo p_i recebe uma mensagem *elected*, ele marca a si mesmo como *não participante*, configura sua variável $elected_i$ com o identificador presente na mensagem e, a não ser que seja o novo coordenador, encaminha a mensagem para seu vizinho.

É fácil ver que a condição E1 é satisfeita. Todos os identificadores são comparados, pois um processo deve receber seu próprio identificador de volta, antes de enviar uma mensagem *elected*. Para quaisquer dois processos, aquele com o identificador maior não passará o identificador do outro. Portanto, é impossível que ambos recebam seus próprios identificadores de volta.

A condição E2 resulta imediatamente da garantia das passagens de mensagem no anel (não existem falhas). Observe como os estados *não participante* e *participante* são usados, de modo que as mensagens que surgem quando outro processo inicia uma eleição ao mesmo tempo, são extintas assim que possível e sempre antes do resultado do vencedor da eleição ter sido anunciado.

Se apenas um processo inicia uma eleição, então o caso de pior desempenho se dá quando seu vizinho no sentido anti-horário tem o identificador mais alto. Então, um total de $N - 1$ mensagens é exigido para chegar a esse vizinho, o qual não anunciará sua eleição até que seu identificador tenha completado outro circuito, exigindo mais N mensagens. A mensagem *elected* é então enviada N vezes, totalizando $3N - 1$ mensagens. O tempo do ciclo de rotação também é de $3N - 1$, pois essas mensagens são enviadas seqüencialmente.

Um exemplo de eleição baseada em anel em andamento aparece na Figura 12.7. A mensagem de *eleição* contém correntemente 24, mas o processo 28 substituirá isso pelo seu identificador, quando a mensagem chegar a ele.

Embora o algoritmo baseado em anel seja útil para se entender as propriedades dos algoritmos de eleição em geral, o fato de ele não tolerar falhas o torna limitado quanto ao seu valor prático. Entretanto, com um detector de falha confiável é possível, em princípio, reconstituir o anel quando um processo falha.

O algoritmo valentão (*bully*) ◊ O algoritmo valentão [Garcia-Molina 1982] permite que os processos falhem durante uma eleição, embora presuma que a distribuição de mensagens entre os processos seja confiável. Ao contrário do algoritmo baseado em anel, este algoritmo presume que o sistema é síncrono: ele usa tempos limites para detectar uma falha de processo. Outra diferença é que o algoritmo baseado em anel presumia que os processos tinham conhecimento *a priori* mínimo uns dos ou-

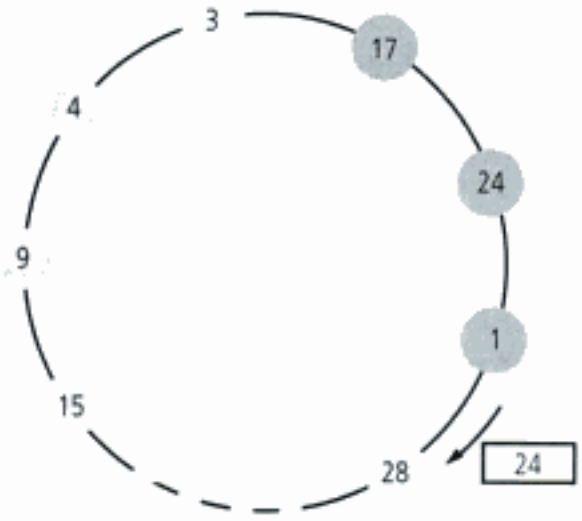


Figura 12.7 Uma eleição baseada em anel (em andamento).
Nota: A eleição foi iniciada pelo processo 17. O identificador de processo mais alto encontrado até aqui é 24. Os processos participantes aparecem com fundo cinza mais escuro.

tros: cada um sabia apenas como se comunicar com seu vizinho, e nenhum conhecia os identificadores dos outros processos. Por outro lado, o algoritmo valentão presume que cada processo sabe quais processos têm identificadores mais altos e que pode se comunicar com todos esses processos.

Existem três tipos de mensagem nesse algoritmo. Uma mensagem de *eleição* é enviada para anunciar uma eleição; uma mensagem de *resposta* é enviada em retorno a uma mensagem de eleição; uma mensagem de *coordenador* é enviada para anunciar a identidade do processo eleito – o novo coordenador. Um processo inicia uma eleição quando observa, por meio dos tempos limites, que o coordenador falhou. Vários processos podem descobrir isso simultaneamente.

Como o sistema é síncrono, podemos construir um detector de falha confiável. Há um atraso de transmissão de mensagem máximo T_{trans} e um atraso máximo $T_{process}$ para processar uma mensagem. Portanto, podemos calcular um tempo $T = 2T_{trans} + T_{process}$, que é um limite superior para o tempo total decorrido desde o envio de uma mensagem até o outro processo receber uma resposta. Se nenhuma resposta chegar dentro do tempo T, o detector de falha local poderá relatar que o destinatário do pedido pretendido falhou.

O processo que sabe que possui o identificador mais alto pode eleger a si mesmo como coordenador simplesmente enviando uma mensagem de *coordenador* para todos os processos com identificadores mais baixos. Por outro lado, um processo com um identificador mais baixo inicia uma eleição enviando uma mensagem de *eleição* para os processos que têm identificador mais alto e esperando uma mensagem de *resposta* em retorno. Se nenhuma resposta chegar dentro do tempo T, o processo se considerará o coordenador e enviará uma mensagem de *coordenador* para todos os processos com identificadores mais baixos, anunciando isso. Caso contrário, o processo esperará por mais um período T', que uma mensagem de *coordenador* chegue do novo coordenador. Se nenhuma resposta chegar, ele iniciará outra eleição.

Se um processo p_i recebe uma mensagem de *coordenador*, ele configura sua variável $elected_i$ com o identificador do coordenador contido dentro dela e trata esse processo como coordenador.

Se um processo recebe uma mensagem de *eleição*, ele envia de volta uma mensagem de *resposta* e inicia outra eleição – a não ser que já tenha iniciado uma.

Quando é iniciado um processo para substituir um processo falho, ele inicia uma eleição. Se tiver o identificador de processo mais alto, decidirá que é o coordenador e anunciará isso para os outros processos. Assim, ele se tornará o coordenador, mesmo que o coordenador corrente esteja funcionando. É por isso que o algoritmo é chamado de "valentão".

O funcionamento do algoritmo aparece na Figura 12.8. Existem quatro processos $p_1 - p_4$. O processo p_1 detecta a falha do coordenador p_4 e anuncia uma eleição (estágio 1 na figura). Ao receber uma mensagem de *eleição* de p_1, os processos p_2 e p_3 enviam mensagens de *resposta* para p_1 e iniciam suas próprias eleições; p_3 envia uma mensagem de *resposta* para p_2, mas p_3 não recebe nenhuma mensagem de *resposta* do processo falho p_4 (estágio 2). Portanto, ele decide que é o coordenador. Mas antes que possa enviar a mensagem de *coordenador*, ele também falha (estágio 3). Quando o período do tempo limite T' de p_1 expira (o qual presumimos que ocorra antes que o tempo limite de p_2 expire), ele deduz a ausência de uma mensagem de *coordenador* e inicia outra eleição. Finalmente, p_2 é eleito coordenador (estágio 4).

Esse algoritmo satisfaz claramente a condição de subsistência E2, pela suposição do envio de mensagem confiável. E se nenhum processo for substituído, então o algoritmo satisfaz a condição E1. É impossível dois processos decidirem que são o coordenador, pois o processo com o identificador mais baixo descobrirá que o outro existe e o acatará.

Mas *não* é garantido que o algoritmo satisfaça a condição de segurança E1, caso processos que tenham falhado sejam substituídos por processos com o mesmo identificador. Um processo que substitui um processo falho p pode decidir que tem o identificador mais alto, assim como outro processo (que detectou a falha de p) decidiu que possui o identificador mais alto. Os dois processos se anunciarão como coordenadores, simultaneamente. Infelizmente, não há garantia da ordem do envio das mensagens e os destinatários dessas mensagens poderão chegar a diferentes conclusões sobre qual é o processo coordenador.

Além disso, a condição E1 pode ser violada, se os valores de tempo limite pressupostos se mostrarem imprecisos – isto é, se o detector de falha dos processos não for confiável.

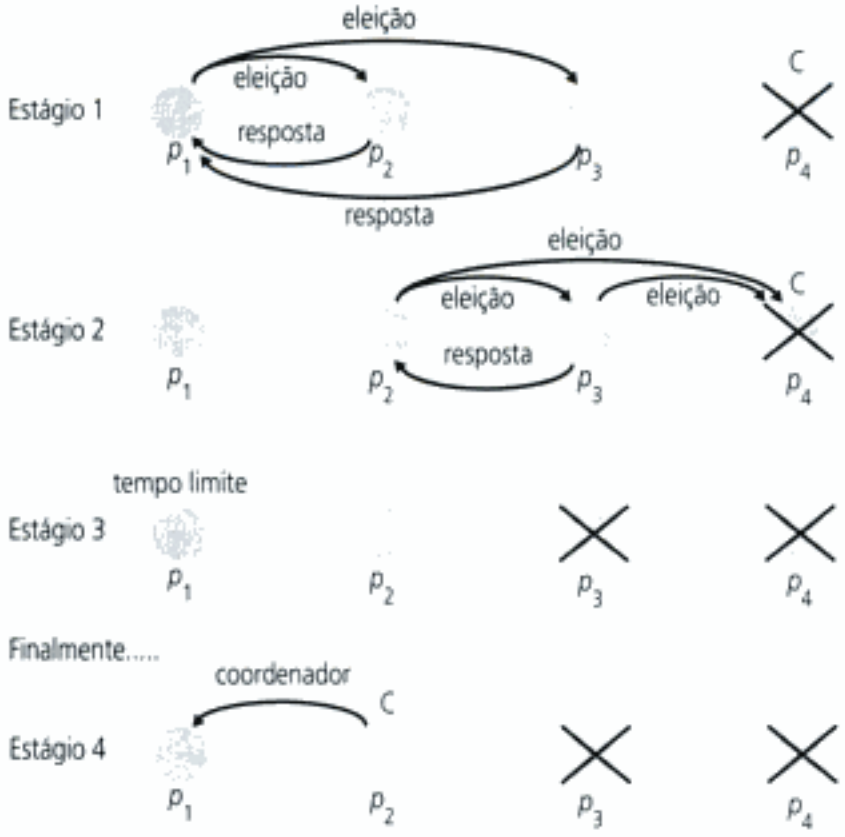


Figura 12.8 O algoritmo valentão.
A eleição do coordenador p_2, após a falha de p_4 e, depois, de p_3.

Pegando o exemplo que acabamos de dar, suponha que p_3 não tivesse falhado, mas estava executando de forma extraordinariamente lenta (isto é, a suposição de que o sistema é síncrono está incorreta), ou que p_3 tivesse falhado, mas então foi substituído. Assim como p_2 envia sua mensagem de *coordenador*, p_3 (ou seu substituto) faz o mesmo. p_2 recebe *a* mensagem de *coordenador* de p_3 após ter enviado a sua própria e, portanto, configura $elected_2 = p_3$. Devido aos atrasos de transmissão de mensagem variáveis, p_1 recebe a mensagem de *coordenador* de p_2 após a de p_3 e, assim, finalmente configura $elected_1 = p_2$. A condição E1 foi violada.

Com relação ao desempenho do algoritmo, no melhor caso, o processo com o segundo identificador mais alto nota a falha do coordenador. Então, ele pode se eleger imediatamente e enviar $N - 2$ mensagens de coordenador. O tempo do ciclo de rotação é o de uma mensagem. O algoritmo valentão exige $O(N^2)$ mensagens, no pior caso – isto é, quando o processo com o menor identificador detecta primeiro a falha do coordenador. Nesse caso, $N - 1$ processos em conjunto iniciam eleições, cada um enviando mensagens para os processos com identificadores mais altos.

12.4 Comunicação *multicast*

A Seção 4.5.1 descreveu o *multicast* IP, que é uma implementação de comunicação em grupo. A comunicação em grupo, ou por *multicast*, exige coordenação e acordo. O objetivo é que cada processo de um grupo de processos receba cópias das mensagens enviadas para o grupo, freqüentemente com garantias de distribuição. As garantias incluem o acordo sobre o conjunto de mensagens que cada processo do grupo deve receber e a ordem de entrega pelos membros do grupo.

Os sistemas de comunicação em grupo são extremamente sofisticados. Mesmo o *multicast* IP, que fornece garantias de distribuição mínimas, exige um trabalho de engenharia importante. O tempo e a eficiência da largura de banda são preocupações sérias e desafiadoras, mesmo para grupos de proces-

sos estáticos. Os problemas são multiplicados quando os processos podem entrar e sair dos grupos em momentos arbitrários.

Estudaremos aqui a comunicação *multicast* para grupos de processos cuja participação como membro é conhecida. O Capítulo 15 expandirá nosso estudo para a comunicação de grupo completa, incluindo o gerenciamento de grupos que variam dinamicamente.

A comunicação *multicast* tem sido o assunto de muitos projetos, incluindo o V-system [Cheriton e Zwaenepoel 1985], Chorus [Rozier *et al.* 1988], Amoeba [Kaashoek *et al.* 1989, Kaashoek e Tanenbaum 1991], Trans/Total [Melliar-Smith *et al.* 1990], Delta-4 [Powell 1991], Isis [Birman 1993], Horus [van Renesse *et al.* 1996], Totem [Moser *et al.* 1996] e Transis [Dolev e Malki 1996] – e citaremos outros trabalhos notáveis no decorrer desta seção.

A característica básica da comunicação *multicast* é que um processo executa apenas uma operação *multicast* para enviar uma mensagem para cada processo de um grupo de processos (em Java, essa operação é *aSocket.send*(*aMessage*)), em vez de executar várias operações *send* para processos individuais. A comunicação com *todos* os processos do sistema, em oposição a um subgrupo deles, é conhecida como *broadcast*.

O uso de uma única operação *multicast*, em vez de várias operações *send*, significa muito mais do que uma conveniência para o programador. Ela permite que a implementação seja eficiente e possibilita fornecer garantias de entrega mais fortes do que seriam possíveis de outra forma.

Eficiência: a informação de que a mesma mensagem vai ser enviada para todos os processos de um grupo permite que a implementação seja eficiente em sua utilização da largura de banda. Podem ser tomadas medidas para enviar a mensagem não mais do que uma vez por qualquer enlace de comunicação, por meio do seu envio para uma árvore de distribuição; e pode-se usar suporte de hardware de rede para *multicast*, onde isso estiver disponível. A implementação também pode minimizar o tempo total gasto para enviar a mensagem para todos os destinos, em vez de transmiti-la separadamente, e em série.

Para ver essas vantagens, compare a utilização da largura de banda e o tempo de transmissão total gasto ao enviar a mesma mensagem de um computador em Londres para dois computadores na mesma rede Ethernet, em Palo Alto, (a) por meio de dois envios de UDP separados e (b) por meio de uma única operação de *multicast* IP. No primeiro caso, duas cópias das mensagens são enviadas independentemente, e a segunda é retardada pela primeira. No segundo caso, um conjunto de roteadores com capacidade *multicast* encaminha uma única cópia da mensagem de Londres para um roteador na rede local de destino. Então, o roteador final usa um *multicast* em hardware (fornecida pela rede Ethernet) para enviar a mensagem para os destinos, em vez de enviá-la duas vezes.

Garantias de entrega: se um processo executa várias operações *send* independentes para processos individuais, então não há como a implementação dar garantias de entrega que afetem o grupo de processos como um todo. Se o remetente falhar na metade do processo de envio, alguns membros do grupo poderão receber a mensagem, enquanto outros não. E a ordem relativa das duas mensagens enviadas para quaisquer dois membros do grupo é indefinida. Na verdade, no caso particular de *multicast* IP nenhuma garantia de ordem, ou confiabilidade, é oferecida. Mas podem ser dadas garantias de *multicast* mais fortes, e em breve definiremos algumas.

Modelo de sistema ◊ O sistema contém um conjunto de processos, os quais podem se comunicar com confiabilidade por meio de canais um para um. Como antes, os processos podem falhar apenas por colapso.

Os processos são membros de grupos, os quais são os destinos das mensagens enviadas com a operação de *multicast*. Geralmente é útil permitir que os processos sejam membros de vários grupos simultaneamente – por exemplo, para permitir que processos recebam informações de várias fontes, entrando em vários grupos. Mas para simplificar nossa discussão sobre as propriedades de ordenação, às vezes restringiremos os processos de modo a serem membros de no máximo um grupo por vez.

A operação *multicast*(*g*, *m*) envia a mensagem *m* para todos os membros do grupo *g* (processos). Correspondentemente, existe uma operação *deliver*(*m*) que distribui (entrega) uma mensagem recebida por *multicast* para o processo que a executa. Usamos o termo *distribuir* ou *entregar*, em vez de *receber*, para tornar claro que uma mensagem *multicast* nem sempre é entregue para a camada de

aplicativo dentro do processo, assim que é recebida no nó do processo. Isso será explicado quando discutirmos a semântica da distribuição por *multicast*, em breve.

Toda mensagem *m* transporta o identificador exclusivo do processo *sender*(*m*) que a enviou e o identificador de grupo de destino exclusivo *group*(*m*). Supomos que os processos não mentem sobre a origem ou destinos das mensagens.

Diz-se que um grupo é *fechado* se apenas os membros do grupo podem fazer *multicast* para ele (Figura 12.9). Um processo em um grupo fechado envia para si mesmo toda mensagem que difunde seletivamente (*multicast*) para o grupo. Um grupo é *aberto* se processos de fora podem enviar mensagens para ele. (As categorias "aberto" e "fechado" também se aplicam às listas de distribuição de e-mail, com significados análogos.) Os grupos fechados de processos são úteis, por exemplo, para servidores em cooperação para enviar uns aos outros mensagens que apenas eles devem receber. Os grupos abertos são úteis, por exemplo, para enviar eventos para grupos de processos interessados.

Alguns algoritmos presumem que os grupos são fechados. O mesmo efeito de abertura pode ser obtido com um grupo fechado, escolhendo-se um membro do grupo e enviando-se a ele uma mensagem (um para um) para difundir seletivamente para seu grupo. Rodrigues *et al.* [1998] discutem o *multicast* para grupos abertos.

12.4.1 *Multicast* básico

É interessante termos à nossa disposição uma primitiva de *multicast* básico que garanta, ao contrário do *multicast* IP, que um processo correto finalmente distribuirá a mensagem, desde que o difusor não falhe. Chamamos a primitiva de *B-multicast* e sua primitiva de distribuição básica correspondente de *B-deliver*. Permitimos que os processos pertençam a vários grupos e cada mensagem é destinada a algum grupo em particular.

Uma maneira simples de implementar *B-multicast* é usando uma operação *send* de um para um confiável, como segue:

Para *B-multicast*(*g*, *m*): para cada processo $p \in g$, *send*(*p*, *m*);

Em *receive*(*m*) em *p*: *B-deliver*(*m*) em *p*.

A implementação pode usar *threads* para executar as operações *send* concorrentemente, em uma tentativa de reduzir o tempo total gasto para distribuir a mensagem. Infelizmente, tal implementação é propensa a sofrer a conhecida *explosão de confirmações*, caso o número de processos seja grande. Os sinais de confirmação, enviados como parte da operação *send* confiável, estão sujeitos a chegar de muitos processos quase ao mesmo tempo. Os buffers dos processos serão consumidos rapidamente e

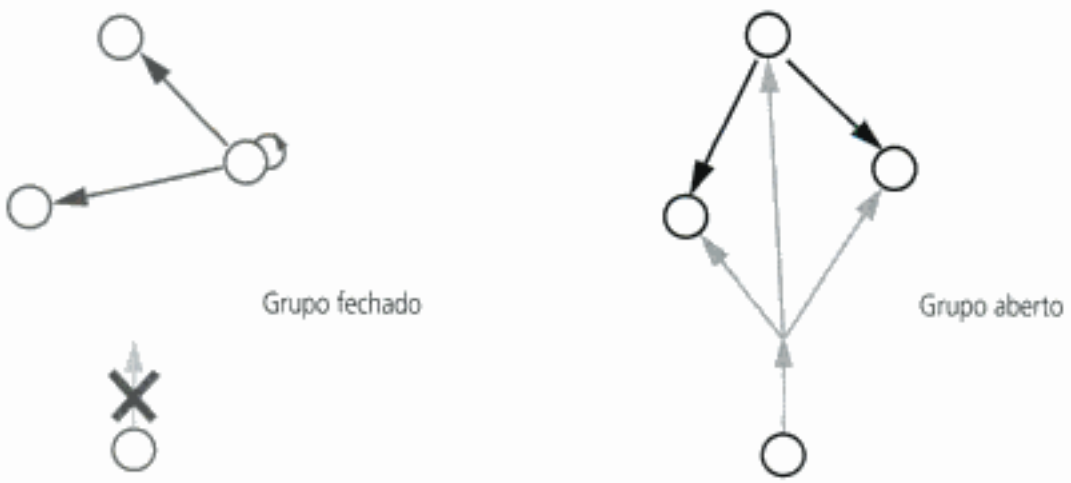


Figura 12.9 Grupos abertos e fechados.

os sinais de confirmação podem ser perdidos. Portanto, ele retransmitirá a mensagem, acarretando ainda mais sinais de confirmações e mais desperdício de largura de banda de rede. Um serviço *multicast* básico mais prático pode ser construído, usando-se *multicast* IP, e convidamos o leitor a mostrar isso.

12.4.2 *Multicast* confiável

A Seção 2.3.2 definiu os canais de comunicação um para um confiáveis entre pares de processos. A propriedade de segurança exigida é chamada de *integridade* – a de que qualquer mensagem entregue é idêntica àquela que foi enviada e que nenhuma mensagem é entregue duas vezes. A propriedade de subsistência exigida é chamada de *validade* – a de que qualquer mensagem é finalmente entregue para o destino, se estiver correta.

Seguindo Hadzilacos e Toueg [1994] e Chandra e Toueg [1996], definiremos agora o *multicast confiável*, com as operações correspondentes *R-multicast* e *R-deliver (R de Reliable)*. Claramente, propriedades análogas à integridade e à validade são altamente desejáveis na distribuição por *multicast* confiável. Mas acrescentamos outra: o requisito de que todos os processos corretos do grupo devem receber uma mensagem, caso qualquer um deles receba. É importante perceber que essa não é uma propriedade do algoritmo *B-multicast*, que é baseado em uma operação *send* de um para um confiável. O remetente pode falhar em qualquer ponto, enquanto *B-multicast* prossegue; portanto, alguns processos podem distribuir uma mensagem, enquanto outros não.

O *multicast* confiável é aquele que satisfaz as seguintes propriedades (explicaremos as propriedades após declará-las).

> *Integridade*: um processo correto p entrega uma mensagem m no máximo uma vez. Além disso, $p \in group(m)$ e m foi fornecida para uma operação *multicast* por $sender(m)$. (Assim como acontece com a comunicação de um para um, as mensagens sempre podem ser diferenciadas por um número de seqüência relativo aos seus remetentes.)
>
> *Validade*: se um processo correto executa um *multicast* da mensagem m, então, ele finalmente entregará m.
>
> *Acordo*: se um processo correto entrega a mensagem m, então todos os outros processos corretos em $group(m)$ finalmente entregarão m.

A propriedade da integridade é análoga à da comunicação um para um confiável. A propriedade da validade garante a subsistência do remetente. Essa propriedade pode parecer incomum, pois é assimétrica (ela menciona apenas um processo em particular). Mas observe que, juntos, a validade e o acordo significam um requisito de subsistência global: se um processo (o remetente) finalmente entregar uma mensagem m, então, como os processos corretos concordam com o conjunto de mensagens que entregam, segue-se que m finalmente será entregue para todos os membros corretos do grupo.

A vantagem de expressar a condição de validade em termos de auto-entrega é a simplicidade. O que necessitamos é que a mensagem seja finalmente entregue por *algum* membro correto do grupo.

A condição do acordo está relacionada à atomicidade, a propriedade do "tudo ou nada", aplicada a entrega de mensagens para um grupo. Se um processo que envia por *multicast* uma mensagem falha antes de tê-la entregue, então é possível que a mensagem não seja entregue para nenhum processo do grupo; mas se ela for entregue para algum processo correto, então todos os outros processos corretos a entregarão. Muitos artigos na literatura usam o termo "atômico" para incluir uma condição de ordenação total; definiremos isso em breve.

Implementando *multicast* confiável por meio de *B-multicast* ◊ A Figura 12.10 fornece um algoritmo de *multicast* confiável, com primitivas *R-multicast* e *R-deliver*, o qual permite aos processos pertencerem a vários grupos fechados simultaneamente. Para enviar uma mensagem com *R-multicast*, um processo a envia com *B-multicast* para os processos no grupo de destino (incluindo ele mesmo). Quando a mensagem é entregue com *B-deliver*, o destinatário, por sua vez, a entrega com *B-multicast* para o grupo (se ele não for o remetente original) e depois a entrega com *R-deliver*. Como uma mensagem pode chegar mais de uma vez, as duplicatas são detectadas e não enviadas.

```
Na inicialização
    Received:= { };
Para o processo p enviar a mensagem m com R-multicast para o grupo g
    B-multicast(g, m);              // p ∈ g é incluído como destino
Em B-deliver(m) no processo q com g = group(m)
    if (m ∉ Received)
    then
            Received := Received ∪ {m};
            if (q ≠ p) then B-multicast(g, m); end if
            R-deliver m;
    end if
```

Figura 12.10 Algoritmo de *multicast* confiável.

Esse algoritmo claramente satisfaz a validade, pois um processo correto finalmente entregará a mensagem para si mesmo com *B-deliver*. Pela propriedade da integridade dos canais de comunicação subjacente usados em *B-multicast*, o algoritmo também satisfaz a propriedade da integridade.

O acordo resulta do fato de que todo processo correto envia a mensagem com *B-multicast* para os outros processos, após tê-la entregue com *B-deliver*. Se um processo correto não entregar a mensagem com *R-deliver*, então só pode ser porque ele nunca o fez com *B-deliver*. Isso, por sua vez, só pode ser porque nenhum outro processo correto a entregou com *B-deliver*; portanto, nenhum a enviará com *R-deliver*.

O algoritmo de *multicast* confiável que descrevemos é correto em um sistema assíncrono, pois não fizemos suposições de temporização. Mas o algoritmo é ineficiente para propósitos práticos. Cada mensagem é enviada $|g|$ vezes para cada processo.

Multicast confiável por meio de _multicast_ IP ◊ Uma realização alternativa de *R-multicast* é usar uma combinação *multicast* IP, confirmações "de carona" (isto é, confirmações anexadas em outras mensagens) e confirmações negativas. Esse protocolo *R-multicast* é baseado na observação de que a comunicação por *multicast* IP é freqüentemente bem-sucedida. No protocolo, os processos não enviam mensagens de confirmação separadas; em vez disso, elas colocam as confirmações "de carona" das mensagens que enviam para o grupo. Os processos enviam uma mensagem de resposta separada apenas quando detectam que perderam uma mensagem. Uma resposta indicando a ausência de uma mensagem esperada é conhecida como *confirmação negativa*.

A descrição presume que os grupos são fechados. Cada processo p mantém um número de seqüência S_g^p para cada grupo g ao qual pertence. O número de seqüência inicialmente é zero. Cada processo também grava R_g^q, o número de seqüência da última mensagem que recebeu do processo q *enviada para o* grupo g.

Para p enviar uma mensagem com *R-multicast* para o grupo g, ele coloca o valor S_g^p e confirmação "de carona" na mensagem, da forma <q, R_g^q>. Uma confirmação informa, para um remetente q, o número de seqüência da mensagem mais recente de q, destinada a g, que p entregou desde que difundiu um *multicast*. Então, o emissor p envia a mensagem por *multicast* IP para g, com seus valores "de carona", e incrementa S_g^p por um.

Os valores "de carona" em uma mensagem *multicast* permitem que os destinatários saibam sobre as mensagens que não receberam. Um processo entrega uma mensagem com *R-deliver*, destinada a g, contendo o número de seqüência S de p, se e somente se $S = R_g^q + 1$, e incrementa R_g^q por um, imediatamente após a distribuição. Se uma mensagem recebida tem $S \leq R_g^q$, então r enviou a mensagem antes e a descarta. Se $S > R_g^q + 1$ ou se $R > R_g^q$ com uma confirmação incluída <q, R>, então existe uma ou mais mensagens ainda não recebidas (e que provavelmente foram eliminadas, no primeiro caso). Ele mantém toda mensagem para a qual $S > R_g^q + 1$, em uma *fila de espera* (Figura 12.11) – tais filas são freqüentemente usadas para satisfazer garantias de distribuição de mensagem. Ele solicita as mensagens ausentes enviando confirmações negativas para o remetente original, ou para um processo q a partir do qual recebeu uma confirmação <q, R_g^q>, com R_g^q não menor do que o número de seqüência exigido.

Hidden page

cliente que acesse um servidor imediatamente antes dele falhar poderá observar uma atualização que nenhum outro servidor processará.

É interessante notar que, se invertermos as linhas "*R-deliver m*" e "*if* ($q \neq p$) *then B-multicast*(g, m); *end if*" na Figura 12.10, o algoritmo resultante não satisfará o acordo uniforme.

Assim como existe uma versão uniforme do acordo, também existem versões uniformes de qualquer propriedade de *multicast*, incluindo validade e integridade e as propriedades de ordenação que estamos para definir.

12.4.3 *Multicast* ordenado

O algoritmo *multicast* básico da Seção 12.4.1 entrega mensagens para processos em uma ordem arbitrária, devido aos atrasos arbitrários nas operações de envio de um para um subjacentes. Essa falta de garantia de ordem não é satisfatória para muitas aplicações. Por exemplo, em uma usina nuclear pode ser importante que os eventos que signifiquem ameaças às condições de segurança e os eventos que signifiquem ações de unidades de controle sejam observados na mesma ordem por todos os processos do sistema.

Os requisitos de ordenação comuns são: ordem total, ordem causal, ordem FIFO e as misturas total-causal e total-FIFO. Para simplificarmos nossa discussão, definiremos essas ordenações sob a suposição de que todo processo pertence no máximo a um grupo. Posteriormente, discutiremos as implicações resultantes de permitir que os grupos se sobreponham.

Ordem FIFO: se um processo correto executa *multicast*(g, m) e depois *multicast*(g, m'), então todo processo correto que entregar m' entregará m antes de m'.

Ordem causal: se *multicast*(g, m) $\rightarrow$ *multicast*(g, m'), onde $\rightarrow$ é a relação antes do acontecido induzida apenas pelas mensagens enviadas entre os membros de g, então todo processo correto que entregar m' entregará m antes de m'.

Ordem total: se um processo correto entregar a mensagem m antes de entregar m', então qualquer outro processo correto que entregue m' entregará m antes de m'.

A ordem causal implica na ordem FIFO, pois quaisquer dois *multicasts* feitos pelo mesmo processo estão sujeitos à relação antes do acontecido. Note que a ordem FIFO e a ordem causal são ordenações apenas parciais: em geral, nem todas as mensagens são enviadas pelo mesmo processo; analogamente, alguns *multicasts* são concorrentes (não ordenados pela relação antes do acontecido).

A Figura 12.12 ilustra as ordenações para o caso de três processos. Uma inspeção detalhada da figura mostra que as mensagens totalmente ordenadas são enviadas na ordem oposta ao tempo físico em que foram enviadas. Na verdade, a definição de ordenação total permite que a distribuição de mensagens seja ordenada arbitrariamente, desde que a ordem seja a mesma em diferentes processos. Como a ordenação total não é necessariamente também uma ordenação FIFO, ou causal, definimos a mistura da ordenação FIFO-total como aquela para a qual a distribuição de mensagens obedece a ordem FIFO e a total; analogamente, sob a ordenação causal-total, a distribuição de mensagens obedece a ordenação causal e a total.

As definições de *multicast* ordenado não presumem nem implicam em confiabilidade. Por exemplo, o leitor deve verificar que, na ordenação total, se o processo correto p entrega a mensagem m e depois m', então um processo correto q poderá entregar m sem também entregar m', ou qualquer outra mensagem ordenada após m.

Também podemos formar misturas dos protocolos ordenados e confiáveis. O *multicast* totalmente confiável é freqüentemente referido na literatura como *multicast atômico*. Analogamente, podemos formar um *multicast* FIFO confiável, um *multicast* causal confiável e versões confiáveis de *multicasts* ordenados mistos.

Ordenar a entrega de mensagens *multicast*, conforme veremos, pode ser dispendioso em termos de latência da distribuição e de consumo de largura de banda. A semântica da ordenação que descrevemos pode atrasar desnecessariamente a distribuição das mensagens. Isto é, no nível aplicativo, uma mensagem pode ser retardada por outra mensagem da qual, na verdade, não depende. Por isso, alguns têm proposto sistemas *multicast* que utilizam apenas a semântica de mensagem específica à própria aplicação para determinar a ordem de distribuição de mensagens [Cheriton e Skeen 1993, Pedone e Schiper 1999].

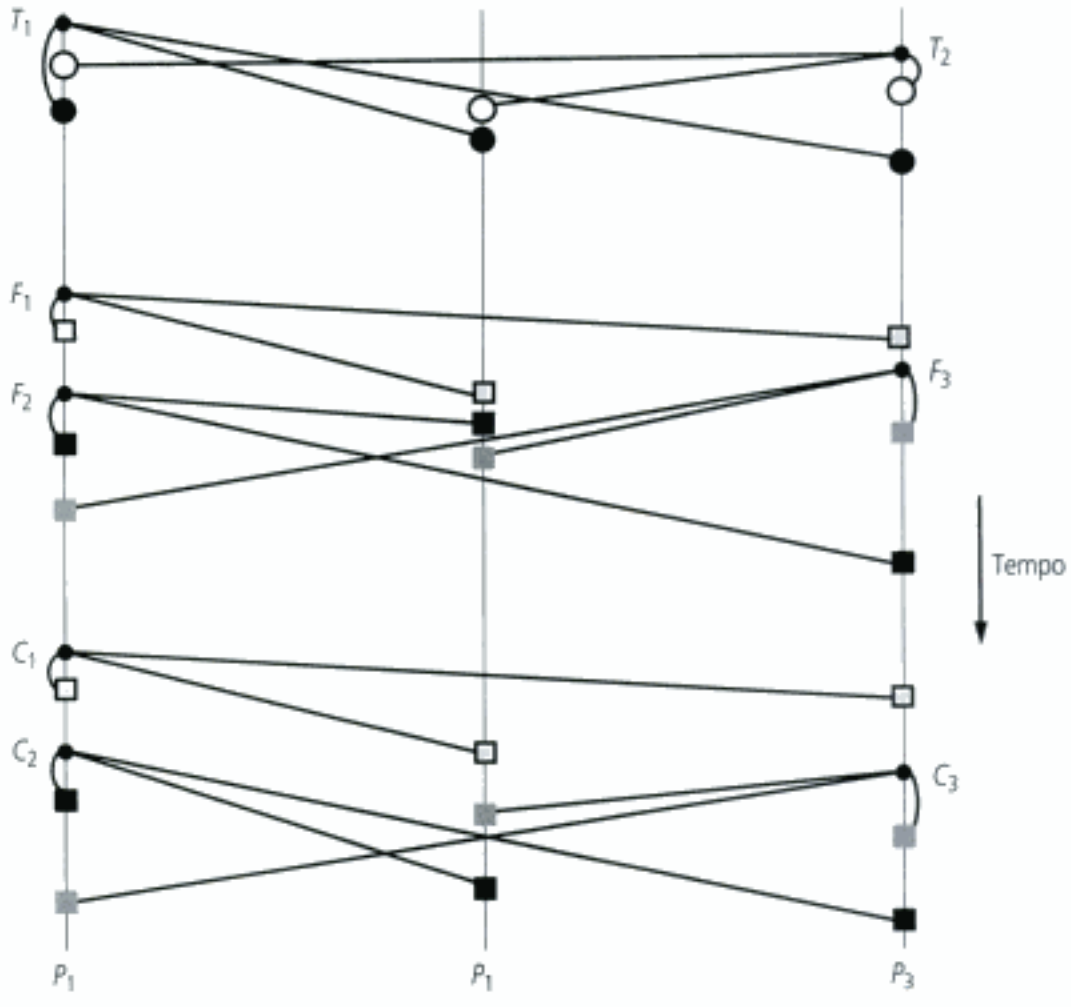


Figura 12.12 Ordenação total, FIFO e causal de mensagens de *multicast*.
Observe a ordem consistente das mensagens totalmente ordenadas T_1 e T_2, das mensagens relacionadas com FIFO F_1 e F_2 e das mensagens relacionadas por causalidade C_1 e C_3 – e a ordem de distribuição arbitrária das mensagens.

O exemplo de listas de discussão ◊ Para tornar a semântica de distribuição por *multicast* mais concreta, considere uma aplicação na qual os usuários postam mensagens para listas de discussão. Cada usuário executa um processo aplicativo de lista de discussão. Cada tópico de discussão tem seu próprio grupo de processos. Quando um usuário posta uma mensagem em uma lista de discussão, o aplicativo envia em *multicast* a postagem do usuário para o grupo correspondente. O processo de cada usuário é um membro do grupo para o assunto no qual ele está interessado, de modo que o usuário receberá apenas as postagens relativas a esse tópico.

O *multicast* confiável é exigido se cada usuário precisa receber cada postagem. Os usuários também têm requisitos de ordenação. A Figura 12.13 mostra as postagens conforme elas aparecem para um usuário em particular. No mínimo, a ordem FIFO é desejável, pois então cada postagem de determinado usuário – digamos, A.Hanlon – será recebida na mesma ordem e os usuários poderão falar consistentemente sobre a segunda postagem de A.Hanlon.

Note que a mensagem cujos assuntos são "Re: Microkernels" (25) e "Re: Mach" (27) aparece após as mensagens às quais elas se referem. Um *multicast* ordenado por causalidade é necessário para garantir esse relacionamento. Caso contrário, atrasos de mensagem arbitrários poderiam significar, digamos, que uma mensagem "Re: Mach" poderia aparecer antes da mensagem original sobre Mach.

Se a distribuição por *multicast* fosse totalmente ordenada, então a numeração na coluna da esquerda seria consistente entre os usuários. Os usuários poderiam se referir sem ambigüidade, por exemplo, à "mensagem 24".

Lista de discussão: os.interesting		
Item	*De*	*Assunto*
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	desempenho do RPC
27	M.Walker	Re: Mach
fim		

Figura 12.13 Tela do programa de listas de discussão.

Na prática, o sistema de listas de discussão USENET não implementa nem ordenação causal nem total. Os custos da comunicação para obter essas ordenações em larga escala superam suas vantagens.

Implementando a ordem FIFO ◊ O *multicast* com ordem FIFO (com operações *FO-multicast* e *FO-deliver*) é obtida com números de seqüência, como obteríamos para comunicação de um para um. Vamos considerar apenas grupos que não se sobrepõem. O leitor deve verificar que o protocolo de *multicast* confiável, que definimos sobre *multicast* IP, na Seção 12.4.2, também garante a ordem FIFO, mas mostraremos como se faz para construir um *multicast* com ordem FIFO sobre qualquer *multicast* básico dado. Usamos as variáveis S_g^p e R_g^q mantidas no processo p, do protocolo de *multicast* confiável da Seção 12.4.2: S_g^p é a contagem de quantas mensagens p enviou para g e, para cada q, R_g^q é o número de seqüência da mensagem mais recente que p entregou do processo q, que foi enviada para o grupo g.

Para p enviar uma mensagem com *FO-multicast* para o grupo g, ele coloca o valor S_g^p 'de carona' na mensagem, envia a mensagem com *B-multicast* para g e, em seguida, incrementa S_g^p por 1. Ao receber uma mensagem de q portando o número de seqüência S, p verifica se $S = R_g^q + 1$. Se for, essa mensagem é a próxima esperada do remetente q e p a entrega com *FO-deliver*, configurando $R_g^q := S$. Se $S > R_g^q + 1$, ele coloca a mensagem na fila de espera até que as mensagens intervenientes tenham sido entregues e $S = R_g^q + 1$.

Como todas as mensagens de determinado remetente são distribuídas na mesma seqüência, e como a distribuição de uma mensagem é retardada até que seu número de seqüência tenha sido atingido, a condição para a ordem FIFO é claramente satisfeita. Mas isso só funciona sob a suposição de que os grupos não se sobrepõem.

Note que podemos usar qualquer implementação de *B-multicast* nesse protocolo. Além disso, se usarmos uma primitiva *R-multicast* confiável, em vez de *B-multicast*, então obteremos um *multicast* FIFO confiável.

Implementando a ordem total ◊ A estratégia básica para implementar a ordem total é atribuir identificadores totalmente ordenados às mensagens de *multicast* para que cada processo tome a mesma decisão de ordenação com base nesses identificadores. O algoritmo de distribuição é muito parecido com aquele que descrevemos para a ordem FIFO; a diferença é que os processos mantêm números de seqüência específicos do grupo, em vez de números de seqüência específicos do processo. Consideramos apenas como ordenar totalmente as mensagens enviadas para grupos que não se sobrepõem. Chamamos as operações de *multicast* de *TO-multicast* e *TO-deliver*.

Discutimos dois métodos principais para atribuir identificadores às mensagens. O primeiro deles usa um processo chamado *seqüenciador* para essa atribuição (Figura 12.14). Um processo que queira enviar uma mensagem m com *TO-multicast* para o grupo g anexa nela um identificador exclusivo $id(m)$. As mensagens de g são enviadas para o seqüenciador de g, *sequencer*(g), assim como para os membros de g. (O seqüenciador pode ser escolhido como um membro de g.) O processo *sequencer*(g) mantém um número de seqüência específico do grupo s_g, o qual utiliza para atribuir números de seqüência cada vez maiores e consecutivos às mensagens que entrega com *B-deliver*. Ele anuncia os números de seqüência por meio do envio de mensagens com *B-multicast* para g (veja os detalhes na Figura 12.14).

1. Algoritmo do membro do grupo p

Na inicialização: $r_g := 0$;

Para enviar a mensagem m com TO-multicast para o grupo g
B-multicast($g \cup \{sequencer(g)\}$, <*m*, *i*>);

Em B-deliver(<*m*, *i*>) *com* $g = group(m)$
Coloca <*m*, *i*> na fila de espera;

Em B-deliver(m_{order} = < 'order' *i*, *S*>) *com* $g = grupo(m_{order})$
espera até <*m*, *i*> na fila de espera e $S = r_g$;
TO-deliver m; // (após excluí-la da fila de espera)
$r_g := S + 1$;

2. Algoritmo do seqüenciador de g

Na inicialização: $s_g := 0$;

Em B-deliver(<*m*, *i*>) *com* $g = group(m)$
B-multicast(*g*, <"order", *i*, s_g >);
$s_g := s_g + 1$;

Figura 12.14 Ordem total usando um seqüenciador.

Uma mensagem permanecerá na fila de espera indefinidamente até que possa ser entregue com *TO-deliver*, de acordo com o número de seqüência correspondente. Como os números de seqüência são bem definidos (pelo seqüenciador), o critério da ordem total é satisfeito. Além disso, se os processos usam uma variante de *B-multicast* com ordem FIFO, então o *multicast* totalmente ordenado também é ordenado por causalidade. Deixamos o leitor mostrar isso.

O problema óbvio de um esquema baseado em seqüenciador é que ele pode se tornar um gargalo e um ponto único de falha. Existem algoritmos práticos que tratam do problema da falha. Chang e Maxemchuk [1984] foram os primeiros a sugerir um protocolo de *multicast* empregando um seqüenciador (que chamaram de *site de token*). Kaashoek *et al.* [1989] desenvolveram um protocolo baseado em seqüenciador para o sistema Amoeba. Esses protocolos garantem que uma mensagem esteja na fila de espera em $f + 1$ nós antes de ser entregue; assim, até f falhas podem ser toleradas. Assim como Chang e Maxemchuk, Birman *et al.* [1991] também empregaram um site contendo um *token* que atua como seqüenciador. O *token* pode ser passado de um processo para outro para que, por exemplo, se apenas um processo enviar *multicasts* totalmente ordenados, então esse processo poderá atuar como seqüenciador, evitando comunicação.

O protocolo de Kaashoek *et al.* usa *multicast* baseado em hardware – disponível em uma rede Ethernet, por exemplo –, em vez da comunicação ponto a ponto confiável. Na variante mais simples desse protocolo, os processos enviam a mensagem *multicast* para o seqüenciador, na base de um para um. O seqüenciador faz um *multicast* para si mesmo, assim como o identificador e o número de seqüência. Isso tem a vantagem de que os outros membros do grupo recebem apenas uma única mensagem *multicast*; sua desvantagem é a maior utilização de largura de banda. O protocolo está completamente descrito no endereço www.cdk4.net/coordination.

O segundo método que examinaremos para obter *multicast* totalmente ordenado é aquele no qual os processos concordam coletivamente sobre a atribuição de números de seqüência às mensagens de maneira distribuída. Um algoritmo simples – semelhante àquele que foi originalmente desenvolvido para implementar distribuição por *multicast* totalmente ordenado para o *toolkit* ISIS [Birman e Joseph 1987a] – aparece na Figura 12.15. Mais uma vez, um processo envia sua mensagem com *B-multicast* para os membros do grupo. O grupo pode ser aberto ou fechado. Os processos receptores propõem números de seqüência para as mensagens quando elas chegam e os retornam para o remetente, o qual os utiliza para gerar números de seqüência acordados.

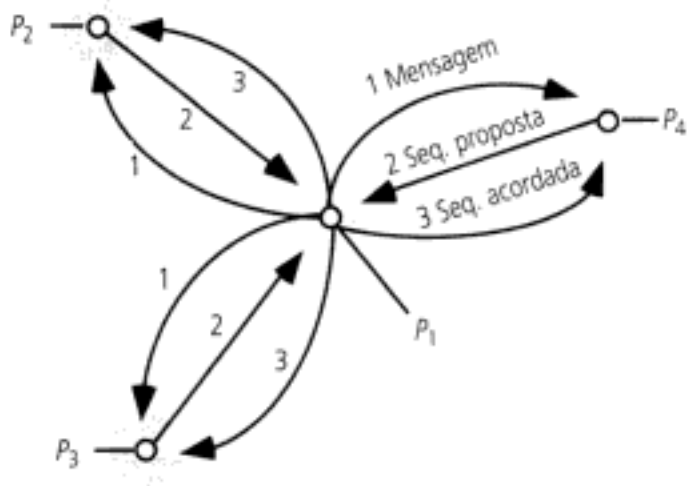


Figura 12.15 O algoritmo ISIS para ordem total.

Cada processo q no grupo g mantém A_g^q, o maior número de seqüência acordado observado até o momento para o grupo g, e p_g^q, seu próprio maior número de seqüência proposto. O algoritmo do processo p para enviar por *multicast* uma mensagem m para o grupo g é o seguinte:

1. p envia <m, i> para g com *B-multicast*, onde i é um identificador exclusivo para m.
2. Cada processo q responde ao remetente p com uma proposta para o número de seqüência acordado da mensagem de $p_g^q := Max(A_g^q, P_g^q) + 1$. Na realidade, devemos incluir identificadores de processo nos valores p_g^q propostos, para garantir uma ordem total, pois de outro modo, diferentes processos poderiam propor o mesmo valor inteiro; mas, por simplicidade, não vamos tornar isso explícito aqui. Cada processo atribui provisoriamente o número de seqüência proposto para a mensagem e a coloca em sua fila de espera, a qual é ordenada com o menor número de seqüência na frente.
3. p coleta todos os números de seqüência propostos e seleciona o maior a como o próximo número de seqüência concordado. Então, ele envia <i, a> para g com *B-multicast*. Cada processo q em g configura $A_g^q := Max(A_g^q, a)$ e anexa a na mensagem (que é identificada por i). Ele reordena a mensagem na fila de espera, caso o número de seqüência acordado seja diferente do proposto. Quando a mensagem que está na frente da fila de espera tiver recebido seu número de seqüência acordado, ela será transferida para o fim da fila de entrega. Entretanto, as mensagens que receberam seu número de seqüência acordado, mas que não estão na frente da fila de espera, ainda não são transferidas.

Se cada processo concordar com o mesmo conjunto de números de seqüência e os entregar na ordem correspondente, então a ordem total será satisfeita. É claro que, em última análise, os processos corretos concordam com o mesmo conjunto de números de seqüência, mas devemos mostrar que eles são monotonicamente cada vez maiores e que nenhum processo correto pode enviar uma mensagem prematuramente.

Suponha que uma mensagem m_1 tenha recebido um número de seqüência acordado e tenha chegado na frente da fila de espera. Por construção, uma mensagem recebida após esse estágio será e deve ser entregue após m_1: ela terá um número de seqüência proposto maior e, portanto, um número de seqüência acordado maior do que m_1. Assim, seja m_2 qualquer outra mensagem que ainda não recebeu seu número de seqüência acordado, mas que está na mesma fila. Temos que:

$$agreedSequence\ (m_2) \geq proposedSequence(m_2)$$

pelo algoritmo que acabamos de ver. Como m_1 está na frente da fila:

$$proposedSequence(m_2) > agreedSequence(m_1)$$

Portanto:

$agreedSequence(m_2) > agreedSequence(m_1)$

e a ordem total está garantida.

Esse algoritmo tem latência mais alta do que o *multicast* baseado em seqüenciador: três mensagens são enviadas em série entre o remetente e o grupo, antes que uma mensagem possa ser entregue.

Note que a ordem total escolhida por esse algoritmo não garante também a ordem por causalidade ou FIFO: quaisquer duas mensagens são enviadas em uma ordem total basicamente arbitrária, influenciada pelos atrasos da comunicação.

Para ver outras estratégias para implementar a ordem total, consulte Melliar-Smith *et al.* [1990], Garcia-Molina e Spauster [1991] e Hadzilacos e Toueg [1994].

Implementando a ordem causal ◊ Fornecemos um algoritmo para grupos fechados que não se sobrepõem, baseado naquele desenvolvido por Birman *et al.* [1991], mostrado na Figura 12.16, no qual as operações de *multicast* ordenados por causalidade são *CO-multicast* e *CO-deliver*. O algoritmo leva em conta o relacionamento antes do acontecido apenas quando é estabelecido por mensagens de *multicast*. Se os processos enviarem mensagens de um para um entre si, então elas não serão consideradas.

Cada processo p_i ($i = 1, 2, \ldots, N$) mantém sua própria *indicação de tempo vetorial* (veja a Seção 11.4). As entradas na indicação de tempo contam o número de mensagens de *multicast* de cada processo que aconteceram antes da mensagem *multicast* seguinte ser enviada.

Para enviar uma mensagem com *CO-multicast* para o grupo g, o processo adiciona 1 em sua entrada na indicação de tempo e envia a mensagem com *B-multicast*, junto com sua indicação de tempo, para g.

Quando um processo p_i entrega uma mensagem com *B-deliver* a partir de p_j, ele deve colocá-la na fila de espera, antes de poder enviá-la com *CO-deliver*: até ter certeza de que enviou todas as mensagens que a precediam por causalidade. Para estabelecer isso, p_i espera até que (a) tenha entregue qualquer mensagem anterior enviada por p_j e (b) tenha entregue qualquer mensagem que p_j tenha distribuído no instante em que ele fez o *multicast* da mensagem. Essas duas condições podem ser detectadas examinando-se as indicações de tempo vetoriais, como mostrado na Figura 12.16. Note que um processo pode entregar imediatamente para si mesmo, com *CO-deliver*, qualquer mensagem que tenha enviado com *CO-multicast*, embora isso não esteja descrito na Figura 12.16.

Cada processo atualiza sua indicação de tempo vetorial ao entregar qualquer mensagem, para manter a contagem de mensagens de causalidade precedentes. Ele faz isso incrementando por um a j-ésima entrada de indicação de tempo. Essa é uma otimização da operação *merge* que apareceu nas regras de atualização de relógios vetoriais na Seção 11.4. Podemos fazer a otimização em vista da condição de distribuição no algoritmo da Figura 12.16, a qual garante que apenas a j-ésima entrada aumentará.

Descrevemos em linhas gerais a prova da correção desse algoritmo, como segue. Suponha que $multicast(g, m) \rightarrow multicast(g, m')$. Sejam V e V' as indicações de tempo vetoriais de m e m', respectivamente. É fácil provar, por indução a partir do algoritmo, que $V < V'$. Em particular, se o processo p_k envia *multicast* m, então $V[k] \leq V'[k]$.

Algoritmo para membro de grupo p_i ($i = 1, 2 \ldots, N$)

Na inicialização

$V_i^g[j] := 0$ ($j = 1, 2 \ldots, N$);

Para enviar a mensagem m com CO-multicast para o grupo g

$V_i^g[i] := V_i^g[i] + 1$;

B-multicast(g, $< V_i^g, m >$);

Em B-deliver($< V_j^g, m >$) *de* $p_j (j \neq i)$, *com* $g = group(m)$

coloca $< V_j^g, m >$ na fila de espera;

espera até que $V_j^g[j] = V_i^g[j] + 1$ e $V_j^g[k] \leq V_i^g[k] (k \neq j)$;

CO-deliver m; // após removê-la da fila de espera

$V_i^g[j] := V_i^g[j] + 1$;

Figura 12.16 Ordem causal usando indicações de tempo vetoriais.

Hidden page

Hidden page

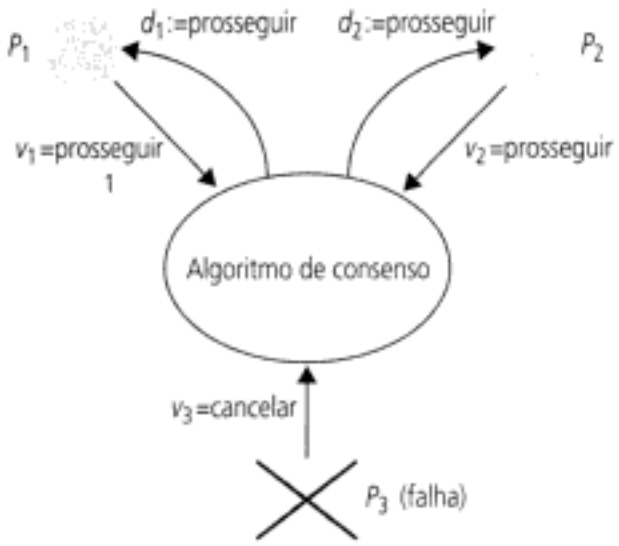


Figura 12.17 Consenso de três processos.

Acordo: o valor da decisão de todos os processos corretos é o mesmo: se p_i e p_j estão corretos e entraram no estado *decidido*, então $d_i = d_j$ ($i, j = 1, 2, \ldots, N$).

Integridade: se todos os processos corretos propuseram o mesmo valor, então qualquer processo correto no estado *decidido* escolheu esse valor.

De acordo com a aplicação, variações na definição da integridade podem ser apropriadas. Por exemplo, um tipo de integridade mais fraca seria o valor da decisão ser igual a um valor que algum processo correto propôs – não necessariamente todos eles. Usaremos a definição declarada anteriormente.

Para ajudar no entendimento de como a formulação do problema se transforma em um algoritmo, considere um sistema no qual os processos não podem falhar. Então, é simples resolver o consenso. Por exemplo, podemos reunir os processos em um grupo e fazer cada processo enviar em *multicast*, de modo confiável, seu valor proposto para os membros do grupo. Cada processo espera até que tenha reunido todos os N valores (incluindo o seu próprio). Então, ele avalia a função *majority*($v_1, v_2, \ldots, v_N$), que retorna o valor que ocorre mais freqüentemente entre seus argumentos, ou o valor especial $\perp \notin D$, caso não haja maioria. O término é garantido pela confiabilidade da operação de *multicast*. O acordo e a integridade são garantidos pela definição de *maioria* e pela propriedade da integridade de um *multicast* confiável. Todo processo recebe o mesmo conjunto de valores propostos e todo processo avalia a mesma função desses valores. Portanto, todos devem concordar e, se todo processo propôs o mesmo valor, então todos eles decidem por esse valor.

Note que *majority* (maioria) é a única função possível que os processos poderiam usar para concordar a respeito de um valor, a partir de valores candidatos. Por exemplo, se os valores fossem ordenados, então as funções *minimum* e *maximum* (mínimo e máximo) poderiam ser apropriadas.

Se os processos podem falhar, então isso introduz a complicação de detectar falhas e não é imediatamente claro que uma execução do algoritmo de consenso possa terminar. Na verdade, se o sistema for assíncrono, então poderá não terminar; voltaremos a esse ponto em breve.

Se os processos podem falhar de maneiras *arbitrárias* (bizantinas), então os processos falhos podem, em princípio, comunicar valores aleatórios para os outros. Isso pode parecer improvável na prática, mas não está fora dos limites da possibilidade o fato de um processo com um erro falhar dessa maneira. Além disso, a falha pode não ser acidental, mas sim o resultado de uma operação nociva ou mal-intencionada. Alguém poderia fazer um processo enviar deliberadamente valores diferentes para diferentes pares, em uma tentativa de frustrar os outros que estariam tentando chegar a um consenso. No caso de haver uma inconsistência, os processos corretos devem comparar os valores que receberam com os que os outros processos dizem ter recebido.

Hidden page

$$CI_i(v_1, v_2, ..., v_N)[j] = GB_i(j, v_j)\ (i, j = 1, 2, ..., N)$$

C a partir de CI: para o caso onde a maioria dos processos está correta, construímos uma solução para C a partir de CI executando CI para produzir um vetor de valores em cada processo, aplicando então uma função apropriada nos valores do vetor para derivar um único valor:

$$C_i(v_1,...v_N) = majority(CI_i(v_1,...,v_N)[1],...,CI_i(v1,...,v_N)[N])$$

($i = 1, 2 \ldots N$), onde *majority* (maioria) é conforme definido anteriormente.

GB a partir de C: construímos uma solução para GB a partir de C, como segue:

- O comandante p_j envia seu valor proposto v para si mesmo e para cada um dos processos restantes;
- Todos os processos executam C com os valores ($v_1, v_2 \ldots, v_N$) que recebem (p_j pode ser falho);
- Eles derivam $GB_i(j, v) = C_i(v_1, v_2 \ldots, v_N)$ ($i = 1, 2, \ldots, N$).

O leitor deve verificar que as condições de término, acordo e integridade são preservadas em cada caso. Fischer [1983] relaciona os três problemas com mais detalhes.

Em sistemas com falhas por colapso, o consenso é equivalente a solucionar *multicast* confiável e totalmente ordenado: dada uma solução para um, podemos solucionar o outro. É fácil implementar o consenso com uma operação de *multicast* confiável e totalmente ordenado *RTO-multicast*. Reunimos todos os processos em um grupo g. Para chegar ao consenso, cada processo p_i executa *RTO-multicast*(g, v_i). Então, cada processo p_i escolhe $d_i = m_i$, onde m_i é o primeiro valor enviado por p_i com *RTO-deliver*. A propriedade do término resulta da confiabilidade do *multicast*. As propriedades do acordo e da integridade resultam da confiabilidade e da ordenação total da distribuição por *multicast*. Chandra e Toueg [1996] demonstram como o *multicast* confiável e totalmente ordenado pode ser derivado a partir do consenso.

12.5.2 Consenso em um sistema síncrono

Esta seção descreve um algoritmo que usa apenas um protocolo de *multicast* básico para resolver o consenso em um sistema síncrono. O algoritmo presume que até f dos N processos apresentam falhas por colapso.

Para chegar a um consenso, cada processo correto reúne os valores propostos pelos outros processos. O algoritmo prossegue em $f + 1$ rodadas, em cada uma das quais os processos corretos distribuem os valores entre eles mesmos com *B-multicast*. Por suposição, no máximo f processos podem falhar. Na pior das hipóteses, todas as f falhas ocorreram durante as rodadas, mas o algoritmo garante que no final das rodadas todos os processos corretos que sobreviveram estão em condições de concordar.

O algoritmo, mostrado na Figura 12.18, é baseado no algoritmo de Dolev e Strong [1983] e em sua apresentação por Attiya e Welch [1998]. A variável $Values_i^r$ contém o conjunto de valores propostos, conhecidos pelo processo p_i no início da rodada r. Cada processo envia por *multicast* o conjunto de valores que não enviou nas rodadas anteriores. Então, o processo recebe mensagens *multicast* de outros processos e registra todos os novos valores. Embora isso não apareça na Figura 12.18, a duração de uma rodada é limitada pela configuração de um tempo limite baseado no tempo máximo para um processo correto fazer o *multicast* de uma mensagem. Após $f + 1$ rodadas, cada processo escolhe como seu valor de decisão o valor mínimo que recebeu.

O término é óbvio, a partir do fato de que o sistema é síncrono. Para verificar a correção do algoritmo, devemos mostrar que cada processo chega ao mesmo conjunto de valores no final da última rodada. Então, resultam o acordo e a integridade, pois os processos aplicam a função *minimum* nesse conjunto.

Suponha, ao contrário, que dois processos difiram em seu conjunto de valores final. Sem perda de generalidade, um processo correto p_i possui um valor v que outro processo correto p_j ($i \neq j$) não possui. A única explicação para o fato de p_i possuir no final um valor proposto v que p_j não possui, é que um terceiro processo, digamos, p_k, que conseguiu enviar v para p_i, falhou antes que v pudesse ser enviado para p_j. Por sua vez, um processo enviando v na rodada anterior deve ter falhado, para explicar porque

```
Algoritmo do processo p_i ∈ g; o algoritmo prossegue em f + 1 rodadas
Na inicialização
    Values_i^1 := {v_i}; Values_i^0 := { };
Na rodada r (1 ≤ r ≤ f + 1)
    B-multicast(g, Values_i^r – Values_i^(r-1)); // Envia apenas os valores que não foram enviados
    Values_i^(r+1) := Values_i^r;
    while (na rodada r)
    {
            Em B-deliver(V_j) de um p_j
                Values_i^(r+1) := Values_i^(r+1) ∪ V_j;
    }
Após (f + 1) rodadas
    Atribui d_i = minimum(Values_i^(f+1));
```

Figura 12.18 Consenso em um sistema síncrono.

p_k possui v nessa rodada, mas p_j não o recebeu. Prosseguindo dessa maneira, temos que postular pelo menos uma falha em cada uma das rodadas anteriores. Mas pressupomos que no máximo f falhas podem ocorrer e que existem $f + 1$ rodadas. Chegamos a uma contradição.

Verifica-se que qualquer algoritmo, para chegar a um consenso, a despeito de até f falhas por colapso, exige pelo menos $f + 1$ rodadas de trocas de mensagem, independentemente de como é construído [Dolev e Strong 1983]. Esse limite inferior também se aplica no caso das falhas bizantinas [Fischer e Lynch 1982].

12.5.3 O problema dos generais bizantinos em um sistema síncrono

Discutimos o problema dos generais bizantinos em um sistema síncrono. Ao contrário do algoritmo de consenso descrito na seção anterior, supomos aqui que os processos podem apresentar falhas arbitrárias. Isto é, a qualquer momento, um processo falho pode enviar qualquer mensagem com qualquer valor; e ele pode omitir o envio de qualquer mensagem. Até f dos N processos podem ser falhos. Os processos corretos podem detectar a ausência de uma mensagem por meio de um tempo limite, mas não podem concluir que o remetente falhou, pois ele pode ficar em silêncio por algum tempo e depois enviar mensagens novamente.

Supomos que os canais de comunicação entre pares de processos são privados. Se um processo pudesse examinar todas as mensagens enviadas por outros processos, então ele poderia detectar as inconsistências no que um processo falho envia para diferentes processos. Nossa suposição básica de confiabilidade do canal significa que nenhum processo falho pode injetar mensagens no canal de comunicação entre processos corretos.

Lamport *et al.* [1982] consideraram o caso de três processos enviando, um para o outro, mensagens não assinadas. Eles mostraram que não existe nenhuma solução que garanta o atendimento das condições do problema dos generais bizantinos, caso um processo possa falhar. Eles generalizaram esse resultado para mostrar que não existe nenhuma solução se $N \leq 3f$. Vamos demonstrar esses resultados em breve. Eles forneceram um algoritmo que resolve o problema dos generais bizantinos em um sistema síncrono se $N \geq 3f + 1$, para mensagens não assinadas (eles as chamam de mensagens "orais").

Impossibilidade com três processos ◊ A Figura 12.19 mostra dois cenários nos quais apenas um de três processos é falho. Na configuração da esquerda, um dos tenentes, p_3, é falho; na da direita, o comandante, p_1, é falho. Cada cenário da Figura 12.19 mostra duas rodadas de mensagens: os valores enviados pelo comandante e os valores que os tenentes enviam subseqüentemente uns para os outros. Os prefixos numéricos servem para especificar as fontes das mensagens e para mostrar as diferentes rodadas. Leia o símbolo ":" nas mensagens como "fala"; por exemplo, "3:1:u" é a mensagem "3 fala 1 fala u".

Hidden page

Os generais corretos chegam a um acordo em duas rodadas de mensagens:

- Na primeira rodada, o comandante envia um valor para cada um dos tenentes.
- Na segunda rodada, cada um dos tenentes envia o valor que recebeu para seus pares.

Um tenente recebe um valor do comandante, mais $N - 2$ valores de seus pares. Se o comandante for falho, então todos os tenentes estão corretos e cada um terá reunido exatamente o conjunto de valores enviados pelo comandante. Caso contrário, um dos tenentes é falho; cada um de seus pares corretos recebe $N - 2$ cópias do valor enviado pelo comandante, mais um valor enviado pelo tenente falho.

Em qualquer caso, os tenentes corretos só precisam aplicar uma função de maioria simples no conjunto de valores que recebem. Como $N \geq 4$, $(N - 2) \geq 2$. Portanto, a função *majority* ignorará qualquer valor enviado por um tenente falho e produzirá o valor enviado pelo comandante, caso o comandante seja correto.

Agora, ilustraremos o algoritmo que acabamos de descrever em linhas gerais, para o caso de quatro generais. A Figura 12.20 mostra dois cenários semelhantes àqueles da Figura 12.19, mas, neste caso, existem quatro processos, sendo um deles falho. Como na Figura 12.19, na configuração da esquerda um dos tenentes, p_3, é falho; na da direita, o comandante, p_1, é falho.

No caso da esquerda, os dois processos tenentes corretos concordam, decidindo-se pelo valor do comandante:

p_2 decide-se por $(v, u, v) = v$

p_4 decide-se por $(v, v, w) = v$

No caso da direita, o comandante é falho, mas os três processos corretos concordam:

p_2, p_3 e p_4 decidem-se por $majority(u, v, w) = \perp$ (o valor especial $\perp$ se aplica onde não existe nenhuma maioria de valores).

O algoritmo leva em conta o fato de que um processo falho pode omitir o envio de uma mensagem. Se um processo correto não recebe uma mensagem dentro de um limite de tempo conveniente (o sistema é síncrono), ele prossegue como se o processo falho tivesse enviado o valor $\perp$.

Discussão ◊ Podemos medir a eficiência de uma solução para o problema dos generais bizantinos – ou qualquer outro problema de acordo – perguntando:

- Quantas rodadas de mensagem são necessárias? (Esse é um fator para o tempo que o algoritmo demora a terminar.)
- Quantas mensagens são enviadas e de que tamanho? (Isso mede a utilização de largura de banda total e tem um impacto sobre o tempo de execução.)

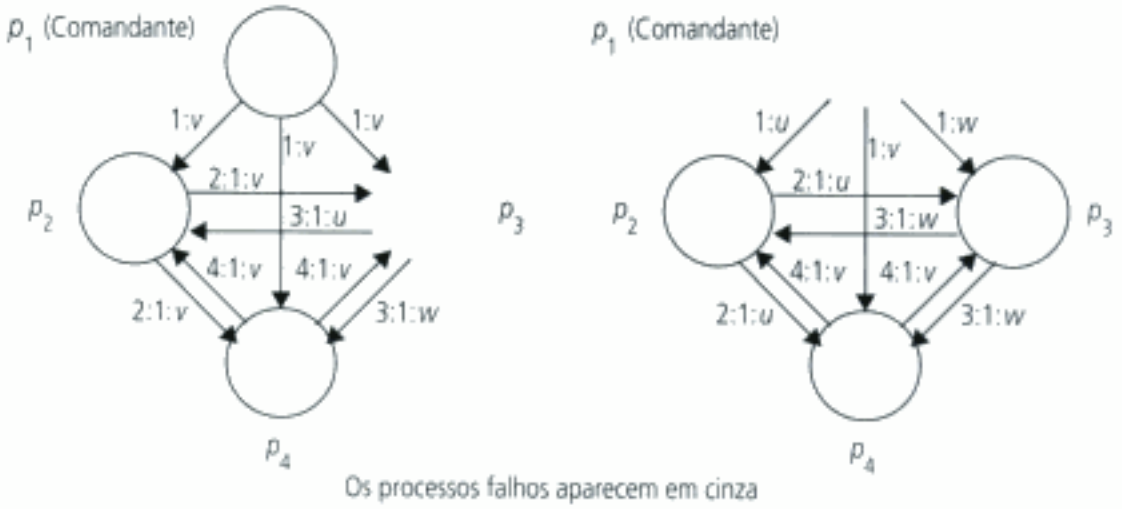


Figura 12.20 Quatro generais bizantinos.

Hidden page

comportará como um processo correto, mas que às vezes leva um longo tempo para executar uma etapa do processamento.

O mascaramento, geralmente, é aplicável ao projeto de sistemas. O Capítulo 14 discutirá como os sistemas transacionais tiram proveito do armazenamento persistente. O Capítulo 15 descreverá como as falhas de processo também podem ser mascaradas pela replicação de componentes de software.

Consenso usando detectores de falha ◊ Outro método para contornar o resultado da impossibilidade emprega detectores de falha. Alguns sistemas práticos empregam detectores de falha "de projeto perfeito" para chegar ao consenso. Nenhum detector de falha em um sistema assíncrono que funcione unicamente pela passagem de mensagens pode ser realmente perfeito. Entretanto, os processos podem concordar em *julgar* falho um processo que não respondeu por mais do que um tempo limitado. Um processo que não responde pode não ter falhado realmente, mas os processos restantes agem como se ele tivesse falhado. Eles tornam a falha "silenciosa", descartando as mensagens subseqüentes que recebem de um processo considerado "falho". Em outras palavras, transformamos efetivamente um sistema assíncrono em síncrono. Essa técnica é usada no sistema ISIS [Birman 1993].

Esse método conta com o fato do detector de falha normalmente ser preciso. Quando ele é impreciso, o sistema tem de prosseguir sem um membro do grupo que, de outro modo, poderia ter contribuído para sua eficácia. Infelizmente, tornar o detector de falha razoavelmente preciso envolve o uso de valores de tempo limite longos, obrigando os processos a esperarem por um tempo relativamente grande (e a não realizar trabalho útil) antes de concluírem que um processo falhou. Outro problema que surge com essa estratégia é o particionamento da rede, que discutiremos no Capítulo 15.

Uma estratégia bastante diferente é usar detectores de falha imperfeitos e chegar ao consenso enquanto se permite que os processos suspeitos se comportem corretamente, em vez de excluí-los. Chandra e Toueg [1996] analisaram as propriedades que um detector de falha deve ter para resolver o problema do consenso em um sistema assíncrono. Eles mostraram que o consenso pode ser resolvido em um sistema assíncrono, mesmo com um detector de falha não confiável, se menos de $N/2$ processos falharem e a comunicação for confiável. O tipo de detector de falha para o qual isso vale é chamado de *detector de falha finalmente fraco*. Ele é:

> *Finalmente pouco completo*: cada processo falho é finalmente tido como suspeito permanentemente por algum processo correto;
>
> *Finalmente pouco preciso*: após algum ponto no tempo, pelo menos um processo correto nunca é tido como suspeito por qualquer processo correto.

Chandra e Toueg mostram que não podemos implementar um detector de falha finalmente fraco em um sistema assíncrono apenas pela passagem de mensagens. Entretanto, descrevemos um detector de falha baseado em mensagem, na Seção 12.1, que se adapta aos seus valores de tempo limite de acordo com os tempos de resposta observados. Se um processo (ou a conexão com ele) estiver muito lento, então o valor do tempo limite crescerá para que os casos de um processo falsamente suspeito se tornem raros. No caso de muitos sistemas reais, esse algoritmo se comporta de forma suficientemente parecida com um detector de falha finalmente fraco para propósitos práticos.

O algoritmo de consenso de Chandra e Toueg permite que processos falsamente suspeitos continuem com suas operações normais, e que os processos que suspeitaram deles recebam suas mensagens e processem essas mensagens normalmente. Isso torna a vida do programador de aplicativos complicada, mas tem a vantagem de que os processos corretos não são desperdiçados pelo fato de serem falsamente excluídos. Além disso, os tempos limites para detecção de falhas podem ser configurados de forma menos conservadora do que com a estratégia do ISIS.

Consenso usando aleatoriedade ◊ O resultado de Fischer *et al.* depende do que podemos considerar como "adversário". Trata-se de um "personagem" (na verdade, apenas um conjunto de eventos aleatórios) que pode explorar os fenômenos dos sistemas assíncronos para frustrar as tentativas dos processos de chegarem ao consenso. O adversário manipula a rede para atrasar as mensagens, de modo que elas cheguem justamente no momento errado e, analogamente, desacelera, ou acelera, os processos o suficiente para que eles estejam no estado "errado" ao receberem uma mensagem.

A terceira técnica que trata do resultado da impossibilidade é a introdução de um elemento de chance no comportamento dos processos, de modo que o adversário não possa exercer sua estratégia de impedimento eficientemente. O consenso pode ainda não ter sido atingido, em alguns casos, mas

esse método permite que os processos cheguem a um consenso em um tempo finito *esperado*. Um algoritmo probabilístico que resolve o consenso, mesmo com falhas bizantinas, pode ser encontrado em Canetti e Rabin [1993].

12.6 Resumo

Este capítulo começou discutindo a necessidade dos processos de acessar recursos compartilhados sob condições de exclusão mútua. Os bloqueios nem sempre são implementados pelos servidores que gerenciam os recursos compartilhados e, então, é exigido um serviço separado de exclusão mútua distribuída. Foram considerados três algoritmos que obtêm exclusão mútua: um empregando um servidor central, um algoritmo baseado em anel e um algoritmo baseado em *multicast*, usando relógios lógicos. Nenhum desses mecanismos, conforme os descrevemos, pode suportar falhas embora possam ser modificados para tolerar algumas delas.

Em seguida, este capítulo considerou um algoritmo baseado em anel e o algoritmo "valentão", cujo objetivo comum é eleger um processo exclusivamente a partir de um conjunto dado – mesmo que várias eleições ocorram concomitantemente. O algoritmo "valentão" poderia ser usado, por exemplo, para eleger um novo servidor de tempo mestre ou um novo servidor de travas (*lock server*), quando o anterior falhar.

Este capítulo descreveu a comunicação por *multicast*. Ele discutiu o *multicast* confiável, na qual os processos corretos concordam com o conjunto de mensagens a serem enviadas, e o *multicast* com ordenação de entrega FIFO, causal e total. Fornecemos algoritmos para *multicast* confiável e para todos os três tipos de ordem de entrega.

Finalmente, descrevemos os três problemas de consenso, generais bizantinos e consistência interativa. Definimos as condições para sua solução e mostramos os relacionamentos entre esses problemas – incluindo o relacionamento entre consenso e *multicast* totalmente ordenado confiável.

Existem soluções em um sistema síncrono e descrevemos algumas delas. Na verdade, existem soluções mesmo quando são possíveis falhas arbitrárias. Descrevemos em linhas gerais parte da solução para o problema dos generais bizantinos de Lamport *et al*. Algoritmos mais recentes têm menor complexidade, mas, em princípio, nenhum pode ser melhor do que as $f+1$ rodadas exigidas por esse algoritmo, a não ser que as mensagens tenham assinatura digital.

O capítulo terminou descrevendo o resultado fundamental de Fischer *et al*. a respeito da impossibilidade de garantir o consenso em um sistema assíncrono. Discutimos como é que, contudo, os sistemas chegam regularmente a um acordo em sistemas assíncronos.

Exercícios

12.1 É possível implementar um (processo) detector de falha confiável, ou um não confiável, usando um canal de comunicação não confiável? *página 409–410*

12.2 Se todos os processos clientes são *single-threaded*, a condição de exclusão mútua EM3, que especifica entrada na ordem antes do acontecido, é relevante? *página 412*

12.3 Forneça uma fórmula para o rendimento máximo (*throughput*) de um sistema de exclusão mútua em termos do atraso de sincronização. *página 412*

12.4 No algoritmo do servidor central para exclusão mútua, descreva uma situação na qual dois pedidos não são processados na ordem antes do acontecido. *página 412–413*

12.5 Adapte o algoritmo do servidor central para exclusão mútua para tratar da falha por colapso de qualquer cliente (em qualquer estado), supondo que o servidor seja correto e dado um detector de falha confiável. Comente se o sistema resultante é tolerante à falha. O que aconteceria se um cliente que possuísse o *token* fosse erroneamente suspeito de ter falhado? *página 412–413*

12.6 Dê um exemplo de execução do algoritmo baseado em anel para mostrar que os processos não têm necessariamente a entrada garantida na seção crítica na ordem antes do acontecido. *página 413–414*

12.7 Em determinado sistema, cada processo normalmente usa uma seção crítica muitas vezes, antes que outro processo a solicite. Explique por que o algoritmo de exclusão mútua baseado em *multicast* de Ricart e Agrawala é ineficiente para esse caso e descreva como fazer para melhorar seu desempenho. Sua adaptação satisfaz a condição de subsistência ME2? *página 415–416*

12.8 No algoritmo valentão, um processo de recuperação inicia uma eleição e se tornará o novo coordenador caso tenha um identificador mais alto do que o encarregado atual. Essa é uma característica necessária do algoritmo? *página 419–421*

12.9 Sugira como fazer para adaptar o algoritmo valentão para tratar com particionamentos temporários de rede (comunicação lenta) e processos lentos. *página 420–422*

12.10 Projete um protocolo para *multicast* básico sobre *multicast* IP. *página 423*

12.11 Como, se houver possibilidade, as definições de integridade, acordo e validade do *multicast* confiável devem mudar para o caso de grupos abertos? *página 424*

12.12 Explique por que inverter a ordem das linhas '*R-deliver m*' e '*if* $(q \neq p)$ *then B-multicast*(g, m); *end if*' na Figura 12.10 faz com que o algoritmo não satisfaça mais o acordo uniforme. O algoritmo de *multicast* confiável baseado no *multicast* IP satisfaz o acordo uniforme? *página 424–425*

12.13 Explique se o algoritmo de *multicast* confiável sobre *multicast* IP funciona para grupos abertos, assim como para grupos fechados. Dado qualquer algoritmo para grupos fechados, como podemos simplesmente derivar um algoritmo para grupos abertos? *página 424–425*

12.14 Considere como fazer para tratar das suposições impraticáveis que fizemos para atender as propriedades de validade e acordo para o protocolo de *multicast* confiável baseado no *multicast* IP. Dica: acrescente uma regra para excluir as mensagens retidas, quando elas tiverem sido entregues para todas as partes; e considere a inclusão de uma mensagem de "pulsação", que nunca é entregue para o aplicativo, mas que o protocolo envia se o aplicativo não tiver nenhuma mensagem para enviar. *página 424–425*

12.15 Mostre que o algoritmo de *multicast* de ordem FIFO não funciona para grupos sobrepostos, considerando duas mensagens enviadas da mesma fonte para dois grupos sobrepostos e considerando um processo na interseção desses grupos. Adapte o protocolo para funcionar para esse caso. Dica: os processos devem incluir com suas mensagens os números de seqüência mais recentes das mensagens enviadas para *todos* os grupos. *página 429–430*

12.16 Mostre que, se o *multicast* básico que usamos no algoritmo da Figura 12.14 também tiver ordem FIFO, então o *multicast* totalmente ordenado resultante também será ordenado por causalidade. É verdade que qualquer *multicast* ordenado com FIFO e totalmente ordenado é, por isso, ordenado por causalidade? *página 430*

12.17 Sugira como fazer para adaptar o protocolo de *multicast* ordenado por causalidade para manipular grupos sobrepostos. *página 432–433*

12.18 Na discussão do algoritmo de exclusão mútua de Maekawa, demos um exemplo de três subconjuntos de um conjunto de três processos que poderiam levar a um impasse. Use esses subconjuntos como grupos de *multicast* para mostrar como uma ordenação total com reconhecimento de pares não é necessariamente acíclica. *página 433*

12.19 Construa uma solução para o *multicast* totalmente ordenado e confiável em um sistema síncrono, usando um *multicast* confiável e uma solução para o problema do consenso. *página 433*

12.20 Fornecemos uma solução para o consenso a partir de uma solução para o *multicast* totalmente ordenado e confiável, que envolvia selecionar o primeiro valor a ser enviado. Explique, a partir dos primeiros princípios, por que em um sistema assíncrono não poderíamos, em vez disso, derivar uma solução usando um serviço de *multicast* confiável, mas não totalmente ordenado, e a função de "maioria". (Note que, se pudéssemos, isso contradiria o resultado da impossibilidade de Fischer *et al.*!) Dica: considere processos lentos/falhos. *página 437–438*

12.21 Mostre que o acordo bizantino pode ser conseguido por três generais, com um deles falho, caso os generais coloquem uma assinatura digital em suas mensagens. *página 438–439*

12.22 Explique como fazer para adaptar o algoritmo de *multicast* confiável sobre *multicast* IP para eliminar a fila de espera – de modo que uma mensagem recebida, que não seja uma duplicata, possa ser enviada imediatamente, mas sem quaisquer garantias de ordenação. Dica: use conjuntos, em vez de números de seqüência, para representar as mensagens que foram enviadas até o momento. *página 425–426*

Transações e Controle de Concorrência 13

Este capítulo discute a aplicação de transações e controle de concorrência em objetos compartilhados gerenciados por servidores.

Uma transação define uma seqüência de operações no servidor que garante que elas sejam atômicas na presença de várias falhas de clientes e de servidor. As transações aninhadas são estruturadas a partir de conjuntos de outras transações. Elas são particularmente úteis nos sistemas distribuídos porque permitem uma maior concorrência.

Todos os protocolos de controle de concorrência são baseados no critério da equivalência serial e são derivados das regras de conflitos entre operações. Serão descritos três métodos:

- Travas (*locks*) e bloqueios são usados para ordenar transações que acessam os mesmos objetos, de acordo com a ordem de chegada de suas operações nos objetos.
- O controle de concorrência otimista permite que as transações prossigam até estarem prontas para serem efetivadas, após isso é feita uma verificação para ver se efetuaram operações conflitantes nos objetos.
- A ordenação por indicação de tempo (*timestamp*) ordena as transações que acessam os mesmos objetos de acordo com seus tempos iniciais.

13.1 Introdução

O objetivo das transações é garantir que todos os objetos gerenciados por um servidor permaneçam em um estado consistente ao serem acessados por várias transações e na presença de falhas do servidor. O Capítulo 2 apresentou um modelo de falha para sistemas distribuídos. As transações tratam falhas por colapso de processos e falhas por omissão na comunicação, mas não com qualquer tipo de comportamento arbitrário (ou bizantino). O modelo de falha para transações será apresentado na Seção 13.1.2.

Os objetos que podem ser recuperados depois que seu servidor falha são chamados de *objetos recuperáveis*. Em geral, os objetos gerenciados por um servidor podem ser armazenados em memória volátil (por exemplo, a RAM), ou em memória persistente (por exemplo, o disco). Mesmo que os objetos sejam armazenados em memória volátil, o servidor pode usar memória persistente para armazenar informações suficientes para que, no caso de falhas do processo servidor, o estado dos objetos seja recuperado. Isso permite que os servidores tornem os objetos recuperáveis. Uma transação é especificada por um cliente como um conjunto de operações sobre os objetos a serem executadas como uma unidade indivisível pelos servidores que estão gerenciando esses objetos. Os servidores devem garantir que a transação inteira seja executada e que os resultados sejam gravados no armazenamento permanente ou, no caso de um, ou mais deles, falhar, que seus efeitos sejam completamente desconsiderados. O próximo capítulo discutirá problemas relacionados às transações que envolvem vários servidores, em particular, como eles decidem a respeito do resultado de uma transação distribuída. Este capítulo se concentra nos problemas de uma transação em um único servidor. A transação de um cliente também é considerada indivisível do ponto de vista das transações dos outros clientes, no sentido de que as operações de uma transação não podem observar os efeitos parciais das operações de outra. A Seção 13.1.1 discutirá a sincronização simples de acesso aos objetos e a Seção 13.2 apresentará as transações, as quais exigem técnicas mais avançadas para evitar interferência entre os clientes. A Seção 13.3 discutirá as transações aninhadas. As Seções 13.4 a 13.6 discutirão três métodos de controle de concorrência para transações cujas operações são todas endereçadas a um único servidor (bloqueios, controle de concorrência otimista e ordenação por indicação de tempo). O Capítulo 14 discutirá como esses métodos são ampliados para uso com transações cujas operações são endereçadas a vários servidores.

Para explicar alguns dos assuntos deste capítulo, usaremos um exemplo de transações bancárias, mostradas na Figura 13.1. Cada conta é representada por um objeto remoto cuja interface *Account* fornece operações para fazer depósitos e saques, e para solicitar e determinar o saldo. Cada agência do banco é representada por um objeto remoto cuja interface *Branch* fornece operações para criar uma nova conta, pesquisar uma conta pelo nome e solicitar os fundos totais nessa agência.

13.1.1 Sincronização simples (sem transações)

Um dos principais problemas deste capítulo é que, a não ser que um servidor seja projetado cuidadosamente, as operações executadas em nome de diferentes clientes às vezes podem interferir umas nas outras. Tal interferência pode resultar em valores incorretos nos objetos. Nesta seção, discutiremos como as operações de clientes podem ser sincronizadas sem apelar para as transações.

Operações atômicas no servidor ◊ Vimos nos capítulos anteriores que o uso de várias *threads* é vantajoso para o desempenho em muitos servidores. Também observamos que o uso de *threads* permite que operações de vários clientes sejam efetuadas concorrentemente e possivelmente acessam os mesmos objetos. Portanto, os métodos dos objetos devem ser projetados para uso em um contexto *multi-threaded*. Por exemplo, se os métodos *deposit* e *withdraw* (depósito e saque) não forem projetados para serem usados em um programa *multi-threaded* é possível que as ações de duas, ou mais, execuções concorrentes do método sejam interpostas arbitrariamente e tenham efeitos estranhos nas variáveis de instância dos objetos conta.

O Capítulo 6 explicou o uso da palavra-chave *synchronized*, que pode ser aplicada em métodos na linguagem Java para garantir que apenas uma *thread* por vez possa acessar um objeto. Em nosso

Operações da interface *Account*

deposit(amount)
deposita *amount* na conta

withdraw(amount)
saca *amount* da conta

getBalance()→ amount
retorna o saldo da conta

setBalance(amount)
configura o saldo da conta como amount

Operações da interface *Branch*

create(name)→ account
cria uma nova conta com um nome informado

lookup(name)→ account
retorna uma referência para a conta com o nome informado

branchTotal()→ amount
retorna o total de todos os saldos da agência

Figura 13.1

exemplo, a classe que implementa a interface *Account* poderá declarar os métodos como *synchronized*. Por exemplo:

```
public synchronized void deposit(int amount) throws RemoteException{
    // adiciona amount ao saldo da conta
}
```

Se uma *thread* invoca um método sincronizado em um objeto, então o acesso a esse objeto é travado (*locked*); assim, se outra *thread* invocar um de seus métodos sincronizados será bloqueada até que a trava (*lock*) seja liberada. Essa forma de sincronização obriga a execução das *threads* serem separadas no tempo e garante que as variáveis de instância de um único objeto sejam acessadas de maneira consistente. Sem a sincronização, duas invocações distintas de *deposit* poderiam ler o saldo antes de uma delas o ter incrementado, resultando em um valor incorreto. Qualquer método que acesse uma variável de instância que possa variar deve ser sincronizado.

As operações que estão livres de interferência de operações concorrentes sendo executadas por outras *threads* são chamadas de *operações atômicas*. O uso de métodos sincronizados em Java é uma maneira de obter operações atômicas. Mas em outros ambientes de programação de servidores *multi-threaded*, as operações sobre os objetos ainda precisam ter operações atômicas para manter seus objetos consistentes. Isso pode ser conseguido com o uso de qualquer mecanismo de exclusão mútua disponível, como um *mutex*.

Melhorando a cooperação dos clientes por meio da sincronização das operações do servidor ◊ Os clientes podem usar um servidor como uma maneira de compartilhar recursos. Isso é conseguido com alguns clientes usando operações para atualizar os objetos do servidor e outros clientes usando operações para acessá-los. O esquema de acesso sincronizado aos objetos, mencionado anteriormente, fornece tudo que é necessário para muitas aplicações – ele impede que as *threads* interfiram umas com as outras. Entretanto, algumas aplicações exigem uma maneira das *threads* se comunicarem entre si.

Por exemplo, pode surgir uma situação na qual a operação solicitada por um cliente não possa ser concluída até que uma operação solicitada por outro cliente tenha sido efetuada. Isso pode acontecer quando alguns clientes são produtores e outros são consumidores – talvez os consumidores tenham

que esperar até que um produtor tenha fornecido mais algumas das mercadorias em questão. Isso também pode ocorrer quando os clientes estão compartilhando um recurso – os clientes que precisam do recurso talvez tenham que esperar que outros clientes o liberem. Veremos, posteriormente neste capítulo, que uma situação semelhante surge quando bloqueios ou indicações de tempo (*timestamps*) são usados para controle de concorrência em transações.

Os métodos Java *wait* e *notify*, apresentados no Capítulo 6, permitem que as *threads* se comuniquem de uma maneira que resolve os problemas mencionados anteriormente. Eles devem ser usados dentro dos métodos sincronizados de um objeto. Uma *thread* chama *wait* em um objeto, para ficar suspenso e permitir que outra *thread* execute um método desse objeto. Uma *thread* chama *notify* para informar qualquer outra *thread* que esteja esperando nesse objeto, que ela alterou alguns de seus dados. O acesso a um objeto ainda é atômico quando as *threads* esperam umas pelas outras: uma *thread* que chama *wait* libera sua trava (*lock*) e fica suspensa como uma única ação atômica; quando for reiniciada, após ser notificada, ela deve adquirir uma nova trava sobre o objeto e retoma a execução após a espera. Uma *thread* que chama *notify* (dentro de um método sincronizado) conclui a execução desse método antes de liberar a trava sobre o objeto.

Considere a implementação de um objeto compartilhado *Queue*, com dois métodos: *first*, que remove e retorna o primeiro objeto da fila; e *append*, que adiciona um objeto dado no final da fila. O método *first* testará se a fila está vazia, no caso em que chamará *wait* na fila. Se um cliente invocar *first* quando a fila estiver vazia, ele não obterá uma resposta até que outro cliente tenha adicionado algo na fila – a operação *append* chamará *notify* quando tiver adicionado um objeto na fila. Isso permite que uma das *threads* que esteja esperando no objeto fila retome a operação e retorne o primeiro objeto da fila para seu cliente. Quando as *threads* sincronizam suas ações em um objeto por meio de *wait* e *notify*, o servidor 'segura' as requisições que não podem ser atendidas imediatamente, e o cliente espera por uma resposta até que outro cliente tenha produzido o que ele precisa.

Na seção posterior, sobre bloqueios de transações, discutiremos a implementação de travas (*locks*) como um objeto com operações sincronizadas. Quando os clientes tentam adquirir uma trava, talvez tenham que esperar até que a trava seja liberada por outros clientes.

Sem a capacidade de sincronizar *threads* dessa maneira, um cliente que não pode ser atendido imediatamente (por exemplo, um cliente que invoca o método *first* em uma fila vazia), é orientado a tentar novamente em um momento posterior. Isso é insatisfatório, pois envolverá o cliente na consulta seqüencial ao servidor, e o servidor na execução de requisições extras. Isso também é potencialmente injusto, pois outros clientes podem fazer suas requisições antes de esperar que o cliente tente novamente.

13.1.2 Modelo de falha para transações

Lampson [1981] propôs um modelo de falhas para transações distribuídas que considera falhas de discos, servidores e comunicação. Nesse modelo, a alegação é de que os algoritmos funcionam corretamente na presença de falhas previsíveis, mas nenhuma alegação é feita a respeito de seu comportamento quando ocorre um desastre. Embora possam ocorrer erros, eles podem ser detectados e tratados, antes que qualquer comportamento incorreto ocorra. O modelo diz o seguinte:

- As escritas em armazenamento permanente podem falhar – ou não gravando nada ou gravando um valor errado – por exemplo, escrever no bloco errado é um desastre. O armazenamento de arquivos também pode deteriorar. As leituras feitas no armazenamento permanente podem detectar (por meio de uma soma de verificação) quando um bloco de dados está danificado.
- Os servidores podem falhar ocasionalmente. Quando um servidor danificado é substituído por um novo processo, primeiro sua memória volátil é colocada em um estado no qual não conhece nenhum dos valores (por exemplo, de objetos) anteriores à falha. Depois disso, ele executa um procedimento de recuperação, usando informações do meio de armazenamento permanente e aquelas obtidas a partir de outros processos, para configurar os valores dos objetos, incluindo aqueles relacionados ao protocolo de efetivação (*commit*) de duas fases (veja a Seção 14.6). Quando um processo falha, ele é obrigado a parar para que seja impedido de enviar mensagens errôneas e de escrever valores errados no meio de armazenamento permanente

Hidden page

e o Arjuna [Shrivastava *et al.* 1991]. Neste último contexto, uma transação consiste na execução de uma seqüência de requisições de cliente como, por exemplo, se vê na Figura 13.2. Do ponto de vista do cliente, uma transação é uma seqüência de operações que formam um único passo, transformando os dados do servidor de um estado consistente para outro.

As transações podem ser fornecidas como parte do *middleware*. Por exemplo, o CORBA fornece a especificação de um *Object Transaction Service* (Serviço de Transação de Objeto) [OMG 2003] com interfaces IDL que permitem às transações dos clientes incluírem diversos objetos em vários servidores. São fornecidas operações para o cliente especificar o início e o fim de uma transação. O cliente ORB mantém um contexto para cada transação, o qual propaga com cada operação nessa transação. No CORBA, os objetos transacionais são invocados dentro do escopo de uma transação e geralmente tem algum meio de armazenamento persistente associado.

Em todos esses contextos, uma transação se aplica aos objetos recuperáveis e se destina a ser atômica. Freqüentemente, ela é chamada de *transação atômica* (veja o quadro a seguir). Existem dois aspectos na atomicidade:

Tudo ou nada: ou uma transação termina com êxito e os efeitos de todas as suas operações são escritos nos objetos ou (se ela falhar ou for deliberadamente cancelada) não há efeito nenhum. Essa propriedade de tudo ou nada tem mais dois aspectos próprios:

atomicidade da falha: os efeitos são atômicos mesmo quando o servidor falha;

durabilidade: após uma transação ter sido concluída com êxito, todos os seus efeitos são escritos no armazenamento permanente. Usamos o termo armazenamento permanente para nos referirmos aos arquivos mantidos no disco ou em outro meio permanente. Os dados escritos em um arquivo sobreviverão, caso o processo servidor falhe.

Isolamento: cada transação deve ser realizada sem interferência de outras transações; em outras palavras, os efeitos intermediários de uma transação não devem ser visíveis por outras transações.

Para suportar os requisitos da atomicidade da falha e da durabilidade, os objetos devem ser *recuperáveis*; quando um processo servidor falha inesperadamente, devido a uma falha de hardware, ou a um erro de software, as alterações atribuídas a todas as transações concluídas devem estar disponíveis no armazenamento permanente para que, quando o servidor for substituído por um novo processo, ele possa recuperar os objetos para refletir o efeito do tudo ou nada. Quando um servidor reconhece o término da transação de um cliente, todas as alterações da transação feitas nos objetos devem ter sido escritas no armazenamento permanente.

Propriedades ACID ◊ Härder e Reuter [1983] sugeriram o mnemônico ACID para lembrar das propriedades das transações, como segue:

Atomicidade: uma transação deve ser do tipo tudo ou nada;

Consistência: uma transação leva o sistema de um estado consistente para outro estado consistente;

Isolamento;

Durabilidade.

Não incluímos a consistência em nossa lista de propriedades das transações porque geralmente os programadores de servidores e clientes são responsáveis por garantir que as transações deixem o banco de dados consistente.

Como exemplo de consistência, suponha que no exemplo de transações bancárias um objeto contenha a soma dos saldos de todas as contas e seu valor seja usado como resultado de *branchTotal*. Os clientes podem obter a soma dos saldos de todas as contas usando *branchTotal*, ou chamando *getBalance* em cada uma das contas. Por consistência, eles devem obter o mesmo resultado a partir de ambos os métodos. Para manter essa consistência, as operações *deposit* e *withdraw* devem atualizar o objeto que contém a soma dos saldos de todas as contas.

Um servidor que suporta transações deve sincronizar as operações o suficiente para garantir que o requisito do isolamento seja satisfeito. Uma maneira de fazer isso é executar as transações em série – uma por vez, em alguma ordem arbitrária. Infelizmente, essa solução geralmente seria inaceitável para servidores cujos recursos são compartilhados por vários usuários interativos. Em nosso exemplo de transações bancárias, é desejável permitir que vários funcionários do banco realizem transações bancárias on-line, ao mesmo tempo que os outros.

O objetivo de qualquer servidor que suporte transações é maximizar a concorrência. Portanto, as transações podem ser executadas concomitantemente, caso tenham o mesmo efeito que uma execução serial – isto é, elas são *equivalentes em série* ou *serializáveis*.

Recursos para transação podem ser adicionados aos servidores de objetos recuperáveis. Cada transação é criada e gerenciada por um coordenador, o qual implementa a interface *Coordinator*, mostrada na Figura 13.3. O coordenador fornece a cada transação um identificador ou TID. O cliente invoca o método *openTransaction* do coordenador para introduzir uma nova transação – um identificador de transação, ou TID, é alocado e retornado. No final de uma transação, o cliente invoca o método *closeTransaction* para indicar seu final – todos os objetos recuperáveis acessados pela transação devem ser salvos. Se, por algum motivo, o cliente quiser cancelar uma transação, ele invoca o método *abortTransaction* – todos os seus efeitos devem desaparecer.

Uma transação é obtida pela cooperação entre um programa cliente, alguns objetos recuperáveis e um coordenador. O cliente especifica a seqüência de invocações nos objetos recuperáveis que devem compreender uma transação. Para obter isso, o cliente envia com cada invocação o identificador de transação retornado por *openTransaction*. Uma maneira de tornar isso possível é incluir um argumento extra em cada operação de um objeto recuperável para transportar o TID. Por exemplo, no serviço de transação bancária, a operação *deposit* poderia ser definida como:

deposit(trans, amount)
deposita *amount* na conta para a transação com TID *trans*

Quando as transações são fornecidas como *middleware*, o TID pode ser passado implicitamente com todas as invocações remotas entre *openTransaction* e *closeTransaction* ou *abortTransaction*. É isso que o serviço de transação do CORBA faz. Não vamos mostrar TIDs em nossos exemplos.

Normalmente, uma transação termina quando o cliente faz uma requisição de *closeTransaction*. Se a transação progrediu normalmente, a resposta diz que ela foi *efetivada* – isso constitui um compromisso para o cliente de que todas as alterações solicitadas na transação foram escritas permanentemente, e que as transações futuras, que acessem os mesmos dados, verão os resultados de todas as alterações feitas durante a transação.

Como alternativa, talvez a transação tenha que ser cancelada por um de vários motivos relacionados à natureza da transação em si, a conflitos com outra transação ou à falha de um processo ou de um computador. Quando uma transação é cancelada, as partes envolvidas (os objetos recuperáveis e o coordenador) devem garantir que nenhum de seus efeitos seja visível para as futuras transações, ou nos objetos, ou em suas cópias no meio de armazenamento permanente.

Uma transação é bem-sucedida, ou é cancelada de uma de duas maneiras – o cliente a cancela (usando uma chamada de *abortTransaction* para o servidor) ou o servidor a cancela. A Figura 13.4

openTransaction( )→ trans;
inicia uma nova transação e gera um TID *trans* exclusivo. Esse identificador será usado nas outras operações da transação.

closeTransaction(trans) → (commit, abort);
termina uma transação: um valor de retorno *commit* indica que a transação foi efetivada; um valor de retorno *abort* indica que ela foi cancelada.

abortTransaction(trans);
cancela a transação.

Figura 13.3 Operações na interface *Coordinator*.

mostra essas três possibilidades alternativas das transações. Nesses dois casos, nos referimos a uma transação como *falha*.

Ações de serviço relacionadas a falhas de processo ◊ Se um processo servidor falha inesperadamente, ele acaba por ser substituído. O novo processo servidor cancela todas as transações não efetivadas e usa um procedimento de recuperação para restaurar os valores dos objetos com os valores produzidos pela transação efetivada mais recentemente. Para tratar com um cliente que falhou inesperadamente durante uma transação, os servidores podem fornecer um tempo de expiração a cada transação e cancelar toda transação que não tiver terminado antes de seu tempo de expiração.

Ações de cliente relacionadas a falhas do processo servidor ◊ Se um servidor falhar enquanto uma transação está em andamento, o cliente saberá disso quando uma das operações retornar uma exceção, após um tempo limite. Se um servidor falhar, e depois, durante o andamento de uma transação for substituído, a transação não será mais válida e o cliente deverá ser informado como uma exceção para a próxima operação. Em qualquer caso, o cliente deverá então formular um plano, possivelmente consultando o usuário humano, para a conclusão, ou cancelamento, da tarefa da qual a transação fazia parte.

13.2.1 Controle de concorrência

Esta seção ilustra dois conhecidos problemas de transações concorrentes no contexto do exemplo de transações bancárias – o problema da "atualização perdida" e o problema das "recuperações inconsistentes". Esta seção mostrará então como esses dois problemas podem ser evitados com o uso de execuções equivalentes em série das transações. Supomos em toda parte que cada uma das operações *deposit*, *withdraw*, *getBalance* e *setBalance* é sincronizada – isto é, seus efeitos sobre a variável de instância que altera o saldo de uma conta são atômicos.

O problema da atualização perdida ◊ O problema da atualização perdida é ilustrado pelas duas transações a seguir nas contas bancárias *A*, *B* e *C*, cujos saldos iniciais são de \$100, \$200 e \$300 respectivamente. A transação *T* transfere um valor da conta *A* para a conta *B*. A transação *U* transfere um valor da conta *C* para a conta *B*. Nos dois casos, o valor transferido é calculado de forma a aumentar o saldo de *B* em 10%. O efeito líquido da execução das transações *T* e *U* sobre a conta *B* deve ser o aumento do seu saldo em 10%, duas vezes; portanto, seu valor final será de \$242.

Agora, considere os efeitos de permitir que as transações *T* e *U* sejam executadas concorrentemente, como na Figura 13.5. As duas transações obtêm o saldo de *B* como \$200 e depois depositam \$20. O resultado é incorreto, aumentando o saldo da conta *B* por \$20, em vez de \$42. Essa é uma ilustração do problema da atualização perdida. A atualização de *U* é perdida porque *T* a sobrescreve sem vê-la. As duas transações lêem o valor antigo, antes que qualquer uma grave o novo valor.

Bem sucedida	*Cancelada pelo cliente*	*Cancelada pelo servidor*	
openTransaction	*openTransaction*		*openTransaction*
operação	*operação*		*operação*
operação	*operação*		*operação*
•	•	o servidor cancela	•
•	•	transação →	•
operação	*operação*		*ERRO de operação relatado ao cliente*
closeTransaction	*abortTransaction*		

Figura 13.4 Alternativas para realização de transações.

Transação T:		*Transação U:*	
balance = b.getBalance();		*balance = b.getBalance();*	
*b.setBalance(balance*1.1);*		*b.setBalance(balance*1.1);*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

Figura 13.5 O problema da atualização perdida.

Da Figura 13.5 em diante, mostramos as operações que afetam o saldo de uma conta em sucessivas linhas na página e o leitor deve supor que uma operação em uma linha específica é executada depois da que está na linha acima dela.

Recuperações inconsistentes ◊ A Figura 13.6 mostra outro exemplo relacionado a uma conta bancária, no qual a transação *V* transfere uma quantia da conta *A* para *B* e a transação *W* ativa o método *branchTotal* para obter a soma dos saldos de todas as contas do banco. Os saldos das duas contas bancárias, *A* e *B*, são ambos inicialmente iguais a $200. O resultado de *branchTotal* inclui a soma de *A* e *B* como $300, o que está errado. Essa é uma ilustração do problema das recuperações inconsistentes. As recuperações de *W* são inconsistentes porque *V* realizou apenas a parte do saque de uma transferência, no momento em que a soma foi calculada.

Equivalência serial ◊ Se for sabido que cada uma de várias transações tem o efeito correto quando executada sozinha, então podemos inferir que, se essas transações forem executadas uma por vez, em alguma ordem, o efeito combinado também será correto. Uma interposição das operações das transações em que o efeito combinado é igual ao que seria se as transações tivessem sido executadas uma por vez, em alguma ordem, é uma interposição equivalente em série. Quando dizemos que duas transações diferentes têm o *mesmo efeito*, queremos dizer que as operações de leitura retornam os mesmos valores e que as variáveis de instância dos objetos têm os mesmos valores no final.

O uso da equivalência serial como critério para uma execução concorrente correta evita a ocorrência de atualizações perdidas e recuperações inconsistentes.

Transação V:		*Transação W:*	
a.withdraw(100)		*aBranch.branchTotal()*	
b.deposit(100)			
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total + b.getBalance()	$300
		total = total + c.getBalance()	
b.deposit(100)	$300	•	
•			

Figura 13.6 O problema das recuperações inconsistentes.

Hidden page

write altera seu valor. O *efeito* de uma operação se refere ao valor de um objeto configurado por uma operação *de escrita* e ao resultado retornado por uma operação *de leitura*. As regras de conflito para as operações *de leitura* e *escrita* aparecem na Figura 13.9.

Para quaisquer duas transações, é possível determinar a ordem dos pares de operações conflitantes nos objetos acessados por ambas. A equivalência serial pode ser definida em termos de conflitos de operação, como segue:

> Para que duas transações sejam *equivalentes em série*, é necessário e suficiente que todos os pares de operações conflitantes das duas transações sejam executados na mesma ordem em todos os objetos que ambas acessam.

Considere como exemplo as transações T e U, definidas como segue:

T: x = read(i); write(i, 10); write(j, 20);

U: y = read(j); write(j, 30); z = read (i);

Então, considere a interposição de suas execuções, mostrada na Figura 13.10. Note que o acesso de cada transação aos objetos i e j se dá em série com relação um ao outro, pois T faz todos os seus acessos a i antes que U o faça, e U faz todos os seus acessos a j antes que T o faça. Mas a ordem não é equivalente em série, pois os pares de operações conflitantes não são executados na mesma ordem nos dois objetos. As ordens equivalentes em série exigem uma das duas condições a seguir:

1. T acessa i antes de U e T acessa j antes de U.
2. U acessa i antes de T e U acessa j antes de T.

A equivalência serial é usada como critério para a derivação de protocolos de controle de concorrência. Esses protocolos tentam dispor as transações em série no acesso aos objetos. Três estratégias de controle de concorrência alternativas são comumente usadas: bloqueio, controle de concorrência otimista e ordenação de indicação de tempo (*timestamp*) Entretanto, a maioria dos sistemas práticos usa o bloqueio, o qual será discutido na Seção 13.4. Quando o bloqueio é usado, o servidor configura

Operações de diferentes transações		*Conflito*	*Motivo*
leitura	*leitura*	Não	Porque o efeito de duas operações *de leitura* não depende da ordem em que elas são executadas
leitura	*escrita*	Sim	Porque o efeito de uma operação *de leitura* e de uma operação *de escrita* depende da ordem de sua execução
escrita	*escrita*	Sim	Porque o efeito de duas operações *de escrita* depende da ordem de sua execução

Figura 13.9 Regras de conflito das operações de *leitura* e *escrita*.

Transação T:	*Transação U:*
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Figura 13.10 Uma interposição não equivalente em série de operações das transações T e U.

Hidden page

Transação T:		*Transação U:*	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

Figura 13.11 Uma leitura suja quando a transação *T* é cancelada.

das. O cancelamento destas últimas transações pode fazer com que ainda mais transações sejam canceladas. Tais situações são chamadas de *cancelamentos em cascata*. Para evitar os cancelamentos em cascata, as transações só podem ler objetos que foram escritos por transações efetivadas. Para garantir que isso ocorra, toda operação de leitura deve ser retardada até que as outras transações que aplicaram uma operação de escrita sobre o mesmo objeto tenham sido efetivadas ou canceladas. O fato de evitar cancelamentos em cascata é uma condição mais importante do que a capacidade de recuperação.

Escritas prematuras ◊ Considere outra implicação da possibilidade de uma transação ser cancelada. Esta relacionada à interação entre operações *de escrita* no mesmo objeto, pertencentes a diferentes transações. Como ilustração, consideremos duas transações *setBalance*, *T* e *U*, na conta *A*, como se vê na Figura 13.12. Antes das transações, o saldo da conta *A* era de $100. As duas execuções são equivalentes em série, com *T* determinando o saldo como $105 e *U* determinando como $110. Se a transação *U* for cancelada e *T* efetivada, o saldo deverá ser de $105.

Alguns sistemas de banco de dados implementam a ação de *cancelamento* restaurando *imagens de antes* de todas as *escritas* de uma transação. Em nosso exemplo, *A* tem inicialmente $100, que é a imagem de antes da escrita de *T*; de modo semelhante $105 é a imagem de antes da escrita de *U*. Assim, se *U* for cancelada, obteremos o saldo correto de $105.

Agora, considere o caso de quando *U* é efetivada e depois *T* é cancelada. O saldo deve ser de $110, mas como a imagem de antes da *escrita* de *T* é $100, obtemos o saldo errado de $100. Analogamente, se *T* for cancelada, e depois *U* for cancelada, a imagem de antes da *escrita* de *U* será $105 e obteremos o saldo errado de $105 – o saldo deve voltar para $100.

Para garantir resultados corretos em um esquema de recuperação que utiliza imagens de antes, as operações *de escritas* devem ser retardadas até que as transações anteriores que atualizaram os mesmos objetos tenham sido efetivadas ou canceladas.

Execuções restritas de transações ◊ Geralmente, é exigido que as transações retardem tanto as operações *de leitura* como *de escrita*, para evitar leituras sujas e escritas prematuras. As execuções das transações são chamadas de *restritas* se o serviço atrasar tanto as operações *de leitura* como *de*

Transação T:		*Transação U:*	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

Figura 13.12 Sobreposição de valores não efetivados.

Hidden page

dos destinatários. Se uma ou mais das subtransações falhar, a transação ascendente poderia registrar o fato, e depois ser efetivada, com o resultado de que todas as transações descendentes bem-sucedidas seriam efetivadas. Então, ela poderia iniciar outra transação para tentar enviar as mensagens que não foram enviadas na primeira vez.

Quando precisamos distinguir nossa forma de transação original das transações aninhadas, usamos o termo transação *plana*. Ela é plana porque todo o seu trabalho é feito no mesmo nível entre uma transação *openTransaction* e uma *efetivação* ou um *cancelamento*, e não é possível efetivar ou cancelar partes dela. As transações aninhadas têm as seguintes vantagens principais:

1. As subtransações de um nível (e suas descendentes) podem ser executadas concorrentemente com outras subtransações de mesmo nível na hierarquia. Isso pode possibilitar um nível maior de concorrência em uma transação. Quando as subtransações são executadas em servidores diferentes, elas podem trabalhar em paralelo. Por exemplo, considere a operação *branchTotal* em nosso exemplo de transações bancárias. Ela pode ser implementada pela invocação de *getBalance* em cada conta da agência. Agora, cada uma dessas invocações pode ser executada como uma subtransação, no caso em que elas podem ser executadas concorrentemente. Como cada operação se aplica a uma conta diferente, não existirão operações conflitantes entre as subtransações.
2. As subtransações podem ser efetivadas ou canceladas independentemente. Em comparação com uma única transação, um conjunto de subtransações aninhadas é potencialmente mais robusto. O exemplo anterior de envio de correspondência mostra que isso é verdade – com uma transação plana, uma falha de transação causaria o reinício da transação inteira. Na verdade, uma transação ascendente pode decidir sobre diferentes ações, de acordo com o fato de uma subtransação ter sido cancelada ou não.

As regras de efetivação das transações aninhadas são bastante refinadas:

- Uma transação só pode ser efetivada ou cancelada depois que suas transações descendentes tiverem sido concluídas.
- Quando uma subtransação é concluída, ela toma uma decisão independente de ser efetivada provisoriamente ou ser cancelada. Sua decisão de cancelar é final.
- Quando uma transação ascendente é cancelada, todas as suas subtransações são canceladas. Por exemplo, se T_2 é cancelada, então T_{21} e T_{211} também devem ser canceladas, mesmo que possam ter sido efetivadas provisoriamente.
- Quando uma subtransação é cancelada, a transação ascendente pode decidir se vai ser cancelada ou não. Em nosso exemplo, T decide ser efetivada, embora T_2 tenha sido cancelada.
- Se a transação de nível superior é efetivada, então todas as subtransações que foram efetivadas provisoriamente também podem ser efetivadas, desde que nenhuma de suas ascendentes tenha sido cancelada. Em nosso exemplo, a efetivação de T permite que T_1, T_{11} e T_{12} sejam efetivadas, mas não T_{21} e T_{211}, pois sua ascendente T_2 foi cancelada. Note que os efeitos de uma subtransação não são permanentes até que a transação de nível superior seja efetivada.

Em alguns casos, a transação de nível superior pode decidir ser cancelada porque uma ou mais de suas subtransações foi cancelada. Como exemplo, considere a seguinte transação *Transferência*:

Transferência de \$100 de *B* para *A*

a.deposit(100)

b.withdraw(100)

Isso pode ser estruturado como um par de subtransações, uma para a operação *withdraw* e a outra para *deposit*. Quando as duas subtransações forem efetivadas, a transação *Transferência* também poderá ser efetivada. Suponha que uma subtransação *withdraw* seja cancelada quando uma conta não tiver fundos. Agora, considere o caso de quando a subtransação *withdraw* é cancelada e a subtransação *deposit* é efetivada – e lembre-se de que a efetivação de uma transação descendente está condicionada à efetivação da transação ascendente. Presumimos que a transação de nível superior (*Transferência*) de-

Hidden page

Hidden page

2. Se uma transação T já executou uma operação *de escrita* sobre um objeto em particular, então uma transação concorrente U não deve *ler* nem *escrever* esse objeto até que T seja efetivada ou cancelada.

Para impor (1), um pedido para uma trava de escrita sobre um objeto é atrasado pela presença de uma trava de leitura pertencente à outra transação. Para impor (2), um pedido para uma trava de leitura, ou para uma trava de escrita, sobre um objeto é atrasado pela presença de uma trava de escrita pertencente à outra transação.

A Figura 13.15 mostra a compatibilidade das travas de leitura e de escrita sobre qualquer objeto específico. As entradas à esquerda da primeira coluna na tabela mostram o tipo de trava já alocada, se houver. As entradas acima da primeira linha mostram o tipo de trava solicitada. A entrada em cada célula mostra o efeito sobre uma transação que solicita o tipo de trava dado acima, quando o objeto tiver sido bloqueado em outra transação com o tipo de trava da esquerda.

As recuperações inconsistentes e atualizações perdidas são causadas pelo conflito entre operações *de leitura* em uma transação e operações de *escrita* em outra, sem a proteção de um esquema de controle de concorrência como o bloqueio por travas. As recuperações inconsistentes são evitadas executando-se a transação de recuperação antes ou depois da transação de atualização. Se a transação de recuperação vier primeiro, suas travas de leitura atrasarão a transação de atualização. Se ela vier depois, seu pedido de trava de leitura a fará ser retardada até que a transação de atualização tenha terminado.

Atualizações perdidas ocorrem quando duas transações lêem um valor de um objeto e depois o utilizam para calcular um novo valor. As atualizações perdidas são evitadas fazendo-se transações posteriores retardar suas leituras até que as anteriores tenham terminado. Isso é conseguido em cada transação configurando uma trava de leitura ao ler um objeto e depois *promovendo-o* a uma trava de escrita, ao escrever o mesmo objeto – quando uma transação subseqüente exigir uma trava de leitura, ela será retardada até que a transação corrente tenha terminado.

Uma transação com uma trava de leitura compartilhada com outras transações não pode promover sua trava de leitura para um trava de escrita, pois esta última entraria em conflito com as travas de leitura mantidas pelas outras transações. Portanto, tal transação deve solicitar uma trava de gravação e esperar que as outras travas de leitura sejam liberadas.

A promoção de travas se refere à conversão de uma trava em uma outra mais forte – isto é, em uma trava mais exclusiva. A tabela de compatibilidade de travas mostra quais delas são mais ou menos exclusivas. A trava de leitura permite outras travas de leitura, enquanto a trava de escrita, não. E também não permite outras travas de escrita. Portanto, uma trava de escrita é mais exclusiva do que uma trava de leitura. As travas podem ser promovidas porque o resultado é uma trava mais exclusiva. Não é seguro rebaixar uma trava mantida por uma transação, antes que ela seja efetivada, pois o resultado será mais permissivo do que o anterior e pode possibilitar execuções de outras transações que sejam inconsistentes com a equivalência serial.

As regras para o uso de travas em uma implementação de bloqueio de duas fases restrito estão resumidas na Figura 13.16. Para garantir que essas regras sejam obedecidas, o cliente não tem acesso às operações de bloqueio ou desbloqueio de itens de dados. O bloqueio é realizado quando os pedidos de operações *de leitura* e *de escrita* estão para ser aplicados aos objetos recuperáveis e o desbloqueio é realizado pelas operações de *efetivação* ou *cancelamento* do coordenador de transação.

Para um objeto		*Trava solicitada*	
		leitura	*escrita*
Trava já alocada	*nenhum*	OK	OK
	leitura	OK	espera
	escrita	espera	espera

Figura 13.15 Compatibilidade de travas.

1. Quando uma operação acessa um objeto dentro de uma transação:
   (a) Se o objeto ainda não estiver travado, ele será travado e a operação prosseguirá.
   (b) Se o objeto tiver uma trava conflitando com a configurada por outra transação, a transação deverá esperar até que ela seja liberada.
   (c) Se o objeto tiver uma trava não-conflitante com a configurada por outra transação, a trava será compartilhada e a operação prosseguirá.
   (d) Se o objeto já tiver sido travado na mesma transação, a trava será promovida, se necessário, e a operação prosseguirá. (Onde a promoção é impedida por uma trava conflitante, será usada a regra (b).)
2. Quando uma transação é efetivada ou cancelada, o servidor destrava todos os objetos que travou para a transação.

Figura 13.16 Uso de travas no bloqueio de duas fases restrito.

Por exemplo, o *Concurrency Control Service* do CORBA [OMG 2000b] pode ser usado para aplicar controle de concorrência em nome de transações ou para proteger objetos sem usar transações. Ele fornece uma maneira de associar uma coleção de travas (chamada de *conjunto de travas*) a um recurso como um objeto recuperável. Um conjunto de travas permite que as travas sejam adquiridas ou liberadas. O método *lock* de um conjunto de travas adquirirá uma trava, ou um bloco, até que a trava seja liberada; outros métodos permitem que as travas sejam promovidas ou liberadas. Os conjuntos de travas transacionais suportam os mesmos métodos que os conjuntos de travas, mas seus métodos exigem identificadores de transação como argumentos. Mencionamos anteriormente que o serviço de transação CORBA rotula todos os pedidos de cliente de uma transação com o identificador de transação. Isso permite que uma trava conveniente seja adquirida, antes que cada um dos objetos recuperáveis seja acessado durante uma transação. O coordenador de transação é responsável por liberar as travas quando uma transação é efetivada ou cancelada.

As regras dadas na Figura 13.16 garantem o rigor, pois as travas são mantidas até que uma transação tenha sido efetivada ou cancelada. Entretanto, não é necessário manter travas de leitura para garantir o rigor. As travas de leitura só precisam ser mantidas até que chegue o pedido para efetivar ou cancelar.

Implementação de travas ◊ O funcionamento de travas e de bloqueio será implementado por um objeto separado no servidor, que chamamos de *gerenciador de travas*. O gerenciador de travas mantém um conjunto de travas, por exemplo, em uma tabela de *hashing*. Cada trava é uma instância da classe *Lock* e é associado a um objeto em particular. A classe *Lock* aparece na Figura 13.17. Cada instância de *Lock* mantém as seguintes informações em suas variáveis de instância:

- o identificador do objeto bloqueado;
- os identificadores das transações que correntemente mantêm as travas (as travas compartilhadas podem ter vários proprietários);
- um tipo de trava.

Os métodos de *Lock* são sincronizados para que as *threads* que estão tentando adquirir ou liberar uma trava não interfiram umas com as outras. Mas, além disso, tentativas de adquirir a trava usam o método *wait* quando precisam esperar que outra *thread* a libere.

O método *acquire* executa as regras dadas na Figura 13.15 e na Figura 13.16. Seus argumentos especificam um identificador de transação e o tipo de trava exigido por essa transação. Ele testa se o pedido pode ser atendido. Se outra transação possui a trava em um modo conflitante, ela ativa *wait*, o que faz a *thread* do chamador ser suspensa até um método *notify* correspondente. Note que o método *wait* é englobado em uma instrução *while*, pois todas as transações que estão esperando são notificadas e algumas delas podem não ser capazes de

```
public class Lock {
    private Object object;          // o objeto que está sendo protegido pela trava
    private Vector holders;         // os TIDs dos proprietários correntes
    private LockType lockType;      // o tipo corrente

    public synchronized void acquire(TransID trans, LockType aLockType ){
        while(/*outra transação possui a trava em modo conflitante*/) {
            try {
                wait();
            }catch ( InterruptedException e){/*...*/ }
        }
        if (holders.isEmpty()) { // nenhum TID mantém travas
            holders.addElement(trans);
            lockType = aLockType;
        } else if (/*outra transação possui a trava e a compartilha*/ ) ){
            if (/* esta transação não é proprietária*/) holders.addElement(trans);
        } else if (/* esta transação é proprietária, mas precisa de uma trava mais exclusiva*/)
                lockType.promote();
        }
    }
    public synchronized void release(TransID trans){
        holders.removeElement(trans); // remove este proprietário
        // configura o tipo de trava como nenhum
        notifyAll();
    }
}
```

Figura 13.17 Classe *Lock*.

prosseguir. Quando finalmente a condição é satisfeita, o restante do método configura a trava apropriadamente:

- se nenhuma outra transação possui a trava, apenas adiciona a transação dada no grupo de proprietários e configura o tipo;
- se outra transação possui a trava, a compartilha, adicionando a transação dada no grupo de proprietários (a não ser que ela já seja proprietária);
- se essa transação é proprietária, mas está solicitando uma trava mais exclusiva, promove a trava.

Os argumentos do método *release* especificam o identificador da transação que está liberando a trava. Ele remove o identificador de transação dos proprietários, configura o tipo de trava como *nenhum* e chama o método *notifyAll*. O método notifica todas as *threads* que estão esperando, para o caso de haver várias transações esperando para adquirir travas de leitura, situação na qual todas elas podem ser capazes de prosseguir.

A classe *LockManager* aparece na Figura 13.18. Todos os pedidos para configurar travas e para liberá-las em nome de transações são enviados para uma instância de *LockManager*.

- Os argumentos do método *setLock* especificam o objeto que a transação dada deseja bloquear e o tipo de trava. Ele encontra uma trava para esse objeto em sua tabela de *hashing* ou, se necessário, cria uma. Então, ele invoca o método *acquire* dessa trava.
- O argumento do método *unLock* especifica a transação que está liberando suas travas. Ele encontra todas as travas na tabela de *hashing* que tem como proprietária a transação dada. Para cada uma, ele chama o método *release*.

```
public class LockManager {
    private Hashtable theLocks;

    public void setLock(Object object, TransID trans, LockType lockType){
        Lock foundLock;
        synchronized(this){
                // localiza a trava associada ao objeto
                // se não houver nenhuma, cria e a adiciona na tabela de hashing
            }
        foundLock.acquire(trans, lockType);
    }

    // sincroniza este, pois queremos remover todas as entradas
    public synchronized void unLock(TransID trans) {
        Enumeration e = theLocks.elements();
        while(e.hasMoreElements()){
            Lock aLock = (Lock)(e.nextElement());
            if(/* trans é um proprietário desta trava*/ ) aLock.release(trans);
        }
    }
}
```

Figura 13.18 Classe *LockManager*.

Algumas questões de política: note que, quando várias *threads* esperam no mesmo item bloqueado, a semântica de *wait* garante que cada transação tenha sua vez. No programa anterior, as regras de conflito permitem que os proprietários de uma trava sejam vários leitores ou um escritor. A chegada de um pedido de trava de leitura é sempre atendida, a não ser que o proprietário tenha uma trava de escrita. Você, leitor, é convidado a considerar o seguinte:

- Qual é a conseqüência das transações *de escrita* na presença de um fluxo constante de pedidos de travas de leitura? Pense em uma implementação alternativa.

Quando o proprietário tem uma trava de escrita, vários leitores e gravadores podem estar esperando. Você, leitor, deve considerar o efeito de *notifyAll* e pensar em uma implementação alternativa. Se o proprietário de uma trava de leitura tentar promover a trava, quando ela é compartilhada, será impedido. Existe uma solução para essa dificuldade?

Regras de bloqueio para transações aninhadas ◊ O objetivo de um esquema de bloqueio para transações aninhadas é dispor em série o acesso aos objetos, para que:

1. cada conjunto de transações aninhadas seja uma entidade única que deve ser impedida de observar os efeitos parciais de qualquer outro conjunto de transações aninhadas;
2. cada transação dentro de um conjunto de transações aninhadas deve ser impedida de observar os efeitos parciais das outras transações do conjunto.

A primeira regra é imposta fazendo-se com que toda trava adquirida por uma subtransação bem-sucedida seja *herdada* por sua ascendente, quando for concluída. As travas herdadas também são herdadas pelas ancestrais. Note que essa forma de herança passa da descendente para a ascendente! A transação de nível superior finalmente herda todas as travas que foram adquiridas por subtransações bem-sucedidas, em qualquer profundidade de uma transação aninhada. Isso garante que as travas possam ser mantidas até que a transação de nível superior tenha sido efetivada ou cancelada, o que impede que membros de diferentes conjuntos de transações aninhadas observem os efeitos parciais uns dos outros.

A segunda regra é imposta como segue:

- as transações ascendentes não podem ser executadas concorrentemente com suas transações descendentes. Se uma transação ascendente tiver uma trava sobre um objeto, ela manterá a

trava durante o período em que sua transação descendente estiver executando. Isso significa que a transação descendente adquire temporariamente a trava de sua ascendente, pelo tempo de sua duração;

- as subtransações no mesmo nível podem ser executadas concorrentemente; portanto, quando elas acessam os mesmos objetos, o esquema de bloqueio deve organizar seus acessos em série.

As regras a seguir descrevem a aquisição e a liberação de travas:

- para que uma subtransação adquira uma trava de leitura sobre um objeto, nenhuma outra transação ativa pode ter uma trava de escrita sobre esse objeto e as únicas detentoras de uma trava de bloqueio são suas ancestrais;
- para que uma subtransação adquira uma trava de escrita sobre um objeto, nenhuma outra transação ativa pode ter uma trava de leitura ou escrita sobre esse objeto e as únicas detentoras de travas de leitura e escrita sobre esse objeto são suas ancestrais;
- quando uma subtransação é efetivada, suas travas são herdadas por sua ascendente, permitindo que esta mantenha as travas no mesmo modo que a descendente;
- quando uma subtransação é cancelada, suas travas são descartadas. Se a ascendente já possui as travas, ela pode continuar a mantê-las.

Note que as subtransações no mesmo nível, que acessam o mesmo objeto, terão sua oportunidade de adquirir as travas mantidas por suas ascendentes. Isso garante que os acessos a um objeto comum sejam organizados em série.

Como exemplo, suponha que as subtransações T_1, T_2 e T_{11}, na Figura 13.13, acessem um objeto comum, o qual não é acessado pela transação de nível superior T. Suponha que a subtransação T_1 seja a primeira a acessar o objeto, e que tenha êxito ao adquirir uma trava, o qual ela passa para T_{11} pelo tempo da duração de sua execução, obtendo-a de volta quando T_{11} termina. Quando T_1 conclui sua execução, a transação de nível superior T herda a trava, a qual a mantém até que o conjunto de transações aninhadas termine. A subtransação T_2 pode adquirir a trava de T pelo tempo da duração de sua execução.

13.4.1 Impasses

O uso de travas pode levar ao impasse (*deadlocks*). Considere o uso de travas mostrado na Figura 13.19. Como os métodos *deposit* e *withdraw* são atômicos, os mostramos adquirindo travas de escrita – embora, na prática, eles leiam o saldo e depois o escrevem. Cada um deles adquire uma trava sobre uma conta e, depois, ao tentar acessar a conta que o outro bloqueou, é impedido. Essa é uma situação de impasse – duas transações estão esperando e cada uma depende da outra liberar uma trava para que possa ser retomada.

Transação T		*Transação U*	
Operações	**Travas**	**Operações**	**Travas**
a.deposit(100);	trava de escrita em *A*		
		b.deposit(200)	trava de escrita em *B*
b.withdraw(100)			
•••	espera pela trava de *U* em *B*	*a.withdraw(200);*	espera pela trava de *T* em *A*
		•••	
•••		•••	
•••		•••	

Figura 13.19 Impasse com travas de escrita.

O impasse é uma situação particularmente comum quando os clientes estão envolvidos em um programa interativo, pois em um programa interativo, uma transação pode durar um longo período de tempo, resultando em muitos objetos sendo bloqueados e permanecendo assim, impedindo com isso que outros clientes os utilizem.

Note que o bloqueio de subitens em objetos estruturados pode ser útil para evitar conflitos e possíveis situações de impasse. Por exemplo, em uma agenda, um dia poderia ser estruturado como um conjunto de repartições de tempo, cada uma das quais podendo ser bloqueada independentemente da atualização. Os esquemas de bloqueio hierárquicos são úteis se a aplicação exigir uma granularidade diferente de bloqueio para diferentes operações. Veja a Seção 13.4.2.

Definição de impasse ◊ O impasse é um estado no qual cada membro de um grupo de transações está esperando que algum outro membro libere uma trava. Um grafo de "espera por" pode ser usado para representar os relacionamentos de espera entre transações correntes. Em um grafo de "espera por", os nós representam transações e as setas representam os relacionamentos de "espera por" entre as transações – existe uma seta do nó T para o nó U, quando a transação T está esperando que a transação U libere uma trava. Veja a Figura 13.20, que ilustra o grafo de "espera por" correspondente à situação de impasse ilustrada na Figura 13.19. Lembre-se de que o impasse surgiu porque as transações T e U tentaram adquirir um objeto mantido pela outra. Portanto T espera por U e U espera por T. A dependência entre as transações é indireta – via uma dependência nos objetos. O diagrama da direita mostra os objetos mantidos e esperados pelas transações T e U. Como cada transação pode esperar pelo único objeto, os objetos podem ser omitidos do grafo de "espera por" – produzindo o grafo simples da esquerda.

Suponha que, como aparece na Figura 13.21, um grafo de "espera por" contenha um ciclo $T \rightarrow U \rightarrow \ldots \rightarrow V \rightarrow T$; então, cada transação está esperando pela próxima transação no ciclo. Todas essas transações estão paradas, esperando pelas travas. Nenhuma das travas pode ser liberada e as transações estão em um impasse. Se uma das transações em um ciclo for cancelada, então suas travas serão liberadas e esse ciclo será quebrado. Por exemplo, se a transação T na Figura 13.21 for cancelada, ela liberará uma trava sobre um objeto pelo qual V está esperando – e V não estará mais esperando por T.

Agora, considere um cenário onde as três transações T, U e V compartilham uma trava de leitura sobre um objeto C, a transação W mantém uma trava de escrita sobre o objeto B, no qual a transação V está esperando para obter uma trava, como mostrado à direita na Figura 13.22. Então, as transações T e W solicitam travas de escrita sobre o objeto C e surge uma situação de impasse, na qual T espera por U

Figura 13.20 O grafo de "espera por" da Figura 13.19.

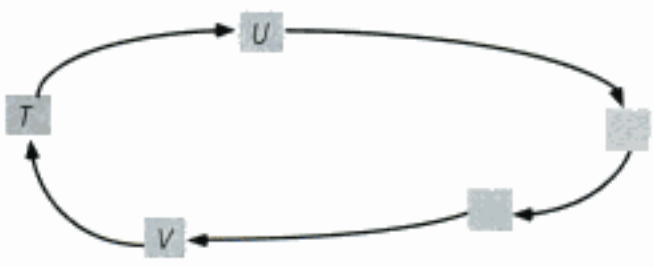


Figura 13.21 Um ciclo em um grafo de "espera por".

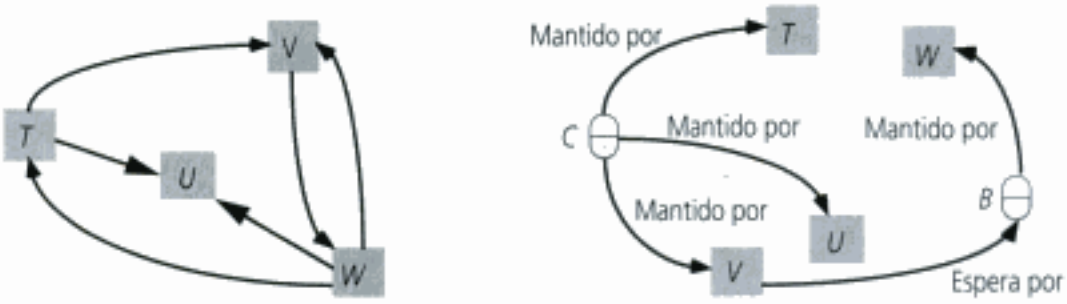


Figura 13.22 Outro grafo de "espera por".

e V, V espera por W e W espera por T, U e V, como mostrado à esquerda na Figura 13.22. Isso mostra que, embora cada transação possa esperar por apenas um objeto por vez, ela pode estar envolvida em vários ciclos. Por exemplo, a transação V está envolvida nos ciclos: $V \rightarrow W \rightarrow T \rightarrow V$ e $V \rightarrow W \rightarrow V$.

Nesse exemplo, suponha que a transação V seja cancelada. Isso liberará a trava de V sobre C e os dois ciclos envolvendo V serão quebrados.

Prevenção de impasses ◊ Uma solução é evitar os impasses. Uma maneira aparentemente simples, mas não muito boa, de superar o impasse é adquirir travas de todos os objetos usados por uma transação, quando ela inicia. Isso precisaria ser feito como um único passo atômico, para evitar um impasse nesse estágio. Tal transação não pode entrar em um impasse com outras transações, mas ela restringe desnecessariamente o acesso aos recursos compartilhados. Além disso, às vezes é impossível prever, no início de uma transação, quais objetos serão usados. Geralmente isso acontece em aplicativos interativos, pois o usuário teria de dizer com antecedência quais objetos exatamente estaria planejando usar, o que é inconcebível em aplicativos que usam navegação, os quais permitem aos usuários localizar objetos que eles não conhecem antecipadamente. O impasse também pode ser evitado pela solicitação de travas sobre objetos em uma ordem predefinida, mas isso pode resultar em bloqueio prematuro e em uma redução na concorrência.

Travas de *upgrade* ◊ O *Concurrency Control Service* do CORBA apresenta um terceiro tipo de trava, chamada de *upgrade*, cuja utilização se destina a evitar impasses. Um impasse é freqüentemente causado por duas transações conflitantes, primeiro obtendo travas de leitura e depois tentando promovê-las a travas de escrita. Uma transação com uma trava de *upgrade* sobre um item de dados pode ler esse item, mas a trava entra em conflito com quaisquer travas de *upgrade* configuradas por outras transações sobre o mesmo item de dados. Esse tipo de trava não pode ser configurado implicitamente pelo uso de uma operação *de leitura*, mas deve ser solicitado pelo cliente.

Detecção de impasses ◊ Os impasses podem ser detectados pela descoberta de ciclos no grafo de "espera por". Tendo detectado um impasse, uma transação deve ser selecionada para cancelamento para quebrar o ciclo.

O software responsável pela detecção de impasses pode fazer parte do gerenciador de travas. Ele deve conter uma representação do grafo de "espera por", para que possa verificar a existência de ciclos de tempos em tempos. Setas são adicionadas e removidas do grafo pelas operações *setLock* e *unLock* do gerenciador de travas. No ponto ilustrado pela Figura 13.22 à esquerda, terá as seguintes informações:

Transação	*Espera pela transação*
T	U, V
V	W
W	T, U, V

Uma seta $T \rightarrow U$ é adicionada quando o gerenciador de travas impede um pedido da transação T por uma trava sobre um objeto que já está bloqueado em nome da transação U. Note que, quando

uma trava é compartilhada, várias setas podem ser adicionadas. Uma seta $T \rightarrow U$ é excluída quando U libera uma trava pela qual T estava esperando e permite que T prossiga. Veja no Exercício 13.14 uma discussão mais detalhada sobre a implementação de detecção de impasses. Se uma transação compartilha uma trava, a trava não é liberada, mas as setas que levam a uma transação em particular são removidas.

A presença de ciclos pode ser verificada sempre que uma seta é adicionada (ou menos freqüentemente, para evitar sobrecarga desnecessária). Quando um impasse é detectado, uma das transações no ciclo deve ser escolhida e, então, cancelada. O nó e as setas correspondentes que a envolvem devem ser removidos do grafo de "espera por". Isso acontecerá quando a transação cancelada tiver suas travas removidas.

A escolha da transação a ser cancelada não é simples. Alguns fatores que podem ser levados em conta são a idade da transação e o número de ciclos em que ela está envolvida.

Tempos limites (*timeouts*) ◊ A limitação do tempo de bloqueio representa um método comumente usado para a solução de impasses. Cada trava recebe um período de tempo limitado, durante o qual ele é invulnerável. Após esse tempo, a trava se torna vulnerável. Desde que nenhuma outra transação esteja competindo pelo objeto bloqueado, um objeto com uma trava vulnerável permanece bloqueado. Entretanto, se qualquer outra transação estiver esperando para acessar o objeto protegido por uma trava vulnerável, a trava será quebrada (isto é, o objeto será desbloqueado) e a transação em espera será retomada. A transação cujo bloqueio foi quebrado normalmente é cancelada.

Existem muitos problemas no uso de tempos limites como uma solução para impasses: o pior deles é que, às vezes, as transações são canceladas porque suas travas se tornam vulneráveis quando outras transações estão esperando por elas, mas, na verdade, não há nenhum impasse. Em um sistema sobrecarregado, o número de transações com tempo limite esgotado aumentará, e as transações que demoram um longo tempo podem ser penalizadas. Além disso, é difícil decidir-se sobre a duração apropriada para um tempo limite. Em contraste, se for usada detecção de impasses, as transações serão canceladas porque ocorreram impasses e pode ser feita uma escolha com relação à qual transação vai ser cancelada.

Usando tempos limites para travas, podemos resolver o impasse da Figura 13.19, como mostrado na Figura 13.23, na qual a trava de escrita de T sobre A se torna vulnerável após seu período de tempo limite ter decorrido. A transação U está esperando para adquirir uma trava de escrita sobre A. Portanto, T é cancelada e libera sua trava sobre A, permitindo que U retome e complete a transação.

Transação T		*Transação U*	
Operações	Travas	Operações	Travas
a.deposit(100);	trava de escrita *A*		
		b.deposit(200)	trava de escrita *B*
b.withdraw(100)			
•••	espera pela trava de *U*	*a.withdraw(200);*	espera pela trava de T
	sobre *B*	•••	sobre *A*
	(decorre o tempo limite)	•••	
a trava de *T* sobre *A* se torna vulnerável, desbloqueia *A*, cancela T			
		a.withdraw(200);	travas de escrita *A*
			desbloqueia *A*, *B*

Figura 13.23 Solução do impasse da Figura 13.19.

Quando as transações acessam objetos localizados em vários servidores diferentes, surge a possibilidade de impasses distribuídos. Em um impasse distribuído, o grafo de "espera por" pode envolver objetos em vários locais. Voltaremos a esse assunto na Seção 14.5.

13.4.2 Aumentando a concorrência nos esquemas de bloqueio

Mesmo quando as regras de bloqueio são baseadas no conflito entre operações *de leitura* e *escrita* e a granularidade com que elas são aplicadas é a menor possível, ainda há uma chance de aumentar a concorrência. Vamos discutir duas estratégias que têm sido usadas. Na primeira (bloqueio de duas versões), a configuração de travas exclusivas é retardada até que uma transação seja efetivada. Na segunda estratégia (travas hierárquicas), são usadas travas com granularidade mista.

Bloqueio de duas versões ◊ Trata-se de um esquema otimista que permite a uma transação gravar versões de tentativa dos objetos, enquanto outras transações lêem a versão efetivada dos mesmos objetos. As operações *de leitura* só ficam na espera se outra transação estiver correntemente efetivando o mesmo objeto. Esse esquema possibilita mais concorrência do que as travas de leitura e escrita, mas as transações de escrita correm o risco de esperar ou mesmo de rejeição, quando tentam ser efetivadas. As transações não podem efetivar suas operações *de escrita* imediatamente, caso outras transações não concluídas tenham lido os mesmos objetos. Portanto, as transações que pedem para ser efetivadas em tal situação são obrigadas a esperar até que as transações de leitura tenham terminado. Um impasse pode ocorrer quando as transações estão esperando para serem efetivadas. Portanto, para resolver os impasses, talvez transações precisem ser canceladas quando estão esperando para serem efetivadas.

Essa variação do bloqueio de duas fases restrito usa três tipos de travas: uma trava de leitura, uma trava de escrita e uma trava de efetivação. Antes que a operação *de leitura* de uma transação seja efetuada, deve ser posta uma trava de leitura sobre o objeto – a tentativa de pôr uma trava de leitura é bem-sucedida, a não ser que o objeto tenha uma trava de efetivação, no caso em que a transação esperará. Antes que a operação *de escrita* de uma transação seja efetuada, uma trava de escrita deve ser posta sobre o objeto – a tentativa de pôr uma trava de escrita é bem-sucedida, a não ser que o objeto tenha uma trava de escrita ou uma trava de efetivação, caso em que a transação esperará.

Quando o coordenador de transação recebe um pedido para efetivar uma transação, ele tenta converter todas as travas de escrita dessa transação em travas de efetivação. Se qualquer um dos objetos tiver travas de leitura pendentes, a transação deverá esperar até que as transações que usam essas travas tenham terminado e as travas sejam liberadas. A compatibilidade das travas de leitura, escrita e efetivação aparecem na Figura 13.24.

Existem duas diferenças principais no desempenho entre o esquema de trava de duas versões e um esquema de trava de leitura e escrita normal. Por um lado, as operações *de leitura* no esquema de travas de duas versões são retardadas apenas enquanto as transações estão sendo efetivadas, em vez de o serem durante a execução inteira das transações – na maioria dos casos, o protocolo de efetivação leva apenas uma pequena fração do tempo exigido para realizar uma transação inteira. Por outro lado, as operações *de leitura* de uma transação podem causar um atraso na efetivação de outras transações.

Travas hierárquicas ◊ Em algumas aplicações, a granularidade conveniente para uma operação não é apropriada para outra. Em nosso exemplo de transações bancárias, a maioria das operações exige o

Para um objeto		*Trava a ser configurada*		
		leitura	*escrita*	*efetivação*
Trava já configurada	*nenhum*	OK	OK	OK
	leitura	OK	OK	espera
	escrita	OK	espera	–
	efetivação	espera	espera	–

Figura 13.24 Compatibilidade entre travas (travas de *leitura*, *escrita* e *efetivação*).

bloqueio com a granularidade de uma conta. A operação *branchTotal* é diferente – ela lê os valores dos saldos de todas as contas e exige uma trava de leitura sobre todas elas. Para reduzir a sobrecarga de bloqueio, seria útil permitir a coexistência de travas de granularidade mista.

Gray [1978] propôs o uso de uma hierarquia de travas com diferentes granularidades. Em cada nível, a configuração de uma trava ascendente tem o mesmo efeito que configurar todas as travas descendentes equivalentes. Isso traz uma economia no número de travas a serem configuradas. Em nosso exemplo de transações bancárias, a agência é a transação ascendente e as contas são as descendentes (veja a Figura 13.25).

As travas de granularidade mista poderiam ser úteis em um sistema de agenda, no qual os dados poderiam ser estruturados, com a agenda de uma semana sendo composta de uma página para cada dia e cada dia subdividido ainda em uma repartição para cada hora, como mostrado na Figura 13.26. A operação para ver uma semana faria uma trava de leitura ser posta no topo dessa hierarquia, enquanto a operação para inserir um compromisso faria uma trava de escrita ser posta em uma repartição de tempo. O efeito de uma trava de leitura sobre uma semana impediria operações de escrita em qualquer uma das subestruturas; por exemplo, as repartições de tempo de cada dia nessa semana.

No esquema de Gray, cada nó na hierarquia pode ser bloqueado – dando ao proprietário da trava acesso explícito ao nó e implícito aos seus descendentes. Em nosso exemplo, na Figura 13.25, uma trava de leitura/escrita sobre a agência bloqueia implicitamente todas as contas para leitura e escrita. Antes que um nó descendente receba uma trava de leitura/escrita, a intenção de usar essa trava é configurada no nó ascendente e em seus ancestrais (se houver). A trava de intenção é compatível com outras travas de intenção, mas entra em conflito com as travas de leitura e escrita, de acordo com as regras normais. A Figura 13.27 fornece a tabela de compatibilidade para travas hierárquicas. Gray também propôs um terceiro tipo de trava de intenção – o qual combina a propriedade de uma trava de leitura com uma intenção de escrita.

Em nosso exemplo de transações bancárias, a operação *branchTotal* solicita uma trava de leitura sobre a agência, a qual configura travas de leitura implicitamente sobre todas as contas. Uma operação *deposit* precisa configurar uma trava de escrita sobre um saldo, mas primeiro ela tenta configurar uma intenção de trava de escrita sobre a agência. Essas regras impedem que essas operações sejam efetuadas concorrentemente.

As travas hierárquicas têm a vantagem de serem em menor número quando é exigido um bloqueio de granularidade mista. As tabelas de compatibilidade e as regras para promover travas são mais complexas.

Figura 13.25 Hierarquia de travas do exemplo de transações bancárias.

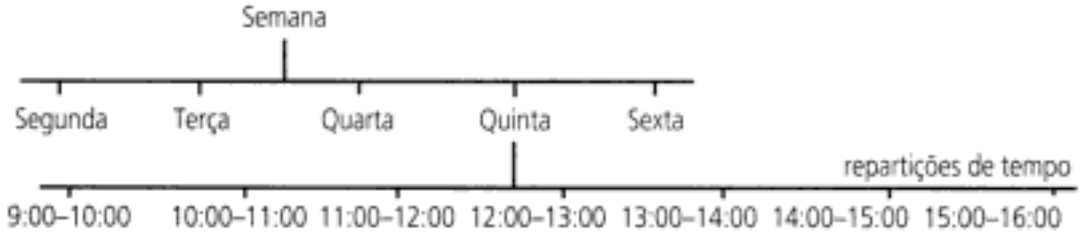


Figura 13.26 Hierarquia de travas para uma agenda.

Para um objeto		*Trava a ser configurada*			
		leitura	*escrita*	*I-leitura*	*I-escrita*
Trava já configurada	*nenhum*	OK	OK	OK	OK
	leitura	OK	espera	OK	espera
	escrita	espera	espera	espera	espera
	I-leitura	OK	espera	OK	OK
	I-escrita	espera	espera	OK	OK

Figura 13.27 Tabela de compatibilidade para travas hierárquicas.

A granularidade mista das travas poderia deixar que cada transação bloqueasse uma parte, cujo tamanho seria escolhido de acordo com suas necessidades. Uma transação longa, que acessasse muitos objetos, poderia bloquear o conjunto inteiro, enquanto uma transação curta poderia fazer o bloqueio com uma granularidade menor.

O *Concurrency Control Service* do CORBA suporta travas de granularidade variável com tipos de travas de intenção de ler e intenção de escrever. Eles podem ser usados conforme descrito anteriormente, para aproveitar a oportunidade de aplicar travas com diferentes granularidades em dados estruturados hierarquicamente.

13.5 Controle de concorrência otimista

Kung e Robinson [1981] identificaram várias desvantagens inerentes ao bloqueio e propuseram uma estratégia otimista alternativa à organização em série das transações, que evita esses inconvenientes. Podemos resumir os inconvenientes do bloqueio, como segue:

- A manutenção da trava representa uma sobrecarga que não está presente em sistemas que não suportam acesso concorrente a dados compartilhados. Em geral, mesmo as transações somente de leitura (consultas), que possivelmente não afetam a integridade dos dados, devem usar travas para garantir que os dados que estão sendo lidos não sejam modificados simultaneamente por outras transações. Mas o bloqueio pode ser necessário apenas no pior caso.

  Por exemplo, considere dois processos clientes que estão incrementando os valores de n objetos concorrentemente. Se os programas clientes começam simultaneamente e são executados durante o mesmo período de tempo, acessando os objetos em duas seqüências não relacionadas e usando uma transação separada para acessar e incrementar cada item, as chances de que os dois programas tentem acessar o mesmo objeto ao mesmo tempo são de apenas uma em n, em média; portanto, o bloqueio é realmente necessário apenas uma vez em cada n transações.

- O uso de travas pode resultar em um impasse. A prevenção de impasses reduz seriamente a concorrência e, portanto, as situações de impasse devem ser resolvidas com o uso de tempos limites ou com detecção de impasses. Nenhuma delas é totalmente satisfatória para uso em programas interativos.
- Para evitar os cancelamentos em cascata, as travas não podem ser liberadas até o final da transação. Isso pode reduzir significativamente o potencial de concorrência.

A estratégia alternativa proposta por Kung e Robinson é otimista porque é baseada na observação de que, na maioria das aplicações, a probabilidade das transações de dois clientes acessarem o mesmo objeto é baixa. As transações podem prosseguir como se não houvesse nenhuma possibilidade de

conflito com outras transações, até que o cliente conclua sua tarefa e emita um pedido de *closeTransaction*. Quando surge um conflito, alguma transação geralmente é cancelada e precisará ser reiniciada pelo cliente. Cada transação tem as seguintes fases:

Fase de trabalho: durante a fase de trabalho, cada transação tem uma versão tentativa de cada um dos objetos que atualiza. Trata-se de uma cópia da versão do objeto efetivada mais recentemente. O uso de versões de tentativa permite que a transação seja cancelada (sem nenhum efeito sobre os objetos) durante a fase de trabalho ou, se ela falhar na validação, devido a outras transações conflitantes. As operações *de leitura* são executadas imediatamente – se já existir uma versão de tentativa para essa transação, uma operação *de leitura* a acessará; caso contrário, ela acessará o valor do objeto efetivado mais recentemente. As operações *de escrita* registram os novos valores dos objetos como valores de tentativa (que são invisíveis para as outras transações). Quando existem várias transações concomitantes, diversos valores de tentativa diferentes do mesmo objeto podem coexistir. Além disso, são mantidos dois registros dos objetos acessados dentro de uma transação: um *conjunto de leitura*, contendo os objetos lidos pela transação, e um *conjunto de escrita*, contendo os objetos modificados por ela. Note que, como todas as operações *de leitura* são realizadas em versões efetivadas dos objetos (ou em cópias deles), não há a ocorrência de leituras sujas.

Fase de validação: quando o pedido de *closeTransaction* é recebido, a transação é validada para estabelecer se suas operações sobre os objetos entram em conflito ou não com as operações de outras transações sobre os mesmos objetos. Se a validação for bem-sucedida, então a transação poderá ser efetivada. Se a validação falhar, então alguma forma de solução de conflito deve ser usada e a transação corrente ou, em alguns casos, aquelas com que ela está em conflito, precisarão ser canceladas.

Fase de atualização: se uma transação é validada, todas as alterações registradas em suas versões de tentativa tornam-se permanentes. As transações somente de leitura podem ser efetivadas imediatamente, após passarem pela validação. As transações de escrita estarão prontas para serem efetivadas quando as versões de tentativa dos objetos tiverem sido registradas no meio de armazenamento permanente.

Validação de transações ◊ A validação usa as regras de conflito de leitura e escrita para garantir que a programação de uma transação em particular seja equivalente em série com relação a todas as outras transações *sobrepostas* – isto é, todas as transações que ainda não tinham sido efetivadas no momento que essa transação começou. Para ajudar na realização da validação, cada transação recebe um número ao entrar na fase de validação (isto é, quando o cliente emite um pedido de *closeTransaction*). Se a transação for validada e terminar com sucesso, ela manterá esse número; se falhar nas verificações da validação e for cancelada, ou se a transação for somente de leitura, o número será liberado para uma nova atribuição posterior. Os números de transação são valores inteiros atribuídos em seqüência ascendente; portanto, o número de uma transação define sua posição no tempo – uma transação sempre termina sua fase de trabalho após todas as transações que possuem números mais baixos. Isto é, uma transação com o número T_i sempre precede uma transação com o número T_j, se $i < j$. (Se o número da transação fosse atribuído no início da fase de trabalho, uma transação que chegasse ao final dessa fase antes de outra com um número menor teria de esperar até que a anterior tivesse terminado, antes de poder ser validada.)

O teste de validação na transação T_v é baseado no conflito entre operações em pares de transação T_i e T_v. Para que uma transação T_v seja disposta em série com relação a uma transação sobreposta T_i, suas operações devem obedecer às seguintes regras:

T_v	T_i	Regra	
escrita	*leitura*	1.	T_i não deve ler objetos escritos por T_v.
leitura	*escrita*	2.	T_v não deve ler objetos escritos por T_i.
escrita	*escrita*	3.	T_i não deve escrever em objetos modificados por T_v e T_v não deve escrever em objetos modificados por T_i.

Como as fases de validação e atualização de uma transação geralmente têm curta duração, comparadas com a fase de trabalho, uma simplificação pode ser obtida criando-se a regra de que apenas uma transação pode estar na fase de validação e atualização em dado momento. Quando duas transações não podem se sobrepor na fase de atualização, a regra 3 é satisfeita. Note que essa restrição sobre as operações *de escrita*, junto com o fato de que não podem ocorrer leituras sujas, produz execuções restritas. Para impedir a sobreposição, as fases de validação e atualização inteiras podem ser implementadas como uma seção crítica para que apenas um cliente por vez possa executá-la. Para aumentar a concorrência, parte da validação e da atualização pode ser implementada fora da seção crítica, mas é fundamental que a atribuição de números de transação seja realizada em seqüência. Notamos que, a qualquer instante, o número da transação corrente é como um pseudo-relógio que começa a funcionar quando uma transação termina com sucesso.

A validação de uma transação deve garantir que as regras 1 e 2 sejam obedecidas, testando as sobreposições entre os objetos de pares de transações T_v e T_i. Existem duas formas de validação – para trás e para frente [Härder 1984]. A validação para trás verifica a transação sendo submetida à validação com outras transações sobrepostas precedentes – aquelas que entraram na fase de validação antes dela. A validação para frente verifica a transação sendo submetida à validação com outras transações posteriores, que ainda estão ativas.

Validação para trás (*backward*) ◊ Como todas as operações *de leitura* das transações sobrepostas anteriores foram executadas antes que a validação de T_v começasse, elas não podem ser afetadas pelas escritas da transação corrente (e a regra 1 é satisfeita). A validação da transação T_v verifica se seu conjunto de leitura (os objetos afetados pelas operações *de leitura* de T_v) se sobrepõe a qualquer um dos conjuntos de escrita das transações sobrepostas anteriores T_i (regra 2). Se houver qualquer sobreposição, a validação falhará.

Seja *startTn* o maior número de transação atribuído (para alguma outra transação efetivada) no momento em que a transação T_v começou sua fase de trabalho e *finishTn* o maior número de transação atribuído no momento em que T_v entrou na fase de validação. O programa a seguir descreve o algoritmo para a validação de T_v:

```
boolean valid = true;
for (int Ti = startTn+1; Ti <= finishTn; Ti++){
        if (conjunto de leitura de Tv possui interseção com o conjunto de escrita de Ti) valid = false;
}
```

A Figura 13.28 mostra transações sobrepostas que poderiam ser consideradas na validação de uma transação T_v. O tempo aumenta da esquerda para a direita. As transações efetivadas anteriores são T_1, T_2 e T_3. T_1 foi efetivada antes que T_v começasse. T_2 e T_3 foram efetivadas antes que T_v terminasse sua fase de trabalho. $StartTn + 1 = T_2$ e $finishTn = T_3$. Na validação para trás, o conjunto de leitura de T_v deve ser comparado com os conjuntos de escrita de T_2 e T_3.

Na validação para trás, o conjunto de leitura da transação que está sendo validada é comparado com os conjuntos de escrita das outras transações que já foram efetivadas. Portanto, a única maneira de resolver qualquer conflito é cancelar a transação que está passando pela validação.

Na validação para trás, as transações que não têm operações *de leitura* (operações somente *de escrita*) não precisam ser verificadas.

O controle de concorrência otimista com validação para trás exige que os conjuntos de escrita de versões efetivadas antigas dos objetos, correspondentes às transações recentemente efetivadas, sejam mantidos até que não existam transações sobrepostas não validadas com as quais poderiam entrar em conflito. Quando uma transação é validada com sucesso, seu número de transação, *startTn*, e o conjunto de escritas são registrados em uma lista de transações precedentes, mantida pelo serviço de transação. Note que essa lista é ordenada pelo número de transação. Em um ambiente com transações longas, a retenção de conjuntos de escrita de objetos antigos pode ser um problema. Por exemplo, na Figura 13.28, os conjuntos de escrita de T_1, T_2, T_3 e T_v devem ser mantidos até que a transação ativa $active_3$ termine. Note que, embora as transações ativas tenham identificadores de transação, elas ainda não possuem números de transação.

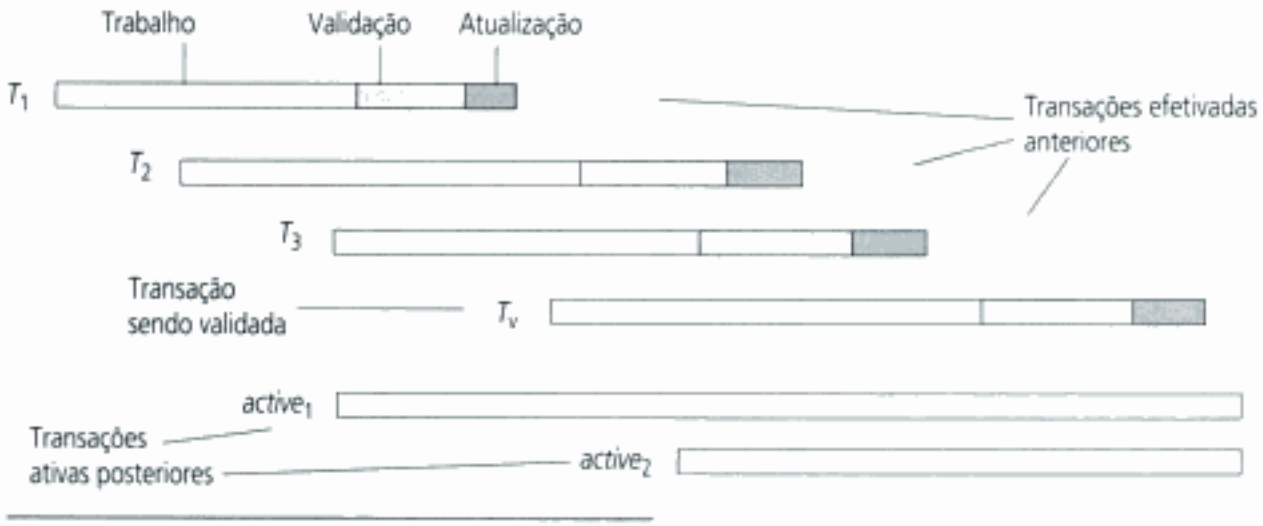


Figura 13.28 Validação de transações.

Validação para frente ◊ Na validação para frente da transação T_v, o conjunto de escrita de T_v é comparado com os conjuntos de leitura de todas as transações ativas sobrepostas – aquelas que ainda estão em sua fase de trabalho (regra 1). A regra 2 é satisfeita automaticamente, pois as transações ativas só escrevem depois que T_v tiver terminado. Se as transações ativas tiverem identificadores de transação (consecutivos) $active_1$ a $active_N$, então o programa a seguir descreve o algoritmo para a validação para frente de T_v:

```
boolean valid = true;
for (int T_id = active_1; T_id <= active_N; T_id++){
    if (conjunto de gravação de T_v possui interseção com o conjunto de leitura de T_id) valid = false;
}
```

Na Figura 13.28, o conjunto de escrita da transação T_v deve ser comparado com os conjuntos de leitura das transações com identificadores $active_1$ e $active_2$. (A validação para frente deve se preparar para o fato de que os conjuntos de leitura das transações ativas podem mudar durante a validação e a escrita.) Como os conjuntos de leitura da transação que está sendo validada não são incluídos na verificação, as transações somente de leitura sempre passam na verificação da validação. Como as transações que estão sendo comparadas com a transação que está sendo validada ainda estão ativas, temos a escolha de cancelar a transação que está sendo validada, ou adotar alguma maneira alternativa de resolver o conflito. Härder [1984] sugere várias estratégias alternativas:

- Adiar a validação até um momento posterior, quando as transações conflitantes tiverem terminado. Entretanto, não há garantia de que a transação que está sendo validada irá passar melhor no futuro. Sempre existe a chance de que mais transações ativas conflitantes possam começar, antes que a validação seja obtida.
- Cancelar todas as transações ativas conflitantes e efetivar a transação que está sendo validada.
- Cancelar a transação que está sendo validada. Esta é a estratégia mais simples, mas tem a desvantagem de que as futuras transações conflitantes podem vir a ser canceladas, no caso em que a transação sob validação tenha sido cancelada desnecessariamente.

Comparação das validações para frente e para trás ◊ Já vimos que a validação para frente permite flexibilidade na solução de conflitos, enquanto a validação para trás permite apenas uma escolha – cancelar a transação que está sendo validada. Em geral, os conjuntos de leitura das transações são muito maiores do que os conjuntos de escrita. Portanto, a validação para trás compara um conjunto de leitura possivelmente grande com os conjuntos de gravação antigos, enquanto a validação para frente verifica um pequeno conjunto de escrita em relação aos conjuntos de leitura das transações ativas. Ve-

Hidden page

presentado por seu membro máximo. Quando a operação *de escrita* de uma transação sobre um objeto é aceita, o servidor cria uma nova versão de tentativa do objeto, com indicação de tempo de escrita configurada na indicação de tempo da transação. A operação *de leitura* de uma transação é direcionada para a versão com a máxima indicação de tempo de escrita menor do que a indicação de tempo da transação. Quando a operação *de leitura* de uma transação sobre um objeto é aceita, a indicação de tempo da transação é adicionada em seu conjunto de indicação de tempo de leitura. Quando uma transação é efetivada, os valores das versões de tentativa tornam-se os valores dos objetos e as indicações de tempo das versões de tentativa tornam-se as indicações de tempo dos objetos correspondentes.

Na ordenação por indicação de tempo, cada pedido de uma transação para uma operação *de leitura* ou *escrita* sobre um objeto é verificado para ver se ele obedece às regras de conflito de operação. Um pedido da transação corrente T_c pode entrar em conflito com operações anteriores executadas por outras transações, T_i, cujas indicações de tempo mostram que elas devem ser posteriores a T_c. Essas regras aparecem na Figura 13.29, na qual $T_i > T_c$ significa que T_i é posterior a T_c e $T_i < T_c$ significa que T_i é anterior a T_c.

Regra de escrita na ordenação de indicação de tempo: combinando as regras 1 e 2, temos a seguinte regra para decidir se devemos aceitar uma operação *de escrita* solicitada pela transação T_c sobre o objeto D:

```
se (Tc ≥ indicação de tempo de leitura máxima em D &&
    Tc > indicação de tempo de escrita na versão efetivada de D)
        executa a operação de escrita na versão de tentativa de D, com indicação de tempo de escrita Tc
senão /* a escrita se deu tarde demais */
    Cancela a transação Tc
```

Se já existe uma versão de tentativa com indicação de tempo de escrita T_c, a operação *de escrita* é endereçada a ela; caso contrário, uma nova versão de tentativa é criada e recebe uma indicação de tempo de gravação T_c. Note que qualquer *escrita* que chegue tarde demais será cancelada – ela ocorreu tarde demais no sentido de que uma transação com uma indicação de tempo posterior já leu ou gravou o objeto.

A Figura 13.30 ilustra a ação de uma operação de escrita pela transação T_3 nos casos onde $T_3 \geq$ indicação de tempo de leitura máxima no objeto (as indicações de tempo de leitura não são mostradas). Nos casos (a) a (c) $T_3 >$ indicação de tempo de escrita na versão efetivada do objeto e uma versão de tentativa com indicação de tempo de gravação T_3 é inserida no lugar apropriado da lista de versões de tentativa ordenada pelas suas indicações de tempo de transação. No caso (d), $T_3 <$ indicação de tempo de escrita na versão efetivada do objeto e a transação é cancelada.

Regra de leitura na ordenação de indicação de tempo: usando a regra 3, temos a seguinte regra para decidir se devemos aceitar imediatamente, esperar ou rejeitar uma operação de leitura solicitada pela transação T_c sobre o objeto D:

```
se (Tc > indicação de tempo de escrita na versão efetivada de D) {
```

Regra	T_c	T_i	
1.	*escrita*	*leitura*	T_c não deve *escrever* um objeto que tenha sido *lido* por qualquer T_i, onde $T_i > T_c$; isso exige que $T_c \geq$ a indicação de tempo de leitura máxima do objeto.
2.	*escrita*	*escrita*	T_c não deve *escrever* um objeto que tenha sido *modificado* por qualquer T_i, onde $T_i > T_c$; isso exige que $T_c >$ indicação de tempo de escrita do objeto efetivado.
3.	*leitura*	*escrita*	T_c não deve *ler* um objeto que tenha sido *modificado* por qualquer T_i, onde $T_i > T_c$; isso exige que $T_c >$ indicação de tempo de escrita do objeto efetivado.

Figura 13.29 Conflito de operação para ordenação da indicação de tempo.

Hidden page

Hidden page

		Indicações de tempo e versões de objetos					
T	*U*	*A*		*B*		*C*	
		ITL	*ITE*	*ITL*	*ITE*	*ITL*	*ITE*
		{}	***S***	{}	***S***	{}	***S***
openTransaction							
bal = b.getBalance()				{*T*}			
	openTransaction						
*b.setBalance(bal*1.1)*					***S***, *T*		
	bal = b.getBalance()						
	wait for T						
a.withdraw(bal/10)	•••		*S*, *T*				
commit	•••		***T***		***T***		
	bal = b.getBalance()			{*U*}			
	*b.setBalance(bal*1.1)*				***T***, *U*		
	c.withdraw(bal/10)						*S*, *U*

Figura 13.32 Indicações de tempo nas transações *T* e *U*.

O método de indicação de tempo que acabamos de descrever evita impasses, mas é bastante propenso a reinícios. Uma modificação conhecida como regra de *ignorar gravação obsoleta* é uma melhoria. Trata-se de uma modificação na regra de escrita na ordenação de indicação de tempo:

- Se uma escrita for feita tarde demais, ela pode ser ignorada, em vez de cancelar a transação, pois se ela tivesse chegado a tempo, seus efeitos teriam sido sobrescritos de qualquer forma. Entretanto, se outra transação tiver lido o objeto, a transação com a escrita tardia falhará, devido à indicação de tempo de leitura presente no item.

Ordenação de indicação de tempo de versão múltipla ◊ Nesta seção, mostramos como a concorrência fornecida pela ordenação de indicação de tempo básica é melhorada, permitindo-se que cada transação escreva suas próprias versões de tentativa dos objetos. Na ordenação por indicação de tempo de versão múltipla, que foi introduzida por Reed [1983], é mantida uma lista de versões efetivadas antigas, assim como das versões de tentativa, para cada objeto. Essa lista representa o histórico dos valores do objeto. A vantagem de usar múltiplas versões é que as operações *de leitura* que chegarem tarde demais não precisam ser rejeitadas.

Cada versão tem uma indicação de tempo de leitura registrando a maior indicação de tempo de toda transação que a tiver lido, além de uma indicação de tempo de escrita. Como antes, quando uma operação *de escrita* é aceita, ela é direcionada para uma versão de tentativa com a indicação de tempo de escrita da transação. Quando uma operação *de leitura* é executada, ela é direcionada para a versão com a maior indicação de tempo de escrita menor do que a indicação de tempo da transação. Se a indicação de tempo da transação for maior do que a indicação de tempo de leitura da versão que está sendo usada, a indicação de tempo de leitura da versão será configurada com a indicação de tempo da transação.

Quando uma leitura chegar tarde, ela poderá ter permissão para ler uma versão efetivada antiga, para que não haja necessidade de cancelar operações *de leitura* tardias. Na ordenação por indicação de tempo de versão múltipla, as operações *de leitura* são sempre permitidas, embora talvez tenham de *esperar* que transações anteriores terminem (efetivadas ou canceladas), o que garante que as execuções sejam recuperáveis. Veja no Exercício 13.22 uma discussão sobre a possibilidade de cancelamentos em cascata. Isso tem a ver com a regra 3 das regras de conflito para ordenação por indicação de tempo.

Hidden page

Quando uma transação é cancelada, todas as versões que criou são excluídas. Quando uma transação é efetivada, todas as versões que criou são mantidas, mas para controlar o uso do espaço de armazenamento, as versões antigas devem ser excluídas de tempos em tempos. Embora tenha a sobrecarga do espaço de armazenamento, a ordenação de indicação de tempo de versão múltipla permite uma concorrência considerável, não sofre do problema dos impasses e sempre permite as operações *de leitura*. Para obter mais informações sobre a ordenação de indicação de tempo de versão múltipla, consulte Bernstein *et al.* [1987].

13.7 Comparação dos métodos de controle de concorrência

Descrevemos três métodos separados para controlar o acesso concomitante a dados compartilhados: bloqueio de duas fases restrito, métodos otimistas e ordenação de indicação de tempo. Todos trazem consigo algumas sobrecargas no tempo e no espaço exigido, e todos eles limitam até certo ponto o potencial para operação concorrente.

O método da ordenação de indicação de tempo é semelhante ao bloqueio de duas fases, pois ambos usam estratégias pessimistas, nas quais os conflitos entre as transações são detectados quando cada objeto é acessado. Por um lado, a ordenação de indicação de tempo decide estaticamente a ordem da organização em série – quando uma transação começa. Por outro lado, o bloqueio de duas fases decide dinamicamente a ordem da organização em série – de acordo com a ordem em que os objetos são acessados. A ordenação de indicação de tempo e, em particular, a ordenação de indicação de tempo de versão múltipla são melhores do que o bloqueio de duas fases restrito para transações somente de leitura. O bloqueio de duas fases é melhor quando as operações nas transações são predominantemente atualizações.

Um trabalho utiliza a observação de que a ordenação de indicação de tempo é vantajosa para transações com operações predominantemente *de leitura* e que o bloqueio é vantajoso para transações com mais *escritas* do que *leituras*, como argumento para permitir a existência de esquemas mistos, nos quais algumas transações usam ordenação de indicação de tempo e outras usam bloqueio para controle de concorrência. Os leitores que estiverem interessados no uso de métodos mistos devem ler Bernstein *et al.* [1987].

Os métodos pessimistas diferem na estratégia usada quando é detectado um acesso conflitante a um objeto. A ordenação de indicação de tempo cancela a transação imediatamente, enquanto o bloqueio faz a transação esperar – mas com a possível penalidade posterior de cancelar para evitar impasse.

Quando é usado o controle de concorrência otimista, todas as transações podem prosseguir, mas algumas são canceladas quando tentam ser efetivadas ou, nas transações com validação para frente, são canceladas anteriormente. Isso resulta em uma operação relativamente eficiente quando existem poucos conflitos, mas um volume de trabalho substancial talvez tenha de ser repetido quando uma transação for cancelada.

O bloqueio está em uso há muitos anos em sistemas de banco de dados, mas a ordenação de indicação de tempo foi usada no sistema de banco de dados SDD-1. Os dois métodos são usados em servidores de arquivo. Entretanto, o método de controle de concorrência para o acesso a dados predominante em sistemas distribuídos é o bloqueio; por exemplo, conforme mencionado anteriormente, o *Concurrency Control Service* do CORBA é baseado inteiramente no uso de travas. Em particular, ele fornece travas hierárquicas, o que possibilita a existência de um bloqueio de granularidade mista em dados estruturados hierarquicamente.

Vários sistemas distribuídos de pesquisa, por exemplo, Argus [Liskov 1988] e Arjuna [Shrivastava *et al.* 1991], têm explorado o uso de travas semânticas, ordenação de indicação de tempo e novas estratégias para transações longas.

O trabalho feito em duas áreas de aplicação mostra que os mecanismos de controle de concorrência anteriores nem sempre são adequados. Uma dessas áreas se preocupa com os aplicativos multiusuário, nos quais todos os usuários esperam ver modos de visualização comuns dos objetos que estão sendo atualizados por qualquer um deles. Tais aplicativos exigem que seus dados sejam atômi-

cos na presença de atualizações concorrentes e falhas de servidor; e as técnicas de transação parecem oferecer uma estratégia para seu projeto. Entretanto, esses aplicativos têm dois novos requisitos relacionados ao controle de concorrência: (1) os usuários exigem notificação imediata sobre as alterações feitas pelos outros usuários – o que é contrário à idéia de isolamento; e (2) os usuários precisam acessar objetos antes que os outros usuários tenham concluído suas transações, o que tem levado ao desenvolvimento de novos tipos de travas que disparam ações quando os objetos são acessados. O trabalho feito nessa área sugere muitos esquemas que suavizam o isolamento e fornecem notificação sobre as alterações. Para examinar esse trabalho, consulte Ellis *et al.* [1991]. A segunda área de aplicação diz respeito ao que às vezes são descritos como aplicativos de banco de dados avançados – como o CAD/CAM cooperativo e os sistemas de desenvolvimento de software. Em tais aplicativos, as transações duram por um longo tempo e os usuários trabalham em versões independentes dos objetos, que são retirados de um banco de dados comum e recolocados quando o trabalho está terminado. A integração das versões exige cooperação entre os usuários. Para examinar esse trabalho, consulte Barghouti e Kaiser [1991].

13.8 Resumo

As transações fornecem um meio pelo qual os clientes podem especificar seqüências de operações, que são atômicas na presença de outras transações concorrentes e de falhas do servidor. O primeiro aspecto da atomicidade é obtido pela execução de transações de modo que seus efeitos sejam equivalentes em série. Os efeitos das transações efetivadas são registrados no meio de armazenamento permanente para que o serviço de transação possa se recuperar de falhas de processo. Para proporcionar às transações a capacidade de serem canceladas, sem efeitos colaterais prejudiciais nas outras transações, as execuções devem ser restritas – isto é, as leituras e escritas de uma transação devem ser retardadas até que as outras transações que modificaram os mesmos objetos tenham sido efetivadas ou canceladas. Para possibilitar às transações a escolha de efetivar ou cancelar, suas operações são executadas em versões de tentativa que não podem ser acessadas pelas outras transações. As versões de tentativa dos objetos são copiadas nos objetos reais e no meio de armazenamento permanente, quando uma transação é efetivada.

As transações aninhadas são formadas por meio da estruturação de transações de outras subtransações. O aninhamento é particularmente útil em sistemas distribuídos, pois ele permite a execução concorrente de subtransações em servidores separados. O aninhamento também tem a vantagem de permitir a recuperação independente das partes de uma transação.

Os conflitos de operação formam a base para a derivação de protocolos de controle de concorrência. Os protocolos não devem apenas garantir a organização em série, mas também permitir a recuperação, usando execuções restritas para evitar os problemas associados ao cancelamento de transações, como os cancelamentos em cascata.

Três estratégias alternativas são possíveis na programação da execução de uma operação em uma transação. São elas: (1) executá-la imediatamente, (2) retardá-la ou (3) cancelá-la.

O bloqueio de duas fases restrito usa as duas primeiras estratégias, contando com o cancelamento apenas em caso de impasse. Ele garante a organização em série por meio da ordenação das transações de acordo com o momento em que elas acessam objetos comuns. Seu principal inconveniente é que podem ocorrer impasses.

A ordenação de indicação de tempo usa todas as três estratégias para garantir a organização em série por meio da ordenação dos acessos das transações aos objetos, de acordo com a hora que as transações começam. Esse método não pode sofrer de impasses e é vantajoso para transações somente de leitura. Entretanto, as transações devem ser canceladas quando chegam tarde demais. A ordenação de indicação de tempo de versão múltipla é particularmente eficiente.

O controle de concorrência otimista permite que as transações prossigam, sem qualquer forma de verificação, até que sejam concluídas. As transações são validadas antes de poderem ser efetivadas. A validação para trás exige a manutenção de vários conjuntos de escrita de transações efetivadas, enquanto a validação para frente precisa ser validada em relação às transações ativas e tem a vantagem

Hidden page

Hidden page

Hidden page

Hidden page

Transações Distribuídas 14

Este capítulo apresenta as transações distribuídas – aquelas que envolvem mais de um servidor. As transações distribuídas podem ser planas ou aninhadas.

Um protocolo de efetivação atômica é um procedimento cooperativo usado por um conjunto de servidores envolvidos em uma transação distribuída. Ele permite que os servidores cheguem a uma decisão conjunta quanto ao fato de uma transação poder ser efetivada ou cancelada. Este capítulo descreve o protocolo de efetivação de duas fases, que é o protocolo atômico de efetivação mais comumente usado.

A seção sobre controle de concorrência em transações distribuídas discute como o bloqueio, a ordenação por indicação de tempo e o controle de concorrência otimista podem ser estendidos para uso com transações distribuídas.

O uso de esquemas de bloqueio pode levar aos impasses distribuídos. Serão discutidos os algoritmos de detecção de impasse distribuído.

Os servidores que fornecem transações incluem um gerenciador de recuperação cuja função é garantir que os efeitos das transações sobre os objetos gerenciados por um servidor possam ser recuperados, quando ele é substituído após uma falha. O gerenciador de recuperação salva os objetos no meio de armazenamento permanente, junto com listas de intenções e informações sobre o *status* de cada transação.

14.1 Introdução

No Capítulo 13, discutimos as transações planas e aninhadas que acessavam objetos em um único servidor. No caso geral, uma transação, seja plana ou aninhada, acessará objetos localizados em vários computadores diferentes. Usamos o termo *transação distribuída* para nos referirmos a uma transação plana ou aninhada que acessa objetos gerenciados por vários servidores.

Quando uma transação distribuída chega ao fim, a propriedade da atomicidade das transações exige que todos os servidores envolvidos efetivem a transação, ou que todos eles a cancelem. Para se chegar a isso, um dos servidores assume o papel de coordenador, o que envolve garantir o mesmo resultado em todos os servidores. A maneira pela qual o coordenador chega a isso depende do protocolo escolhido. Um protocolo conhecido como protocolo de efetivação de duas fases é o mais comumente usado. Esse protocolo permite que os servidores se comuniquem para chegar a uma decisão conjunta quanto a efetivar ou cancelar uma transação.

O controle de concorrência em transações distribuídas é baseado nos métodos discutidos no Capítulo 13. Cada servidor aplica controle de concorrência local em seus próprios objetos, o que garante que as transações sejam organizadas em série de forma local. As transações distribuídas devem ser organizadas em série de forma global. O modo como isso é obtido varia de acordo com o fato de estar em uso bloqueio, ordenação por indicação de tempo ou controle de concorrência otimista. Em alguns casos, as transações podem ser organizadas em série nos servidores individuais, mas, ao mesmo tempo, pode ocorrer um ciclo de dependências entre os diferentes servidores e surgir um impasse distribuído.

A recuperação de transação se preocupa em garantir que todos os objetos envolvidos nas transações sejam recuperáveis. Além disso, ela garante que os valores dos objetos reflitam todas as alterações feitas pelas transações efetivadas e nenhuma das alterações feitas pelas transações canceladas.

14.2 Transações distribuídas planas e aninhadas

Uma transação cliente se torna distribuída se ativa operações em vários servidores diferentes. Existem duas maneiras distintas pelas quais as transações distribuídas podem ser estruturadas: como transações planas e como transações aninhadas.

Em uma transação plana, um cliente faz pedidos para mais de um servidor. Por exemplo, na Figura 14.1(a), a transação T é uma transação plana que invoca operações sobre objetos nos servidores X, Y e Z. Uma transação cliente plana conclui cada um de seus pedidos antes de passar para o próximo. Portanto, cada transação acessa objetos dos servidores em seqüência. Quando os servidores usam bloqueio, uma transação só pode estar esperando um objeto por vez.

Em uma transação aninhada, a transação de nível superior pode abrir subtransações, e cada subtransação pode abrir mais subtransações em qualquer profundidade de aninhamento. A Figura 14.1(b) mostra a transação T de um cliente, que abre duas subtransações T_1 e T_2, as quais acessam objetos nos servidores X e Y. As subtransações T_1 e T_2 abrem mais subtransações, T_{11}, T_{12}, T_{21} e T_{22}, as quais acessam objetos nos servidores M, N e P. No caso aninhado, as subtransações no mesmo nível podem ser executadas concomitantemente, de modo que T_1 e T_2 são concorrentes e, como invocam objetos em servidores diferentes, elas podem ser executadas em paralelo. As quatro subtransações T_{11}, T_{12}, T_{21} e T_{22} também são executadas concomitantemente.

Considere uma transação distribuída na qual um cliente transfere \$10 da conta A para C, e depois transfere \$20 de B para D. As contas A e B estão em servidores separados X e Y e as contas C e D estão no servidor Z. Se essa transação está estruturada como um conjunto de quatro transações aninhadas, como mostrado na Figura 14.2, os quatro pedidos (dois depósitos [*deposit*] e dois saques [*withdraw*]) podem ser executados em paralelo e o efeito global pode ser obtido com melhor desempenho do que em uma transação simples, na qual as quatro operações são ativadas em seqüência.

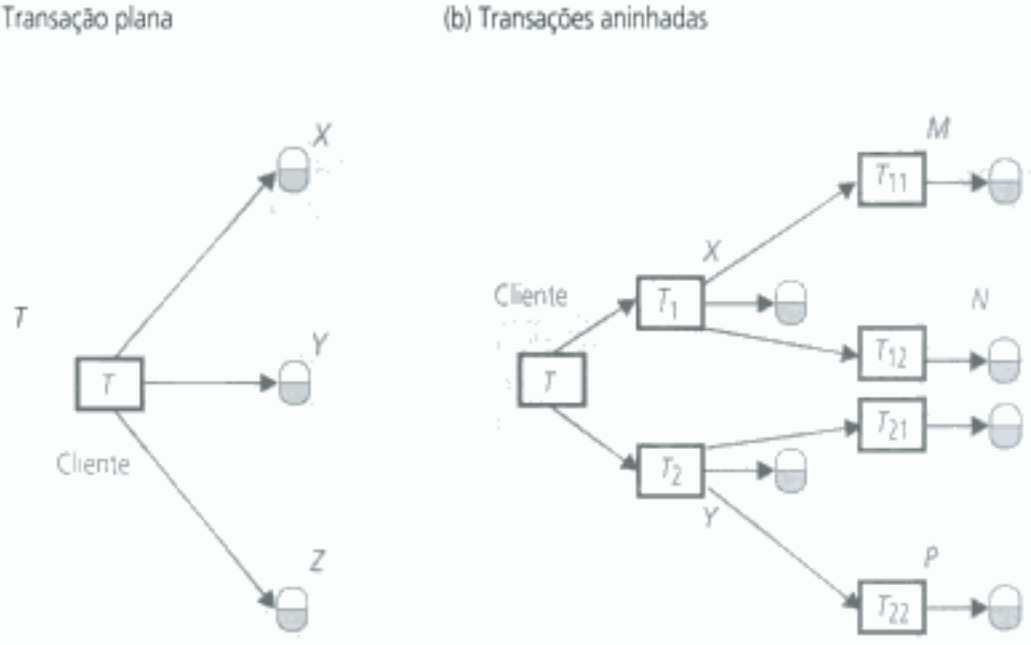


Figura 14.1 Transações distribuídas.

14.2.1 O coordenador de uma transação distribuída

Os servidores que executam pedidos como parte de uma transação distribuída precisam se comunicar uns com os outros para coordenar suas ações quando a transação é efetivada. Um cliente inicia uma transação enviando um pedido de *openTransaction* para um coordenador em qualquer servidor, conforme descrito na Seção 13.2. O coordenador contatado executa o pedido de *openTransaction* e retorna para o cliente o identificador de transação resultante. Os identificadores das transações distribuídas devem ser exclusivos dentro de um sistema distribuído. Uma maneira simples de obter isso é fazer com que um TID contenha duas partes: o identificador (por exemplo, um endereço IP) do servidor que o criou e um número exclusivo para o servidor.

O coordenador que abriu a transação torna-se o *coordenador* da transação distribuída e, no final, é responsável por efetivá-la, ou cancelá-la. Cada um dos servidores que gerencia um objeto acessado

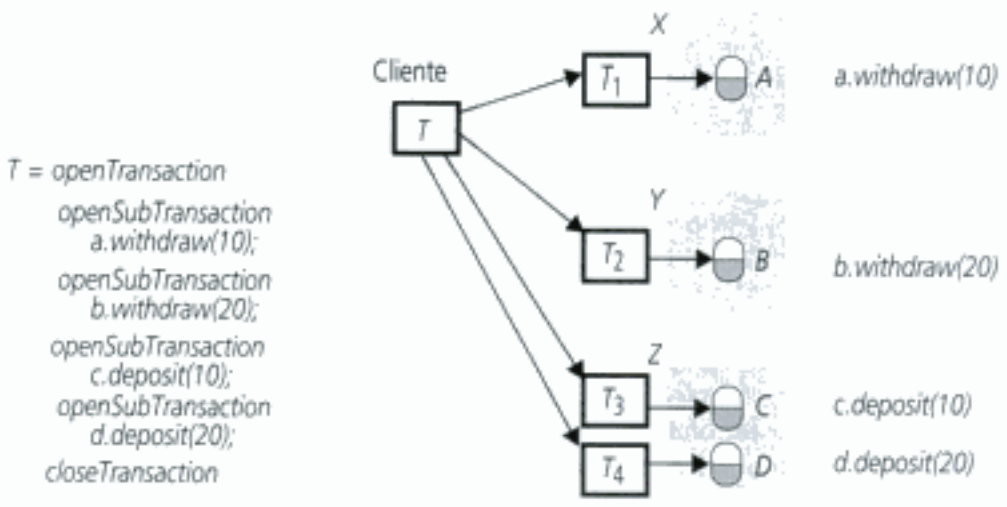


Figura 14.2 Transação bancária aninhada.

por uma transação é um participante da transação. Cada participante é responsável por rastrear todos os objetos recuperáveis envolvidos na transação nesse servidor. Os participantes são responsáveis por cooperar com o coordenador na execução do protocolo de efetivação.

Durante o andamento da transação o coordenador registra uma lista de referências nos participantes, e cada participante registra uma referência no coordenador.

A interface para *Coordinator*, mostrada na Figura 13.3, fornece um método adicional, *join*, usado quando um novo participante entra na transação:

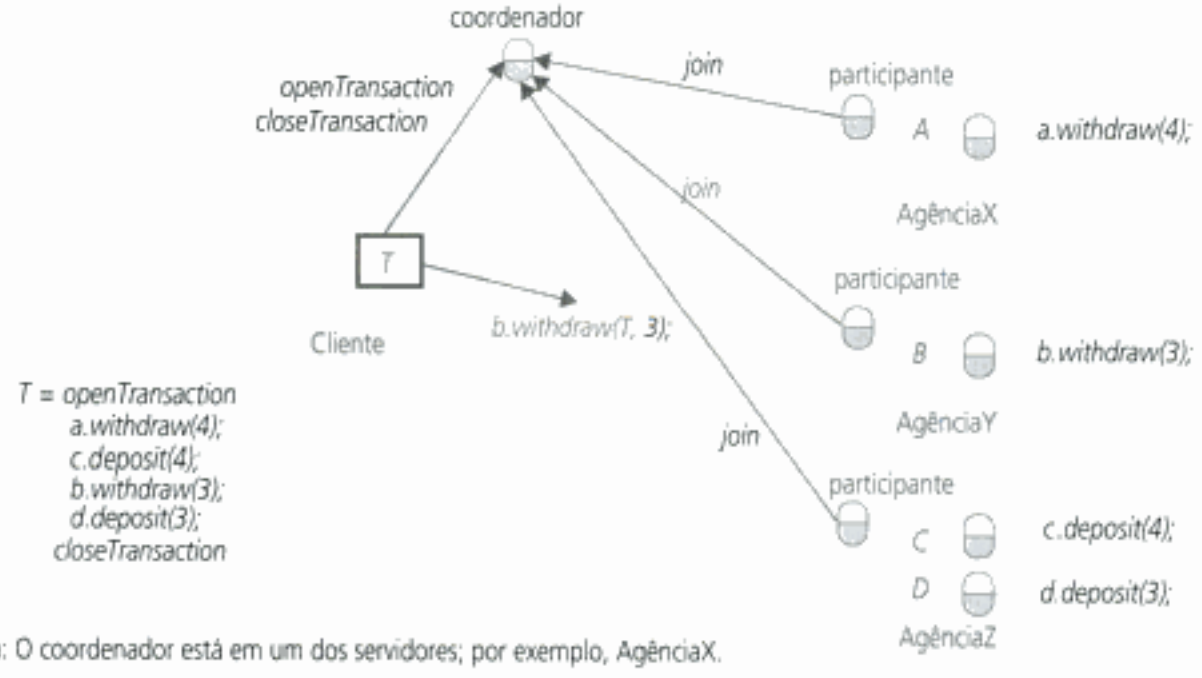
---

join(Trans, referência ao participante)
Informa ao coordenador que um novo participante entrou na transação *Trans*.

O coordenador registra o novo participante em sua lista de participantes. O fato de o coordenador conhecer todos os participantes, e cada participante conhecer o coordenador, permitirá que eles reúnam as informações que serão necessárias no momento da efetivação.

A Figura 14.3 mostra um cliente cuja transação bancária (plana) envolve as contas *A*, *B*, *C* e *D*, nos servidores AgênciaX, AgênciaY e AgênciaZ. A transação do cliente, *T*, transfere \$4 da conta *A* para a conta *C*, e depois transfere \$3 da conta *B* para a conta *D*. A transação descrita à esquerda é expandida para mostrar que *openTransaction* e *closeTransaction* são direcionados para o coordenador, o qual estará situado em um dos servidores envolvidos na transação. Cada servidor é mostrado com um participante, o qual entra na transação invocando o método *join* no coordenador. Quando o cliente invoca um dos métodos na transação, por exemplo, *b.withdraw(T, 3)*, o objeto que recebe a invocação (*B* em AgênciaY, neste caso) informa ao seu objeto participante que o objeto pertence à transação *T*. Se ainda não tiver informado ao coordenador, o objeto participante usa a operação *join* para fazer isso. Neste exemplo, mostramos o identificador de transação sendo passado como um argumento adicional, para que o destinatário possa passá-lo para o coordenador. Quando o cliente chama *closeTransaction*, o coordenador tem referências para todos os participantes.

Note que é possível um participante chamar *abortTransaction* no coordenador, se por algum motivo não for capaz de continuar com a transação.



Nota: O coordenador está em um dos servidores; por exemplo, AgênciaX.

Figura 14.3 Uma transação bancária distribuída.

14.3 Protocolos de efetivação atômica

Os protocolos de efetivação (*commit*) de transação foram inventados no início dos anos 70 e o protocolo de efetivação de duas fases apareceu em Gray [1978]. A atomicidade das transações exige que, quando uma transação distribuída chegar ao fim, todas as suas operações sejam executadas ou que nenhuma delas seja executada. No caso de uma transação distribuída, o cliente solicita as operações em mais de um servidor. Uma transação chega ao fim quando o cliente solicita que ela seja efetivada ou cancelada. Uma maneira simples de concluir a transação de maneira atômica é fazer com que o coordenador comunique o pedido de efetivação, ou cancelamento, para todos os participantes da transação e fique repetindo o pedido até que todos tenham reconhecido que foram levados adiante. Esse é um exemplo de *protocolo de efetivação atômica de uma fase*.

Esse protocolo de efetivação atômica de uma fase é inadequado, pois, quando o cliente solicita uma efetivação, ele não permite que um servidor tome a decisão unilateral de cancelar uma transação. Os motivos que impedem um servidor de efetivar sua parte de uma transação geralmente estão relacionados a problemas de controle de concorrência. Por exemplo, se estiver sendo usado bloqueio, a solução de um impasse poderá levar ao cancelamento de uma transação sem que o cliente saiba, a não ser que faça outro pedido para o servidor. Se estiver em uso o controle de concorrência otimista, a falha da validação em um servidor o faria decidir cancelar a transação. O coordenador pode não saber quando um servidor falhou e foi substituído durante o andamento de uma transação distribuída – tal servidor precisará cancelar a transação.

O *protocolo de efetivação de duas fases* é projetado de forma a permitir que qualquer participante cancele sua parte de uma transação. Devido ao requisito da atomicidade, se uma parte de uma transação for cancelada, a transação inteira também deverá ser cancelada. Na primeira fase do protocolo, cada participante vota na transação a ser efetivada ou cancelada. Quando um participante tiver votado na efetivação de uma transação, ele não poderá cancelá-la. Portanto, antes que um participante vote na efetivação de uma transação, ele deve garantir que finalmente poderá executar sua parte do protocolo de efetivação, mesmo que falhe e seja substituído nesse meio-tempo. Diz-se que um participante de uma transação está em um estado *preparado* para uma transação se finalmente puder efetivá-la. Para garantir isso, cada participante salva no meio de armazenamento permanente todos os objetos que tiver alterado na transação, junto com seu status – preparado.

Na segunda fase do protocolo, todos os participantes da transação tomam uma decisão conjunta. Se um participante votar pelo cancelamento, a decisão deverá ser o cancelamento da transação. Se todos os participantes votarem na efetivação, a decisão será a de efetivar a transação.

O problema é garantir que todos os participantes votem, e que todos cheguem à mesma decisão. Se nenhum erro ocorrer, isso é muito simples, mas o protocolo deve funcionar corretamente mesmo quando alguns dos servidores falham, mensagens são perdidas ou servidores são temporariamente incapazes de se comunicar uns com os outros.

Modelo de falha dos protocolos de efetivação ◊ A Seção 13.1.2 apresentou um modelo de falha para transações que se aplica igualmente ao protocolo de efetivação de duas fases (ou qualquer outro). Os protocolos de efetivação são feitos para funcionar em um sistema assíncrono, no qual os servidores podem falhar e mensagens podem ser perdidas. Presume-se que um protocolo de solicitação e resposta subjacente remova as mensagens corrompidas e replicadas. Não existem falhas bizantinas – ou os servidores falham ou obedecem as mensagens enviadas.

O protocolo de efetivação de duas fases é um exemplo de protocolo para chegar a um consenso. O Capítulo 12 declarou que o consenso não pode ser atingido em um sistema assíncrono, caso os processos às vezes falhem. Entretanto, o protocolo de efetivação de duas fases chega ao consenso sob essas condições. Isso porque as falhas por colapso dos processos são mascaradas pela substituição de um processo falho por um novo processo, cujo estado é configurado a partir das informações gravadas no meio de armazenamento permanente e das informações mantidas por outros processos.

14.3.1 O protocolo de efetivação de duas fases

Durante o andamento de uma transação, não há nenhuma comunicação entre o coordenador e os participantes, a não ser os participantes informando o coordenador quando entram na transação. O pedido

Hidden page

Hidden page

Outro ponto no qual um participante pode ser retardado é quando tiver executado todos os pedidos de seu cliente na transação, mas ainda não tiver recebido uma chamada de *canCommit* do coordenador. Quando o cliente envia o pedido de *closeTransaction* para o coordenador, um participante só pode detectar tal situação se notar que não tinha um pedido em uma transação específica por um longo tempo; por exemplo, por um período de tempo limite sobre uma trava. Como nenhuma decisão foi tomada nesse estágio, o participante pode decidir cancelar unilateralmente, após algum período de tempo.

O coordenador pode ser retardado quando estiver esperando pelos votos dos participantes. Como ele ainda não decidiu o destino da transação, pode decidir cancelá-la após algum período de tempo. Ele deve então anunciar *doAbort* aos participantes que já enviaram seus votos. Alguns participantes lentos podem tentar votar em *Sim* depois disso, mas seus votos serão ignorados e eles entrarão no estado *de incerteza*, conforme descrito anteriormente.

Desempenho do protocolo de efetivação de duas fases ◊ Desde que tudo corra bem – isto é, que o coordenador, os participantes e a comunicação entre eles não falhem, o protocolo de efetivação de duas fases envolvendo N participantes pode ser concluído com N mensagens e respostas *canCommit*, seguidas de N mensagens *doCommit*. Ou seja, o custo nas mensagens é proporcional a $3N$ e o custo no tempo é o de três rodadas de mensagens. As mensagens *haveCommitted* não são contadas no custo estimado do protocolo, que pode funcionar corretamente sem elas – sua função é permitir que os servidores excluam informações velhas do coordenador.

No pior caso, pode existir arbitrariamente muitas falhas de servidor e de comunicação durante o protocolo de efetivação de duas fases. Entretanto, o protocolo é feito para tolerar uma sucessão de falhas (falhas de servidor ou mensagens perdidas) e é garantido que finalmente seja concluído, embora não seja possível especificar um limite de tempo dentro do qual ele terminará.

Conforme mencionado na seção sobre tempos limites, o protocolo de efetivação de duas fases pode causar atrasos consideráveis para participantes no estado *de incerteza*. Esses atrasos ocorrem quando o coordenador falhou e não pode responder aos pedidos de *getDecision* dos participantes. Mesmo que um protocolo cooperativo permita aos participantes fazerem pedidos de *getDecision* para outros participantes, ocorrerão atrasos se todos os participantes ativos estiverem *incertos*.

Foram projetados protocolos de efetivação de três fases para diminuir tais atrasos. Eles são mais dispendiosos no número de mensagens e no número de rodadas exigidas para o caso normal (livre de falhas). Para ver uma descrição dos protocolos de efetivação de três fases, consulte o Exercício 14.2 e Bernstein *et al.* [1987].

14.3.2 Protocolo de efetivação de duas fases para transações aninhadas

A transação mais externa em um conjunto de transações aninhadas é chamada de *transação de nível superior*. As outras transações são chamadas de *subtransações*. Na Figura 14.1(b), T é a transação de nível superior, T_1, T_2, T_{11}, T_{12}, T_{21} e T_{22} são subtransações. T_1 e T_2 são transações descendentes de T, que é referida como sua ascendente. Analogamente, T_{11} e T_{12} são transações descendentes de T_1, e T_{21} e T_{22} são transações descendentes de T_2. Cada subtransação começa depois de sua ascendente e termina antes dela. Assim, por exemplo, T_{11} e T_{12} começam depois de T_1 e terminam antes dela.

Quando uma subtransação termina, ela toma uma decisão, independente de ser efetivada provisoriamente ou cancelada. Uma efetivação provisória é diferente de estar preparada para efetivar: não é feito uma escrita no meio de armazenamento permanente. Se subseqüentemente o servidor falhar, seu substituto não poderá efetivar. Após todas as subtransações terem terminado, aquelas que foram provisoriamente efetivadas participam de um protocolo de efetivação de duas fases, no qual os servidores das subtransações efetivadas provisoriamente expressam sua intenção de efetivar e aquelas com uma ancestral cancelada serão canceladas. Uma efetivação preparada garante que uma subtransação poderá ser efetivada, enquanto que uma efetivação provisória significa apenas que ela terminou corretamente – e provavelmente concordará em ser efetivada, quando for solicitada a fazer isso.

Um coordenador de uma subtransação fornecerá uma operação para abrir uma subtransação junto com uma operação que permite ao coordenador da subtransação perguntar se sua ascendente já foi efetivada ou cancelada, como mostra a Figura 14.7.

openSubTransaction(trans) → *subTrans*
Abre uma nova subtransação cuja ascendente é *trans* e retorna um identificador de subtransação exclusivo.

getStatus(trans) → *efetivada, cancelada, provisória*
Pede ao coordenador para que informe o status da transação *trans*. Retorna valores representando uma das seguintes opções: *efetivada, cancelada, provisória*.

Figura 14.7 Operações no coordenador de transações aninhadas.

Um cliente inicia um conjunto de transações aninhadas abrindo uma transação de nível superior com uma operação *openTransaction*, a qual retorna um identificador de transação para a transação de nível superior. O cliente inicia uma subtransação ativando a operação *openSubTransaction*, cujo argumento especifica sua transação ascendente. A nova subtransação se *junta* automaticamente à transação ascendente e é retornado um identificador de transação para a subtransação.

O identificador de uma subtransação deve ser uma extensão do TID de sua ascendente construído de tal modo que o identificador da transação de nível superior, ou ascendente dessa subtransação, possa ser determinado a partir de seu próprio identificador de transação. Além disso, todos os identificadores de subtransação devem ser globalmente exclusivos. O cliente termina um conjunto de transações aninhadas invocando *closeTransaction*, ou *abortTransaction*, no coordenador da transação de nível superior.

Nesse meio tempo, cada uma das transações aninhadas executa suas operações. Quando elas tiverem terminado, o servidor que estiver gerenciando a subtransação registrará informações quanto ao fato da subtransação ter sido efetivada provisoriamente ou ter sido cancelada. Note que, se sua ascendente for cancelada, a subtransação também será obrigada a ser cancelada.

Lembre-se, do Capítulo 13, que uma transação ascendente – incluindo uma transação de nível superior – pode ser efetivada mesmo que uma de suas subtransações descendentes tenha sido cancelada. Em tais casos, a transação ascendente será programada para executar ações diferentes, de acordo com o fato de uma subtransação ser efetivada ou cancelada. Por exemplo, considere uma transação bancária projetada para executar todos os pedidos de posição de uma agência em um dia específico. Essa transação é expressa como várias subtransações *Transfer* aninhadas, cada uma das quais decomposta em subtransações *deposit* e *withdraw* aninhadas. Supomos que, quando uma conta está sem fundos, *withdraw* é cancelada e, então, a subtransação *Transfer* correspondente é cancelada. Mas não há necessidade de cancelar todos os pedidos de posição apenas porque uma subtransação *Transfer* foi cancelada. Em vez de cancelar, a transação de nível superior notará as subtransações *Transfer* que foram canceladas e executará as ações apropriadas.

Considere a transação de nível superior T e suas subtransações, mostradas na Figura 14.8, que é baseada na Figura 14.1(b). Cada subtransação foi provisoriamente efetivada ou cancelada. Por exemplo, T_{12} foi provisoriamente efetivada e T_{11} foi cancelada, mas o destino de T_{12} depende de sua ascendente T_1 e, finalmente, da transação de nível superior, T. Embora T_{21} e T_{22} tenham sido provisoriamente

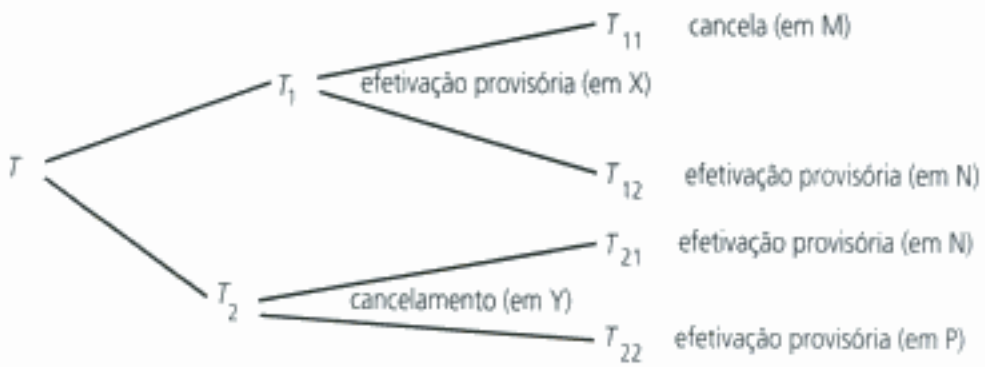


Figura 14.8 A transação T decide se vai ser efetivada.

Hidden page

Hidden page

Hidden page

Hidden page

T		*U*	
Read(A)	em *X*	*Read(B)*	em *Y*
Write(A)		*Write(B)*	
Read(B)	em *Y*	*Read(A)*	em *X*
Write(B)		*Write(A)*	

As transações acessam os objetos na ordem *T* antes de *U* no servidor *X*, e na ordem *U* antes de *T* no servidor *Y*. Agora, suponha que *T* e *U* iniciem a validação praticamente ao mesmo tempo, mas que o servidor *X* valide *T* primeiro e o servidor *Y* valide *U* primeiro. Lembre-se de que a Seção 13.5 recomendava uma simplificação do protocolo de validação, que produzia uma regra dizendo que apenas uma transação por vez poderia executar as fases de validação e atualização. Portanto, cada servidor não poderá validar a outra transação até que a primeira tenha terminado. Esse é um exemplo de impasse de efetivação.

As regras de validação da Seção 13.5 presumem que a validação é rápida, o que é verdade para transações com um único servidor. Entretanto, em uma transação distribuída, o protocolo de efetivação de duas fases pode levar algum tempo e retardar a entrada de outras transações na validação até que uma decisão sobre a transação corrente seja tomada. Nas transações distribuídas otimistas, cada servidor aplica um protocolo de validação em paralelo. Essa é uma extensão da validação para trás, ou para frente, para permitir que várias transações estejam na fase de validação ao mesmo tempo. Nessa ampliação, a regra 3 deve ser verificada, assim como a regra 2 de validação para trás. Isto é, o conjunto de escrita da transação que está sendo validada deve ser verificado para saber se existem sobreposições com o conjunto de escrita de transações sobrepostas anteriores. Kung e Robinson [1981] descrevem a validação em paralelo em seu artigo.

Se for usada a validação em paralelo, as transações não sofrerão do impasse de efetivação. Entretanto, se os servidores simplesmente realizarem validações independentes, é possível que diferentes servidores de uma transação distribuída possam dispor em série o mesmo conjunto de transações, em ordens diferentes; por exemplo, com *T* antes de *U* no servidor *X*, e *U* antes de *T* no servidor *Y*, em nosso exemplo.

Os servidores de transações distribuídas devem impedir que isso aconteça. Uma estratégia é a realização de uma validação global após uma validação local em cada servidor [Ceri e Owicki 1982]. A validação global verifica se a combinação das ordens nos servidores individuais pode ser organizada em série; isto é, se a transação que está sendo validada não está envolvida em um ciclo.

Outra estratégia é que todos os servidores de uma transação em particular utilizem o mesmo número de transação globalmente exclusivo no início da validação [Schlageter 1982]. O coordenador do protocolo de efetivação de duas fases é responsável por gerar o número de transação globalmente exclusivo e passá-lo para os participantes nas mensagens de *canCommit*. Como diferentes servidores podem coordenar diferentes transações, os servidores devem ter uma ordem acordada para os números de transação que geram (como no protocolo de ordenação da indicação de tempo distribuída).

Agrawal *et al.* [1987] propuseram uma variação do algoritmo de Kung e Robinson que favorece as transações somente de leitura, junto com um algoritmo chamado MVGV (*Multi-Version Generalized Validation* – validação generalizada de versão múltipla). O MVGV é uma forma de validação em paralelo que garante que os números de transação reflitam a ordem serial, mas ele exige que a visibilidade de algumas transações seja retardada, após terem sido efetivadas. Ele também permite que o número de transação seja alterado, para deixar que algumas transações validem aquelas que, de outro modo, teriam falhado. O artigo também propõe um algoritmo para efetivar transações distribuídas. Ele é semelhante à proposta de Schlageter, no sentido de que um número de transação global precisa ser encontrado. No final da fase de leitura, o coordenador propõe um valor para o número de transação global, e cada participante tenta validar suas transações locais usando esse número. Entretanto, se o número de transação global proposto for pequeno demais, alguns participantes podem não conseguir validar suas transações e negociam um número maior com o coordenador. Se nenhum número conveniente puder ser encontrado, então esses participantes terão que cancelar suas transações. Finalmente, se todos os participantes puderem validar suas transações, o coordenador terá recebido propostas de números de transação de cada um deles. Se puderem ser encontrados números comuns, a transação será efetivada.

14.5 Impasses distribuídos

A discussão sobre impasses da Seção 13.4 mostrou que eles podem surgir dentro de um único servidor quando é usado bloqueio para controle de concorrência. Os servidores devem impedir, ou detectar e resolver os impasses. Usar tempos limites para solucionar possíveis impasses é uma estratégia deselegante – é difícil escolher um intervalo de tempo limite apropriado e transações são canceladas desnecessariamente. Com os esquemas de detecção de impasse, uma transação só é cancelada quando está realmente envolvida em um impasse. A maioria dos esquemas de detecção de impasse funciona encontrando ciclos no grafo de "espera por" da transação. Teoricamente, em um sistema distribuído envolvendo vários servidores sendo acessados por múltiplas transações, um grafo de "espera por" global pode ser construído a partir dos grafos locais. Pode haver um ciclo no grafo de "espera por" global que não aparece em nenhum outro grafo local – isto é, pode haver um *impasse distribuído*. Lembre-se de que o grafo de "espera por" é um grafo direcionado no qual os nós representam transações e objetos, e os arcos representam um objeto retido por uma transação, ou uma transação esperando por um objeto. Existe um impasse se, e somente se, existe um ciclo no grafo de "espera por".

A Figura 14.12 mostra as interposições das transações *U*, *V* e *W* envolvendo os objetos *A* e *B* gerenciados pelos servidores *X* e *Y* e os objetos *C* e *D* gerenciados pelo servidor *Z*.

O grafo de "espera por" completo da Figura 14.13(a) mostra que um ciclo de impasse consiste em arcos alternados, os quais representam uma transação esperando por um objeto e um objeto retido por uma transação. Como qualquer transação só pode estar esperando um objeto por vez, os objetos podem ser omitidos dos grafos de "espera por", como mostrado na Figura 14.13(b).

A detecção de um impasse distribuído exige que um ciclo seja encontrado no grafo de "espera por" de transação global, que é distribuído entre os servidores que estavam envolvidos nas transações. Os grafos de "espera por" locais podem ser construídos pelo gerenciador de travas em cada servidor, conforme discutido no Capítulo 13. No exemplo anterior, os grafos de "espera por" locais dos servidores são:

servidor *Y*: $U \rightarrow V$ (adicionado quando *U* solicita *b.withdraw(30)*)

servidor *Z*: $V \rightarrow W$ (adicionado quando *V* solicita *c.withdraw(20)*)

servidor *X*: $W \rightarrow U$ (adicionado quando *W* solicita *a.withdraw(20)*)

Como o grafo de "espera por" global é mantido parcialmente em cada um dos vários servidores envolvidos, é exigida uma comunicação entre esses servidores para encontrar ciclos no grafo.

Uma solução simples é usar detecção de impasse centralizada, na qual um servidor assume o papel de detector de impasse global. De tempos em tempos, cada servidor envia a cópia mais recente de seu grafo de "espera por" local para o detector de impasse global, o qual reúne as informações dos grafos locais para construir um grafo de "espera por" global. O detector de impasse global verifica a existência de ciclos no grafo de "espera por" global. Quando encontra um ciclo, ele toma uma decisão sobre como resolver o impasse e informa os servidores sobre a transação a ser cancelada para resolver o impasse.

U		*V*		*W*	
d.deposit(10)	trava *D*				
		b.deposit(10)	trava *B*		
a.deposit(20)	trava *A*		em *Y*		
	em *X*				
				c.deposit(30)	trava *C*
b.withdraw(30)	espera em *Y*				em *Z*
		c.withdraw(20)	espera em *Z*		
				a.withdraw(20)	espera em *X*

Figura 14.12 Interposições das transações *U*, *V* e *W*.

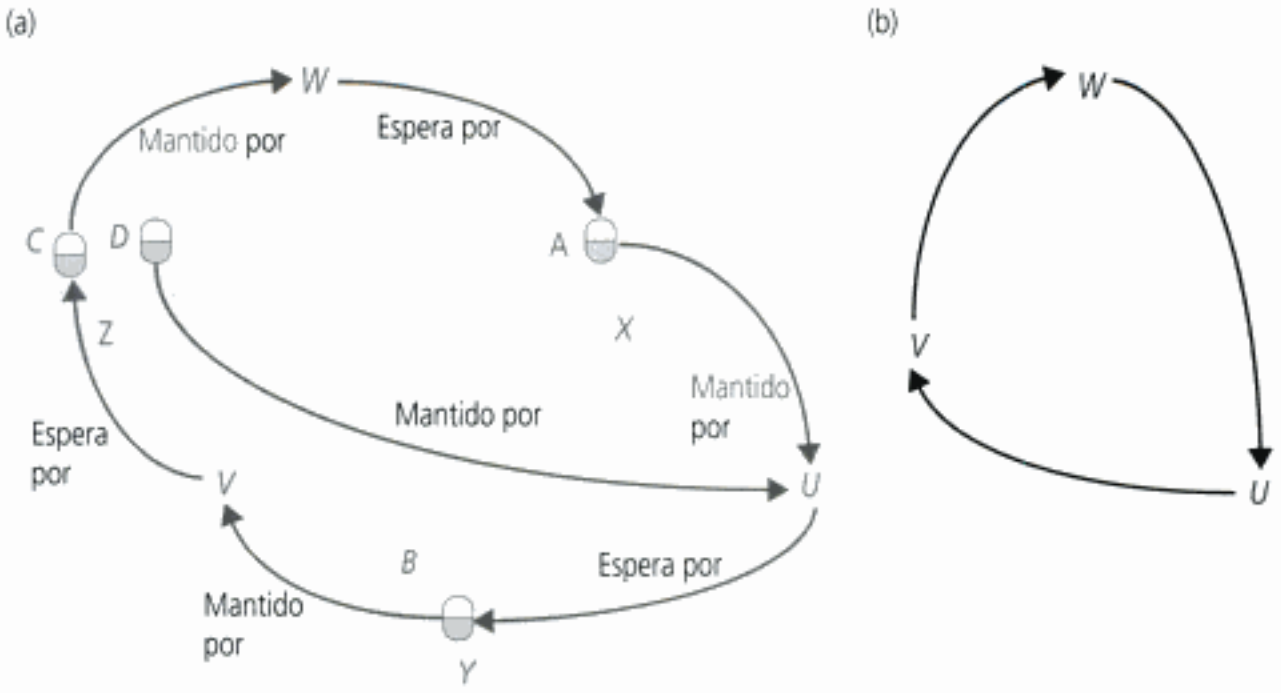


Figura 14.13 Impasse distribuído.

A detecção de impasse centralizada não é uma boa idéia, pois depende de um único servidor para executá-la. Ela sofre dos problemas comuns associados às soluções centralizadas em sistemas distribuídos – disponibilidade deficiente, falta de tolerância a falha e nenhuma escalabilidade. Além disso, o custo da transmissão freqüente de grafos de "espera por" locais é alto. Se o grafo global for montado com menos freqüência, os impasses poderão demorar mais para serem detectados.

Impasses fantasmas ◊ Um impasse que é detectado, mas não é realmente um impasse, é chamado de impasse fantasma. Na detecção de impasse distribuído, as informações sobre relacionamentos de "espera por" entre as transações são transmitidas de um servidor para outro. Se houver um impasse, as informações necessárias serão finalmente reunidas em um só lugar e um ciclo será detectado. Como esse procedimento levará algum tempo, existe a chance de que uma das transações que mantém uma trava a tenha liberado nesse meio-tempo, no caso em que o impasse não existirá mais.

Considere o caso de um detector de impasse global que recebe grafos de "espera por" locais dos servidores X e Y, como mostrado na Figura 14.14. Suponha que a transação U libere então um objeto no servidor X e solicite o objeto que é mantido por V no servidor Y. Suponha também que o detector global receba o grafo local do servidor Y antes do grafo do servidor X. Nesse caso, ele detectaria um ciclo $T \rightarrow U \rightarrow V \rightarrow T$, embora a seta $T \rightarrow U$ não exista mais. Esse é um exemplo de impasse fantasma.

O leitor atento terá percebido que, se as transações estão usando travas de duas fases, elas não podem liberar objetos e depois obter mais objetos, e ciclos de impasse fantasma não podem ocorrer da

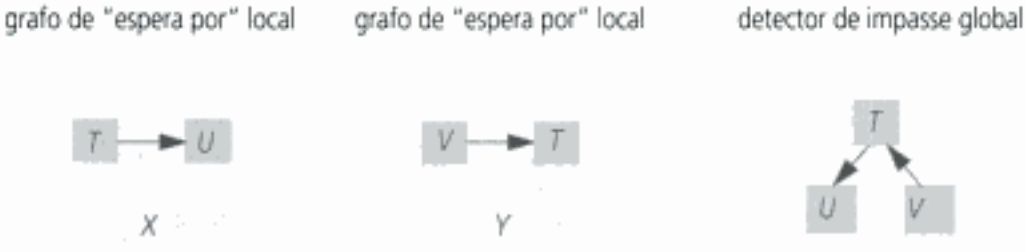


Figura 14.14 Grafos de "espera por" local e global.

Hidden page

Hidden page

O algoritmo anterior deve encontrar qualquer ocorrência de impasse, desde que as transações que estão esperando não sejam canceladas, e que não existam falhas como mensagens perdidas ou servidores falhando. Para entender isso, considere um ciclo de impasse no qual a última transação, W, começa a esperar e completa o ciclo. Quando W começa a esperar por um objeto, o servidor inicia uma mensagem de sondagem (*probe*) que vai para o servidor do objeto retido por cada transação pela qual W está esperando. Os destinatários estendem e encaminham as mensagens de sondagem para os servidores dos objetos solicitados por todas as transações que estão esperando que eles encontrarem. Assim, toda transação pela qual W espera, direta ou indiretamente, será adicionada na mensagem de sondagem, a não ser que um impasse seja detectado. Quando há um impasse, W está esperando por si mesma indiretamente. Portanto, a mensagem de sondagem retornará o objeto que W contém.

Em princípio, um grande número de mensagens é enviado para detectar um impasse. No exemplo anterior, vimos duas mensagens de sondagem para detectar um ciclo envolvendo três transações. Em geral, cada uma das mensagens de sondagem é, na verdade, duas mensagens (do objeto para o coordenador e depois do coordenador para o objeto).

Uma mensagem de sondagem que detecta um ciclo envolvendo N transações será encaminhada por $(N-1)$ coordenadores de transação por intermédio de $(N-1)$ servidores de objetos, exigindo $2(N-1)$ mensagens. Felizmente, a maioria dos impasses envolve ciclos contendo apenas duas transações, e não há necessidade de preocupação excessiva com o número de mensagens envolvidas. Essa observação foi feita a partir dos estudos dos bancos de dados. Ela também pode ser demonstrada considerando-se a probabilidade de acesso conflitante aos objetos. Consulte Bernstein *et al.* [1987].

Prioridades de transação ◊ No algoritmo anterior, toda transação envolvida em um ciclo de impasse pode causar o início da detecção de impasse. O efeito de várias transações iniciando a detecção de impasse é que esse procedimento pode acontecer em vários servidores diferentes que participam do ciclo, com o resultado de que mais de uma transação pode vir a ser cancelada.

Na Figura 14.16(a), considere as transações T, U, V e W, onde U está esperando por W e V está esperando por T. Praticamente ao mesmo tempo, T solicita o objeto mantido por U e W solicita o objeto mantido por V. Duas mensagens de sondagem separadas $<T \rightarrow U>$ e $<W \rightarrow V>$ são iniciadas pelos servidores desses objetos e circulam até que o impasse seja detectado por cada um dos dois diferentes servidores. Veja na Figura 14.16(b), onde o ciclo é $<T \rightarrow U \rightarrow W \rightarrow V \rightarrow T>$, e em (c), onde o ciclo é $<W \rightarrow V \rightarrow T \rightarrow U \rightarrow W>$.

Para garantir que apenas uma transação em um ciclo seja cancelada, as transações recebem *prioridades*, de maneira tal que todas as transações são totalmente ordenadas. Indicações de tempo, por exemplo, podem ser usadas como prioridades. Quando é encontrado um ciclo de impasse, a transação com a menor prioridade é cancelada. Mesmo que vários servidores diferentes detectem o mesmo ciclo, todos chegarão à mesma decisão com relação a qual transação deve ser cancelada. Escrevemos T

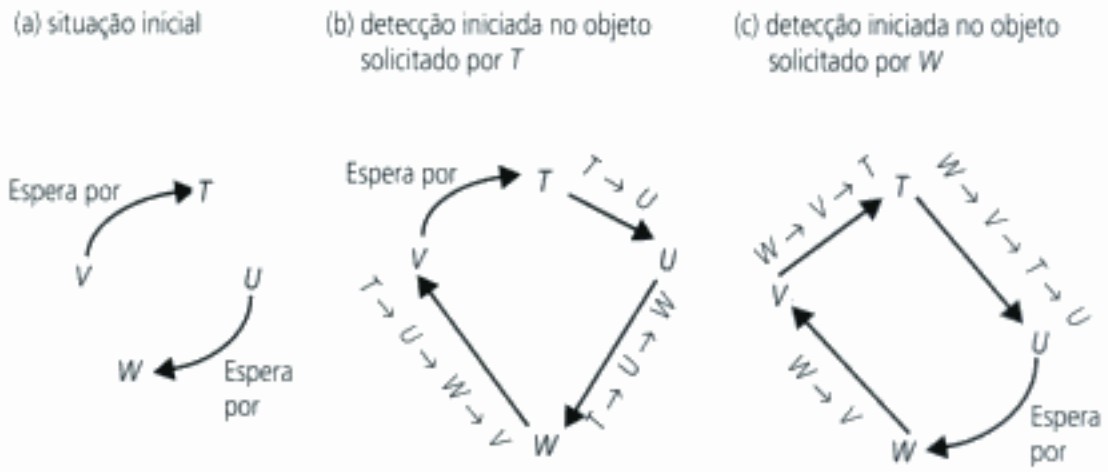


Figura 14.16 Duas mensagens de sondagem disparadas.

$> U$ para indicar que T tem prioridade mais alta do que U. No exemplo anterior, suponha que $T > U > V > W$. Então, a transação W será cancelada quando um dos ciclos $<T \rightarrow U \rightarrow W \rightarrow V \rightarrow T>$ ou $<W \rightarrow V \rightarrow T \rightarrow U \rightarrow W>$ for detectado.

As prioridades de transação também poderiam ser usadas para reduzir o número de situações que causam o início da detecção de impasse, usando a regra de que a detecção é iniciada apenas quando uma transação de prioridade mais alta começa a esperar por outra de prioridade mais baixa. Em nosso exemplo da Figura 14.16, como $T > U$, a mensagem de sondagem $<T \rightarrow U>$ seria enviada, mas como $W < V$, a mensagem de sondagem $<W \rightarrow V>$ não seria enviada. Se supusermos que quando uma transação começa a esperar por outra transação, é igualmente provável que a transação que está esperando tenha prioridade mais alta, ou mais baixa, do que a transação pela qual se espera, então o uso dessa regra provavelmente reduz praticamente pela metade o número de mensagens de sondagem.

As prioridades de transação também poderiam ser usadas para reduzir o número de mensagens de sondagem encaminhadas. A idéia geral é que as mensagens de sondagem devem 'descer a ladeira' – isto é, das transações com prioridades mais altas para as transações com prioridades mais baixas. Para fazer isso, os servidores usam a regra de que eles não encaminham nenhuma mensagem de sondagem para um proprietário que tenha prioridade mais alta do que o iniciador. O argumento para se fazer isso é que, se o proprietário estiver esperando por outra transação, então deverá ter iniciado a detecção enviando uma mensagem de sondagem quando começou a esperar.

Entretanto, há uma armadilha associada a essas aparentes melhorias. Em nosso exemplo da Figura 14.15, as transações U, V e W são executadas em uma ordem na qual U está esperando por V e V está esperando por W, quando W começa a esperar por U. Sem regras de prioridade, a detecção é iniciada quando W começa a esperar, enviando uma mensagem de sondagem $<W \rightarrow U>$. Sob a regra da prioridade, essa mensagem de sondagem não será enviada, pois $W < U$ e o impasse não será detectado.

O problema é que a ordem em que as transações começam a esperar pode determinar se o impasse será detectado ou não. A armadilha acima pode ser evitada usando-se um esquema em que os coordenadores salvam cópias de todas as mensagens de sondagem recebidas em nome de cada transação, em uma fila de mensagens de sondagem. Quando uma transação começa a esperar por um objeto, ela encaminha as mensagens de sondagem dessa fila para o servidor do objeto, o qual as propaga nas rotas "ladeira abaixo".

Em nosso exemplo da Figura 14.15, quando U começar a esperar por V, o coordenador de V salvará a mensagem de sondagem $<U \rightarrow V>$. Veja a Figura 14.17(a). Então, quando V começar a esperar por W, o coordenador de W armazenará $<V \rightarrow W>$ e V encaminhará sua fila de mensagens de sondagem $<U \rightarrow V>$ para W. Veja a Figura 14.17(b), na qual a fila de mensagens de sondagem de W tem $<U \rightarrow V>$ e $<V \rightarrow W>$. Quando W começar a esperar por A, encaminhará sua fila de mensagens de sondagem $<U \rightarrow V \rightarrow W>$ para o servidor de A, o qual também notará a nova dependência $W \rightarrow U$ e a combinará com as informações das mensagens de sondagem recebidas para determinar que $U \rightarrow V \rightarrow W \rightarrow U$. O impasse foi detectado.

Quando um algoritmo exige que as mensagens de sondagem sejam armazenadas em filas, ele também exige providências para passar essas mensagens para novos proprietários e para descartar as mensagens que se refiram às transações efetivadas ou canceladas. Se mensagens de sondagens

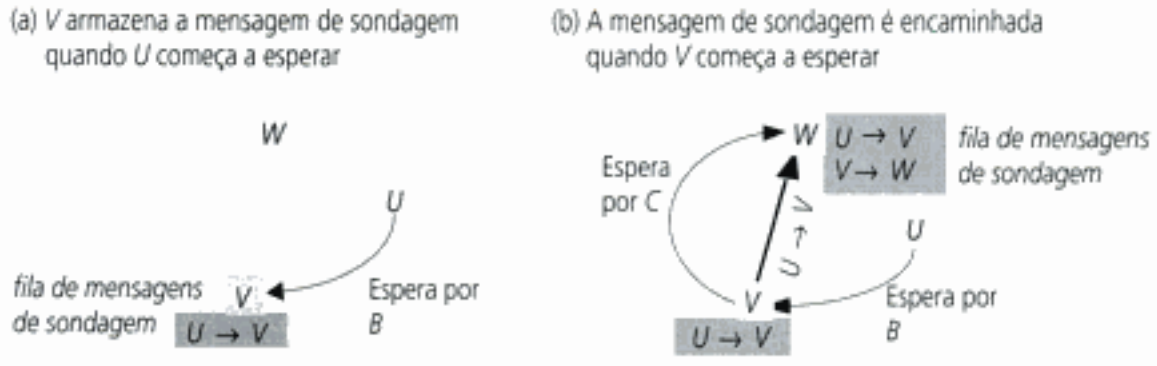


Figura 14.17 As mensagens de sondagem são enviadas "ladeira abaixo".

relevantes forem descartadas, poderão ocorrer impasses não detectados, e se mensagens de sondagem obsoletas forem mantidas poderão ser detectados impasses falsos. Isso aumenta muito a complexidade de qualquer algoritmo caminhamento pelas arestas. Os leitores que estiverem interessados nos detalhes de tais algoritmos devem consultar Sinha e Natarajan [1985] e Choudhary *et al.* [1989], que apresentam algoritmos para uso com travas exclusivas. Mas eles verão que Choudhary *et al.* mostraram que o algoritmo de Sinha e Natarajan está incorreto, não detecta todos os impasses e pode até relatar falsos impasses. Kshemkalyani e Singhal [1991] corrigiram o algoritmo de Choudhary *et al.* (que não detecta todos os impasses e pode relatar impasses falsos) e fornecem uma prova da exatidão do algoritmo corrigido. Em um artigo subseqüente, Kshemkalyani e Singhal [1994] argumentam que os impasses distribuídos não são muito bem entendidos, pois não existe nenhum estado, ou tempo global, em um sistema distribuído. Na verdade, qualquer ciclo reunido pode conter seções registradas em diferentes momentos. Além disso, os sites podem ouvir falar sobre impasses, mas não saber que eles foram resolvidos, até depois de terem atrasos aleatórios. O artigo descreve os impasses distribuídos em termos do conteúdo da memória distribuída, usando relacionamentos causais entre eventos em diferentes sites.

14.6 Recuperação de transações

A propriedade atômica das transações exige que os efeitos de todas as transações efetivadas, e nenhum dos efeitos das transações incompletas, ou canceladas, sejam refletidos nos objetos que elas acessaram. Essa propriedade pode ser descrita em termos de dois aspectos: durabilidade e atomicidade da falha. A durabilidade exige que os objetos sejam escritos no meio de armazenamento permanente e daí em diante estejam disponíveis indefinidamente. Portanto, uma confirmação de um pedido de efetivação de um cliente implica que todos os efeitos da transação foram armazenados no meio permanente, assim como nos objetos (voláteis) do servidor. A atomicidade da falha exige que os efeitos das transações sejam atômicos, mesmo que o servidor falhe. A recuperação está relacionada a garantir que os objetos de um servidor sejam duráveis e que o serviço forneça atomicidade de falha.

Embora os servidores de arquivo e de banco de dados mantenham dados em meios permanentes, outros tipos de servidores de objetos recuperáveis não precisam fazer isso, exceto para propósitos de recuperação. Neste capítulo, presumimos que, quando um servidor está sendo executado, ele mantém todos os seus objetos em sua memória volátil e registra os objetos efetivados em um (ou mais) *arquivo de recuperação*. Portanto, a recuperação consiste em restaurar o servidor com as versões mais recentemente efetivadas de seus objetos, a partir do meio de armazenamento permanente. Os bancos de dados precisam tratar com grandes volumes de dados. Eles geralmente mantêm os objetos em armazenamento estável no disco, com uma cache na memória volátil.

Os dois requisitos de durabilidade e de atomicidade de falha não são independentes um do outro e podem ser tratados com um único mecanismo – o *gerenciador de recuperação*. As tarefas de um gerenciador de recuperação são:

- salvar objetos das transações efetivadas em um meio permanente (em um arquivo de recuperação);
- restaurar os objetos do servidor após uma falha;
- reorganizar o arquivo de recuperação para melhorar o desempenho da recuperação;
- recuperar espaço de armazenamento (no arquivo de recuperação).

Em alguns casos, é exigido que o gerenciador de recuperação trate rapidamente de falhas de mídia – falhas de seu arquivo de recuperação, de modo que alguns dados do disco sejam perdidos, ou por se tornarem corrompidos durante uma falha, deteriorando-se aleatoriamente, ou ainda, devido a uma falha permanente. Em tais casos, precisamos de outra cópia do arquivo de recuperação. Isso pode ser feito no armazenamento estável, que é implementado de forma a uma falha ser muito improvável, usando discos espelhados ou cópias em um local diferente.

Hidden page

14.6.1 *Log*

Na técnica do *log*, o arquivo de recuperação é um registro contendo o histórico de todas as transações realizadas por um servidor. O histórico consiste nos valores dos objetos, em entradas de status da transação e nas listas de intenções das transações. A ordem das entradas no *log* reflete a ordem em que as transações foram preparadas, efetivadas e canceladas nesse servidor. Na prática, o arquivo de recuperação conterá um instantâneo recente dos valores de todos os objetos presentes no servidor, seguido de um histórico das transações após o instantâneo.

Durante a operação normal de um servidor, seu gerenciador de recuperação é chamado quando uma transação se prepara para efetivar, efetivando-a ou cancelando-a. Quando o servidor está preparado para efetivar uma transação, o gerenciador de recuperação anexa todos os objetos de sua lista de intenções no arquivo de recuperação, seguidos do status corrente dessa transação (*preparada*), junto com sua lista de intenções. Quando uma transação é finalmente efetivada, ou cancelada, o gerenciador de recuperação anexa o status correspondente da transação em seu arquivo de recuperação.

Presume-se que a operação de anexação seja atômica, pois ela grava uma, ou mais, entradas completas no arquivo de recuperação. Se o servidor falhar, apenas a última gravação poderá estar incompleta. Para fazer uso eficiente do disco, várias gravações subseqüentes podem ser colocadas em um buffer e, então, escritas no disco como uma única operação. Uma vantagem adicional da técnica de *log* é que escritas seqüenciais no disco são mais rápidas do que as escritas randômicas.

Após uma falha, toda transação que não tenha o status de *efetivada* no *log* é cancelada. Portanto, quando uma transação é efetivada, sua entrada de status *efetivada* deve ser *forçada* no *log* – isto é, escrita no *log*, junto com todas as outras entradas colocadas no buffer.

O gerenciador de recuperação associa um identificador exclusivo a cada objeto para que as sucessivas versões de um objeto no arquivo de recuperação possam ser associadas aos objetos do servidor. Por exemplo, uma forma durável de referência de objeto remoto, como uma referência persistente do CORBA, servirá como identificador de objeto.

A Figura 14.19 ilustra o mecanismo de *log* para as transações do serviço de transações bancárias *T* e *U* da Figura 13.7. O *log* foi recentemente reorganizado e as entradas à esquerda da linha dupla representam um instantâneo dos valores de *A*, *B* e *C*, antes que as transações *T* e *U* começassem. Nesse diagrama, usamos os nomes *A*, *B* e *C* como identificadores exclusivos dos objetos. Mostramos a situação de quando a transação *T* foi efetivada e a transação *U* estava preparada, mas não tinha sido efetivada. Quando a transação *T* se prepara para ser efetivada, os valores dos objetos *A* e *B* são gravados nas posições P_1 e P_2 do *log*, seguidos de uma entrada de status de transação preparada para *T*, com sua lista de intenções (<*A*, P_1>, <*B*, P_2>). Quando a transação *T* é efetivada, uma entrada status de transação efetivada para *T* é colocada na posição P_4. Então, quando a transação *U* se prepara para ser efetivada, os valores dos objetos *C* e *B* são gravados nas posições P_5 e P_6 do *log*, seguidos de uma entrada de status de transação preparada para *U*, com sua lista de intenções (<*C*, P_5>, <*B*, P_6>).

Cada entrada de status de transação contém um ponteiro para a posição no arquivo de recuperação da entrada de status de transação anterior, para permitir que o gerenciador de recuperação siga a

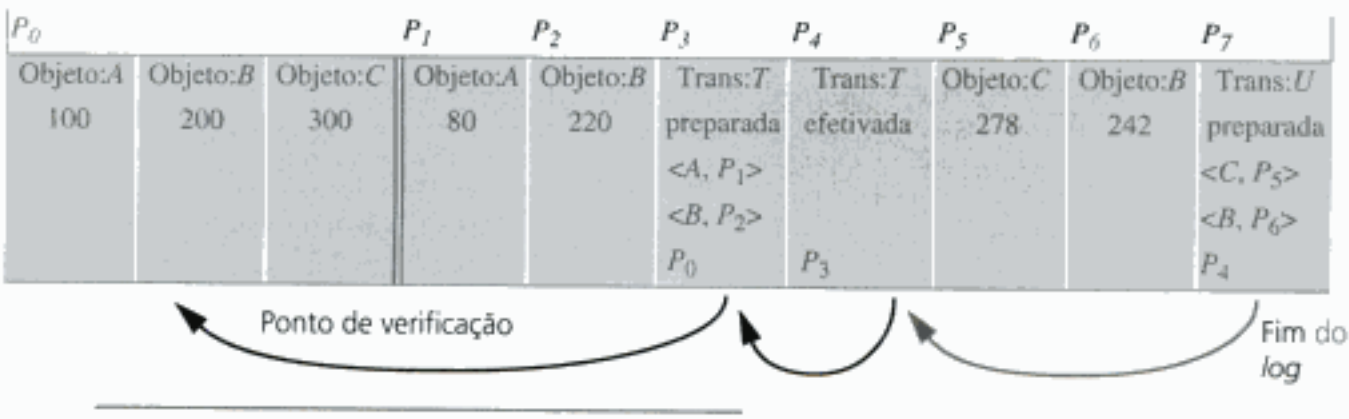


Figura 14.19 *Log* do serviço de transação bancária.

Hidden page

não foram totalmente resolvidas (incluindo informações relacionadas ao protocolo de efetivação de duas fases). O termo *ponto de verificação* (*checkpoint*) é usado para se referir às informações armazenadas pelo processo de estabelecer um ponto de verificação. O objetivo de estabelecer pontos de verificação é reduzir o número de transações para se tratar durante a recuperação e para reaver espaço de arquivo.

O estabelecimento de pontos de verificação pode ser feito imediatamente após a recuperação, mas antes que quaisquer novas transações sejam iniciadas. Entretanto, a recuperação pode não ocorrer muito freqüentemente. Portanto, o estabelecimento de pontos de verificação talvez precise ser feito de tempos em tempos, durante a atividade normal de um servidor. O ponto de verificação é gravado em um arquivo de recuperação futura e o arquivo de recuperação corrente permanece em uso até que o ponto de verificação esteja concluído. O estabelecimento de pontos de verificação consiste em "adicionar uma marca" no arquivo de recuperação, quando o processo começa, gravando os objetos do servidor no arquivo de recuperação futura e copiando (1) as entradas antes da marca relacionadas às transações ainda não resolvidas e (2) todas as entradas após a marca do arquivo de recuperação, no arquivo de recuperação futura. Quando o ponto de verificação está concluído, o arquivo de recuperação futura se torna o arquivo de recuperação.

O sistema de recuperação pode reduzir o uso de espaço, descartando o arquivo de recuperação antigo. Quando o gerenciador de recuperação está executando o processo de recuperação, ele pode encontrar um ponto de verificação no arquivo de recuperação. Quando isso acontece, ele pode restaurar imediatamente todos os objetos pendentes a partir do ponto de verificação.

14.6.2 Versões de sombra

A técnica de *log* grava entradas de status de transação, listas de intenções e objetos, tudo no mesmo arquivo – o *log*. A técnica das *versões de sombra* é uma maneira alternativa de organizar um arquivo de recuperação. Ela usa um *mapa* para localizar versões dos objetos do servidor em um arquivo chamado *repositório de versões*. O mapa associa os identificadores dos objetos do servidor às posições de suas versões correntes no repositório de versões. As versões gravadas por cada transação são *sombras* das versões efetivadas anteriores. As entradas de status de transação e as listas de intenções são tratadas separadamente. As versões de sombra serão descritas primeiro.

Quando uma transação está preparada para ser efetivada, todos os objetos alterados pela transação são anexados ao repositório de versões, deixando as versões efetivadas correspondentes intactas. Essas novas (até agora) versões de tentativa são chamadas de versões de sombra. Quando uma transação é efetivada, é feito um novo mapa, copiando o mapa antigo e inserindo as posições das versões de sombra. Para concluir o processo de efetivação, o novo mapa substitui o antigo.

Para restaurar os objetos quando um servidor é substituído após uma falha, seu gerenciador de recuperação lê o mapa e usa as informações constantes nele para localizar os objetos no repositório de versões.

Essa técnica será ilustrada com o mesmo exemplo envolvendo as transações T e U na Figura 14.20. A primeira coluna da tabela mostra o mapa antes das transações T e U, quando os saldos das

	Mapa no início			*Mapa quando T é efetivada*			
	$A \rightarrow P_0$			$A \rightarrow P_1$			
	$B \rightarrow P_0'$			$B \rightarrow P_2$			
	$C \rightarrow P_0''$			$C \rightarrow P_0''$			
	P_0	P_0'	P_0''	P_1	P_2	P_3	P_4
Repositório de versões	100	200	300	80	220	278	242
	Ponto de verificação						

Figura 14.20 Versões de sombra.

Hidden page

- No controle de concorrência com ordenação da indicação de tempo, às vezes um servidor sabe que uma transação poderá ser efetivada e envia uma confirmação para o cliente – nesse momento, os objetos são gravados no arquivo de recuperação (veja o Capítulo 13) para garantir sua permanência. Entretanto, a transação talvez tenha que esperar para ser efetivada, até que as transações anteriores tenham sido efetivadas. Em tais situações, as entradas de status de transação correspondentes no arquivo de recuperação estarão *esperando para ser* efetivadas e, então, passam a ser *efetivadas* para garantir a ordenação da indicação de tempo de transações efetivadas no arquivo de recuperação. Na recuperação, as transações que estão esperando para ser efetivadas podem receber permissão para isso, pois aquelas que estavam esperando acabaram de ser efetivadas ou, caso contrário, foram canceladas devido à falha do servidor.

14.6.4 Recuperação do protocolo de efetivação de duas fases

Em uma transação distribuída, cada servidor mantém seu próprio arquivo de recuperação. O gerenciamento de recuperação descrito na seção anterior deve ser ampliado para tratar com as transações que estão executando o protocolo de efetivação de duas fases no momento em que um servidor falha. Os gerenciadores de recuperação usam dois novos valores de status: *pronto*, *incerto*. Esses valores de status aparecem na Figura 14.6. Um coordenador usa *efetivada* para indicar que o resultado do voto é *Sim* e *pronto* para indicar que o protocolo de efetivação de duas fases está concluído. Um participante usa *incerto* para indicar que votou em *Sim*, mas ainda não conhece o resultado. Dois tipos adicionais de entrada permitem que um coordenador registre uma lista de participantes e que um participante registre seu coordenador:

Tipo de entrada	*Descrição do conteúdo da entrada*
Coordenador	Identificador de transação, lista de participantes
Participante	Identificador de transação, coordenador

Na fase 1 do protocolo, quando o coordenador está preparado para efetivar (e já adicionou uma entrada de status preparada em seu arquivo de recuperação), seu gerenciador de recuperação adiciona uma entrada *coordenador* em seu arquivo de recuperação. Antes que um participante possa votar em *Sim*, ele já deve ter se preparado para efetivar (e já deve ter adicionado uma entrada de status *preparada* em seu arquivo de recuperação). Quando ele vota em *Sim*, seu gerenciador de recuperação registra uma entrada *participante* e adiciona um status de transação *incerto* em seu arquivo de recuperação. Quando um participante vota em *Não*, ele adiciona um status de transação *cancelar* em seu arquivo de recuperação.

Na fase 2 do protocolo, o gerenciador de recuperação de um coordenador adiciona um status de transação *efetivada*, ou *cancelada*, em seu arquivo de recuperação, de acordo com a decisão. Essa deve ser uma gravação forçada, isto é, garantir que é feita neste momento. Os gerenciadores de recuperação dos participantes adicionam um status de transação *efetivar*, ou *cancelar*, em seus arquivos de recuperação, de acordo com a mensagem recebida do coordenador. Quando um coordenador tiver recebido uma confirmação de todos os seus participantes, seu gerenciador de recuperação adicionará um status de transação *pronto* em seu arquivo de recuperação – essa escrita não precisa ser forçada. A entrada de status *pronto* não faz parte do protocolo, mas é usada quando o arquivo de recuperação é reorganizado. A Figura 14.21 mostra as entradas em um *log* da transação T, na qual o servidor desempenhou o papel de coordenador, e da transação U, na qual o servidor desempenhou o papel de participante. Para as duas transações, a entrada de status de transação *preparada* vem primeiro. No caso de um coordenador, ela é seguida de uma entrada de coordenador e de uma entrada de status de transação *efetivada*. A entrada de status de transação *pronto* não é mostrada na Figura 14.21. No caso de um participante, a entrada de status de transação *preparada* é seguida de uma entrada de participante cujo estado é *incerto* e depois de uma entrada de status de transação *efetivada* ou *cancelada*.

Quando a servidor é substituído após uma falha, o gerenciador de recuperação precisa tratar com o protocolo de efetivação de duas fases, além de restaurar os objetos. Para qualquer transação onde o

Hidden page

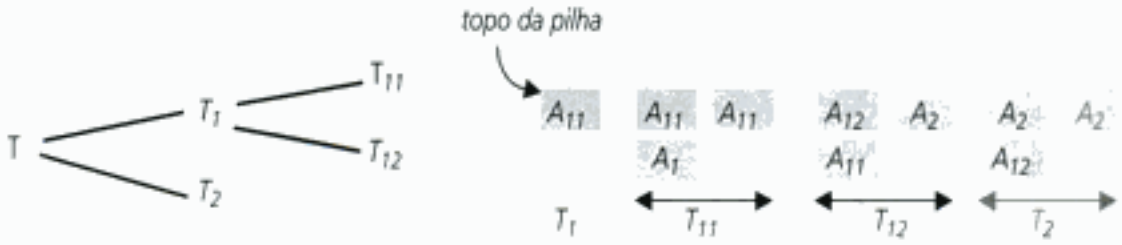


Figura 14.23 Transações aninhadas.

Recuperação de transações aninhadas ◊ No caso mais simples, cada subtransação de uma transação aninhada acessa um conjunto de objetos diferente. Quando cada participante se prepara para efetivar, durante o protocolo de efetivação de duas fases, ele grava seus objetos e suas listas de intenções no arquivo de recuperação local, associando-os ao identificador da transação de nível superior. Embora as transações aninhadas utilizem uma variante especial do protocolo de efetivação de duas fases, o gerenciador de recuperação usa os mesmos valores de status das transações planas.

Entretanto, a recuperação do cancelamento é complicada pelo fato de que várias subtransações nos mesmos níveis, e em níveis diferentes na hierarquia do aninhamento, podem acessar o mesmo objeto. A Seção 13.4 descreveu um esquema de bloqueio no qual as transações ascendentes herdam travas e as subtransações adquirem travas de suas ascendentes. O esquema de bloqueio obriga as transações ascendentes e as subtransações a acessarem objetos de dados comuns em diferentes momentos e garante que os acessos feitos por subtransações concorrentes aos mesmos objetos sejam organizados em série.

Os objetos acessados de acordo com as regras das transações aninhadas tornam-se recuperáveis por meio do fornecimento de versões de tentativa para cada subtransação. O relacionamento entre as versões de tentativa de um objeto usado pelas subtransações de uma transação aninhada é semelhante ao relacionamento entre as travas. Para suportar a recuperação dos cancelamentos, o servidor de um objeto compartilhado por transações em vários níveis fornece uma pilha de versões de tentativa – uma para cada transação aninhada a ser usada.

Quando a primeira subtransação de um conjunto de transações aninhadas acessa um objeto, ela recebe uma versão de tentativa que é uma cópia da versão efetivada corrente do objeto. Esse é considerado o topo da pilha, mas, a não ser que outras subtransações acessem o mesmo objeto, a pilha não se materializará.

Quando uma de suas subtransações acessa o mesmo objeto, ela copia a versão do topo da pilha e a coloca de volta na pilha. Todas as atualizações dessa subtransação são aplicadas na versão de tentativa do topo da pilha. Quando uma subtransação é efetivada provisoriamente, sua ascendente herda a nova versão. Para conseguir isso, a versão da subtransação, e a versão de sua ascendente, são descartadas da pilha e depois a nova versão da subtransação é colocada de volta na pilha (efetivamente substituindo a versão de sua ascendente). Quando uma subtransação é cancelada, sua versão que está no topo da pilha é descartada. Finalmente, quando a transação de nível superior é efetivada, a versão que está no topo da pilha (se houver) se torna a nova versão efetivada.

Por exemplo, na Figura 14.23, suponha que as transações T_1, T_{11}, T_{12} e T_2 acessem, o mesmo objeto, A, na ordem T_1; T_{11}; T_{12}; T_2. Suponha que suas versões de tentativa se chamem A_1, A_{11}, A_{12} e A_2. Quando T_1 começa a executar, A_1 é baseado na versão efetivada de A e colocado na pilha. Quando T_{11} começa a executar, ela baseia sua versão A_{11} em A_1 e o coloca na pilha; quando termina, substitui pela versão de seu ascendente na pilha. As transações T_{12} e T_2 agem de maneira semelhante, deixando finalmente o resultado de T_2 no topo da pilha.

14.7 Resumo

No caso mais geral, a transação de um cliente solicitará operações sobre objetos em vários servidores diferentes. Uma transação distribuída é qualquer transação cuja atividade envolve vários servidores di-

Hidden page

Hidden page

14.14 As transações *T*, *U* e *V* do Exercício 14.12 usam bloqueio de duas fases restrito e seus pedidos são interpostos como segue:

T	*U*	*V*
x = Read(i);		
		Write(k, 77);
	Write(i, 55)	
Write(j, 44)		
		Write(k,88)
	Write(j, 66)	

Supondo que o gerenciador de recuperação, em vez de esperar até o final da transação, anexe imediatamente a entrada de dados correspondente a cada operação *Write* no arquivo de *log*, descreva as informações gravadas no arquivo de *log* em nome das transações *T*, *U* e *V*. A gravação antecipada afeta a correção do procedimento de recuperação? Quais são as vantagens e desvantagens da gravação antecipada? *páginas 509–511*

14.15 As transações *T* e *U* são executadas com controle de concorrência com ordenação por indicação de tempo. Descreva as informações gravadas no arquivo de *log* em nome de *T* e *U*, admitindo o fato de que *U* tem uma indicação de tempo posterior a *T* e que deve esperar ser efetivada após *T*. Por que é fundamental que as entradas de efetivação no arquivo de *log* sejam ordenadas pelas indicações de tempo? Descreva o efeito da recuperação caso o servidor falhe (i) entre as duas operações *Commit* e (ii) após as duas operações.

T	*U*
X= Read(i);	
	Write(i, 55);
	Write(j, 66);
Write(j, 44);	
	Commit
Commit	

Quais são as vantagens e desvantagens da gravação antecipada com ordenação por indicação de tempo? *páginas 513–514*

14.16 As transações *T* e *U* do Exercício 14.15 são executadas com controle de concorrência otimista usando validação para trás e reiniciando as transações que falham. Descreva as informações gravadas no arquivo de *log* em nome delas. Por que é fundamental que as entradas de efetivação no arquivo de *log* sejam ordenadas pelos números de transação? Como os conjuntos de gravação das transações efetivadas são representados no arquivo de *log*? *páginas 509–511*

14.17 Suponha que o coordenador de uma transação falhe após ter gravado a entrada da lista de intenções, mas antes de ter gravado a lista de participantes ou enviado os pedidos de *canCommit?*. Descreva como os participantes resolvem a situação. O que o coordenador fará quando recuperar? Seria melhor gravar a lista de participantes antes da entrada da lista de intenções? *página 514–515*

Hidden page

15.1 Introdução

Neste capítulo, estudaremos a replicação de dados: a manutenção de cópias dos dados em vários computadores. A replicação é o segredo da eficácia dos sistemas distribuídos, pois ela pode fornecer um melhor desempenho, alta disponibilidade e tolerância a falhas. A replicação é amplamente usada. Por exemplo, o armazenamento de recursos de servidores web na cache dos navegadores e em servidores *proxies* web é uma forma de replicação, pois os dados mantidos nas caches e nos *proxies* são réplicas uns dos outros. O serviço de atribuição de nomes DNS, descrito no Capítulo 9, mantém cópias de mapeamentos entre nomes e atributos dos computadores e conta-se com ele para o acesso diário aos serviços pela Internet.

A replicação é uma técnica para melhorar *serviços*. As motivações para a replicação são: melhorar o desempenho de um serviço, aumentar sua disponibilidade ou torná-lo tolerante a falhas.

Melhoria de desempenho: a colocação dos dados na cache em clientes e servidores é uma maneira conhecida de melhorar o desempenho. Por exemplo, o Capítulo 2 mencionou que navegadores e servidores *proxies* colocam na cache cópias de recursos web para evitar a latência da busca desses recursos no servidor de origem. Além disso, às vezes os dados são replicados de forma transparente entre vários servidores de origem no mesmo domínio. A carga de trabalho é compartilhada entre esses servidores por meio da vinculação de seus endereços IP ao nome DNS do site, digamos *www.aWebSite.org*. Uma pesquisa de DNS de *www.aWebSite.org* resulta no retorno de um dos vários endereços IP de servidores, em um sistema de rodízio (veja a Seção 9.2.3). A replicação de dados imutáveis é simples: ela aumenta o desempenho com pouco custo para o sistema. A replicação de dados mutáveis (dinâmicos), como os da web, acarreta sobrecargas nos protocolos para garantir que os clientes recebam dados atualizados (veja a Seção 2.2.5). Assim, existem limites para a eficácia da replicação como uma técnica de melhoria de desempenho.

Maior disponibilidade: os usuários exigem que os serviços sejam de alta disponibilidade, isto é, a proporção do tempo durante a qual o serviço está acessível com tempo de resposta razoável deve ser próxima a 100%. Fora os atrasos decorrentes dos conflitos do controle de concorrência pessimista (bloqueio de dados), os fatores relevantes para a alta disponibilidade são:

- falhas do servidor;
- particionamento da rede e operação desconectada: as desconexões da comunicação, que freqüentemente não são planejadas, são um efeito colateral da mobilidade do usuário.

Para considerarmos o primeiro deles, a replicação é uma técnica para manter automaticamente a disponibilidade dos dados, a despeito das falhas do servidor. Se os dados são replicados em dois ou mais servidores que falham independentemente, então o software cliente pode acessar os dados em um servidor alternativo, caso o servidor padrão falhe ou se torne inacessível. Dessa forma, a porcentagem do tempo durante a qual o *serviço* está disponível pode ser melhorada pela replicação dos dados do servidor. Se cada um de n servidores tiver uma probabilidade independente p de falhar, ou se tornar inacessível, então a disponibilidade de um objeto armazenado em cada um desses servidores é de:

$$1 - \textit{probabilidade (todos os gerenciadores falharem ou se tornarem inacessíveis)} = 1 - p^n$$

Por exemplo, se existe uma probabilidade de 5% de falha de qualquer servidor individual, em determinado período de tempo, e se existem dois servidores, então a disponibilidade é de $1 - 0{,}05^2 = 1 - 0{,}0025 = 99{,}75\%$. Uma diferença importante entre os sistemas que usam cache e a replicação do servidor é que as caches não contêm necessariamente conjuntos de objetos, como arquivos, em sua totalidade. Portanto, o uso de cache não melhora necessariamente a disponibilidade no nível da aplicação – um usuário pode ter um arquivo necessário, mas não outro.

O particionamento da rede (veja a Seção 12.1) e a operação desconectada são o segundo fator que vai contra a alta disponibilidade. Os usuários móveis desconectam deliberadamente seus computadores ou, ao se moverem, podem ser desconectados involuntariamente de uma rede sem fio. Por exemplo, um usuário em um trem, portando um laptop, pode não ter acesso à rede (a interligação em uma rede sem fio pode ser interrompida ou pode não existir essa capacidade

disponível). Para poder trabalhar nessas circunstâncias – o assim chamado *trabalho desconectado* ou *operação desconectada* –, o usuário freqüentemente se preparará, copiando os dados muito utilizados, como o conteúdo de uma agenda compartilhada, de seu ambiente normal de trabalho para o laptop. Mas, freqüentemente, existe uma contrapartida com relação à disponibilidade durante tal período de desconexão: quando o usuário consulta ou atualiza a agenda, corre o risco de ler dados que foram alterados por outra pessoa nesse meio-tempo. Por exemplo, ele pode marcar um compromisso em uma hora já ocupada. O trabalho desconectado só será possível se o usuário (ou a aplicação, em nome do usuário) puder aceitar dados antigos e resolver posteriormente os conflitos que surgirem.

Tolerância a falhas: dados de alta disponibilidade não são necessariamente dados rigorosamente corretos. Eles podem estar desatualizados, por exemplo, ou dois usuários em lados opostos de uma rede que foi particionada podem fazer atualizações conflitantes e que precisem ser resolvidas. Em contraste, um serviço tolerante a falhas sempre garante comportamento rigorosamente correto, apesar de certo número e tipo de falhas. A correção está relacionada ao caráter atual dos dados fornecidos para o cliente e aos efeitos das operações do cliente sobre os dados. Às vezes, a correção também está relacionada ao tipo de respostas do serviço – como, por exemplo, no caso de um sistema de controle de tráfego aéreo, onde dados corretos são necessários em escalas de tempo curtas.

A mesma técnica básica usada para alta disponibilidade – de replicação de dados e funcionalidade entre computadores – também pode ser aplicada para se obter tolerância a falhas. Se até f de $f + 1$ servidores falham, então, em princípio, pelo menos um permanece para fornecer o serviço. E se até f servidores podem apresentar falhas bizantinas, então, em princípio, um grupo de $2f + 1$ servidores pode fornecer um serviço correto, fazendo-se com que os servidores corretos vençam por voto os servidores falhos (que podem fornecer valores espúrios). Mas a tolerância a falhas é mais sutil do que essa descrição simples faz parecer. O sistema precisa gerenciar a coordenação de seus componentes precisamente para manter as garantias de correção diante de falhas, as quais podem ocorrer a qualquer momento.

Um requisito comum quando dados são replicados é a *transparência da replicação*, isto é, normalmente, os clientes não devem saber que existem várias cópias *físicas* dos dados. No que diz respeito a eles, os dados são organizados como objetos *lógicos* individuais (ou objetos) e identificam apenas um item em cada caso, quando solicitam a execução de uma operação. Além disso, os clientes esperam que as operações retornem apenas um conjunto de valores, apesar das operações poderem ser executadas sobre mais de uma cópia física de comum acordo.

O outro requisito geral para dados replicados – que pode variar com os níveis de rigor exigidos pelos aplicativos – é a consistência. Isso diz respeito ao fato das operações executadas sobre um conjunto de objetos replicados produzirem resultados que satisfaçam a especificação da correção desses objetos.

Vimos, no exemplo da agenda, que durante a operação desconectada os dados podem se tornar inconsistentes, pelos menos temporariamente. Mas quando os clientes permanecem conectados, freqüentemente não é aceitável que diferentes clientes (usando diferentes cópias físicas dos dados) obtenham resultados inconsistentes ao fazerem pedidos que afetem os mesmos objetos lógicos. Isto é, isso não é aceitável se os resultados violarem os critérios de correção do aplicativo.

Examinaremos agora, com mais detalhes, os problemas de projeto que surgem quando replicamos dados para obter serviços de alta disponibilidade e tolerantes a falhas. Examinaremos também algumas soluções e técnicas padrão para tratar desses problemas. Primeiramente, as Seções 15.2 a 15.4 abordarão o caso onde os clientes fazem invocações individuais sobre dados compartilhados. A Seção 15.2 apresentará uma arquitetura geral para gerenciamento de dados replicados e apresentará a comunicação em grupo como uma ferramenta importante. A comunicação em grupo é particularmente útil para se obter tolerância a falhas, que será o assunto da Seção 15.3. A Seção 15.4 descreverá técnicas para se obter alta disponibilidade, incluindo a operação desconectada. Ele incluirá estudos de caso da arquitetura Gossip, Bayou e o sistema de arquivos Coda. A Seção 15.5 examinará como se faz para suportar transações em dados replicados. Conforme os Capítulos 13 e 14 explicaram, as transações são constituídas de seqüências de operações, em vez de operações simples.

15.2 Modelo de sistema e comunicação em grupo

Os dados de nosso sistema consistem em um conjunto de itens que chamamos de objetos. Um objeto poderia ser, digamos, um arquivo ou um objeto Java. Mas cada objeto lógico é implementado por um conjunto de cópias *físicas* chamadas de *réplicas*. As réplicas são objetos físicos, cada um armazenado em um único computador, com dados e comportamento ligados até certo grau de consistência pela operação do sistema. As 'réplicas' de determinado objeto não são necessariamente idênticas, pelo menos não em um ponto em particular no tempo. Algumas réplicas podem ter recebido atualizações que as outras não receberam.

Nesta seção, forneceremos um modelo de sistema geral para gerenciar réplicas e depois descreveremos os sistemas de comunicação em grupo, os quais são particularmente úteis para se obter tolerância a falhas por meio de replicação.

15.2.1 Modelo de sistema

Supomos um sistema assíncrono em que os processos só podem falhar por colapso. Nossa suposição padrão é a de que não pode ocorrer particionamento da rede, mas às vezes consideraremos o que acontece se elas ocorrem. O particionamento da rede torna mais difícil construir detectores de falha, os quais usamos para obter *multicast* confiável e totalmente ordenado.

Tendo em vista a generalidade, descreveremos os componentes da arquitetura por meio de suas funções, o que não implica que eles devam ser necessariamente implementados por processos (ou hardware) distintos. O modelo envolve réplicas mantidas por *gerenciadores de réplica* distintos (veja a Figura 15.1), que são componentes que contêm as réplicas em determinado computador e executam operações diretamente sobre elas. Esse modelo geral pode ser aplicado a um ambiente cliente-servidor, no caso em que um gerenciador de réplica é um servidor. Às vezes, os chamaremos simplesmente de servidores. Do mesmo modo, ele pode ser empregado para aplicativos e, nesse caso, os processos aplicativos podem atuar como clientes e gerenciadores de réplica. Por exemplo, o laptop do usuário em um trem pode conter um aplicativo que atue como gerenciador de réplica para sua agenda.

Sempre exigimos que um gerenciador de réplica aplique operações em suas réplicas de forma recuperável. Isso nos permite presumir que uma operação em um gerenciador de réplica não deixa resultados inconsistentes, caso venha a falhar no meio do processo. Às vezes, exigimos que cada gerenciador de réplica seja uma *máquina de estados* [Lamport 1978, Schneider 1990]. Tal gerenciador de réplica aplica operações em suas réplicas de forma atômica (indivisivelmente), de modo que sua execução é equivalente a efetuar operações em alguma seqüência restrita. Além disso, o estado de suas réplicas é uma função determinística de seus estados iniciais e da seqüência de operações nelas aplicadas. Outros estímulos, como a leitura em um relógio, ou de um sensor vinculado, não têm relação com esses valores de estado. Sem essa suposição, não poderiam ser dadas garantias de consistência entre os gerenciadores de réplica que aceitam operações de atualização independentemente. O sistema só pode

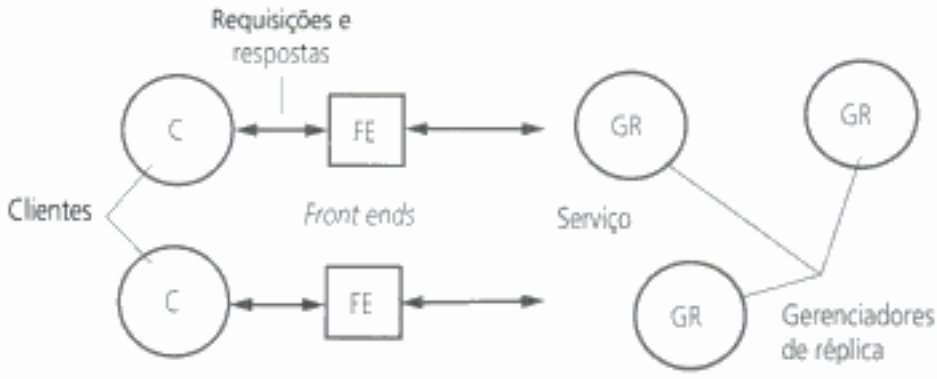


Figura 15.1 Um modelo de arquitetura básico para o gerenciamento de dados replicados.

Hidden page

Hidden page

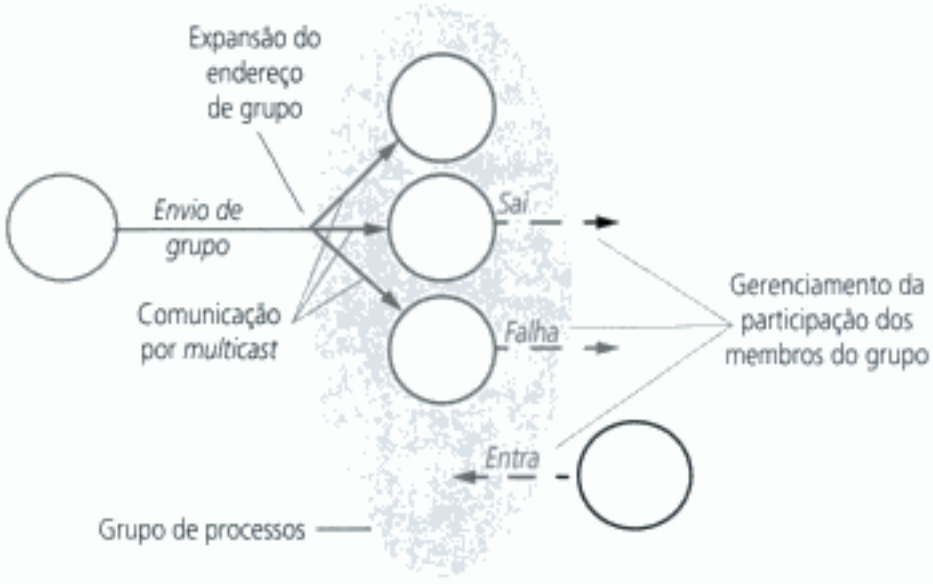


Figura 15.2 Serviços fornecidos para grupos de processos.

Notificar os membros sobre alterações na participação como membro do grupo: o serviço notifica os membros do grupo quando um processo é adicionado, ou quando um processo é excluído (por meio de falha ou quando o processo é deliberadamente retirado do grupo).

Realizar a expansão de endereço do grupo: quando um processo difunde uma mensagem *multicast*, ele fornece o identificador do grupo, em vez de listar os processos do grupo. O serviço de gerenciamento de participação dos membros expande o identificador de grupo em seus membros participantes para envio. O serviço pode coordenar o envio *multicast* mesmo com alterações na participação de membros, controlando a expansão do endereço. Isto é, ele pode decidir consistentemente para onde vai enviar determinada mensagem, mesmo que a participação como membro possa estar mudando durante o envio. Discutiremos a comunicação de *modo de visualização síncrono* a seguir.

Note que o *multicast* IP é um caso fraco de serviço de participação como membro do grupo, contendo algumas dessas propriedades, mas não todas. Ele permite que os processos entrem ou saiam dos grupos dinamicamente e realiza a expansão de endereço, de modo que os remetentes só precisam fornecer um único endereço *multicast* IP como destino para uma mensagem. Mas o *multicast* IP sozinho não fornece aos membros do grupo informações sobre a participação como membro corrente, e o envio *multicast* não é coordenado com alterações na participação dos membros.

Os sistemas que podem se adaptar à medida que os processos entram, saem e falham – sistemas tolerantes a falhas, em particular – exigem os recursos mais avançados de detecção e notificação de falhas nas alterações da participação como membro. Um serviço de participação como membro do grupo completo mantém *modos de visualização do grupo*, que são listas dos membros correntes do grupo, identificados pelos seus identificadores de processo exclusivos. A lista é ordenada, por exemplo, de acordo com a seqüência na qual os membros entraram no grupo. Um novo modo de visualização de grupo é gerado quando os processos são adicionados ou excluídos.

É importante entender que um serviço de participação como membro do grupo pode excluir um processo de um grupo porque ele é *Suspeito*, mesmo que possa não ter falhado. Uma falha de comunicação pode ter tornado o processo inacessível, enquanto ele continua a ser executado normalmente. Um serviço de participação como membro sempre está livre para excluir tal processo. O efeito da exclusão é que, daí em diante, nenhuma mensagem será enviada para esse processo. Além disso, no caso de um grupo fechado, se esse processo se conectar novamente, todas as mensagens que ele tentar enviar não serão enviadas para os membros do grupo. Esse processo terá que entrar novamente no grupo (como uma "reencarnação" de si mesmo, com um novo identificador) ou cancelar suas operações.

Uma falsa suspeita de um processo, e sua conseqüente exclusão do grupo, pode reduzir a eficácia do grupo. O grupo precisa sair-se bem sem a confiabilidade, ou o desempenho extra, que o processo retirado possa ter fornecido. O desafio de projeto, além de projetar detectores de falhas mais precisos possível, é garantir que um sistema baseado na comunicação em grupo não se comporte *incorretamente* se um processo for falsamente suspeito.

Uma consideração importante é como o serviço de gerenciamento de grupo trata o particionamento da rede. A desconexão, ou a falha de componentes, como um roteador em uma rede, podem dividir um grupo de processos em dois ou mais subgrupos, de modo que a comunicação entre os subgrupos seja impossível. Os serviços de gerenciamento de grupo diferem no fato de serem de *particionamento primário* ou *particionáveis*. No primeiro caso, o serviço de gerenciamento permite que no máximo um subgrupo (uma maioria) sobreviva a um particionamento; os processos restantes são informados de que devem suspender as operações. Esse arranjo é apropriado para os casos onde os processos gerenciam dados importantes e os custos das inconsistências entre dois ou mais subgrupos superam qualquer vantagem do trabalho desconectado.

Por outro lado, em algumas circunstâncias é aceitável que dois ou mais subgrupos continuem a operar, e um serviço de gerenciamento de grupo do tipo particionável permite isso. Por exemplo, em uma aplicação em que os usuários participam de uma conferência por áudio, ou vídeo, para discutir alguns problemas, pode ser aceitável que dois ou mais subgrupos de usuários continuem suas discussões independentemente, a despeito de uma divisão. Eles podem reunir seus resultados quando o particionamento for sanado e os subgrupos forem novamente conectados.

Envio de modo de visualização ◊ Considere a tarefa de um programador escrevendo uma aplicação que é executada, em cada processo, em um grupo que deve aceitar membros novos e perdidos. O programador precisa saber se o sistema trata cada membro de uma maneira consistente quando a participação como membro muda. Seria complicado se o programador tivesse que consultar o estado de todos os outros membros e chegar a uma decisão global quando ocorresse uma alteração da participação dos membros, em vez de poder tomar uma decisão local sobre como responder à alteração. A vida do programador se torna mais difícil, ou mais fácil, de acordo com as garantias que se aplicam quando o sistema envia modos de visualização para os membros do grupo.

Para cada grupo g, o serviço de gerenciamento de grupo envia para qualquer processo membro $p \in g$, uma série de modos de visualização $v_0(g)$, $v_1(g)$, $v_2(g)$, etc. Por exemplo, uma série de modos de visualização poderia ser $v_0(g) = (p)$, $v_1(g) = (p, p')$ e $v_2(g) = (p, p') - p$ entra em um grupo vazio, depois p' entra no grupo, em seguida p' sai dele. Embora várias alterações da participação de membros possam ocorrer paralelamente, como quando um processo entra no grupo exatamente quando outro sai, o sistema impõe uma ordem na seqüência de modos de visualização fornecida para cada processo.

Falamos sobre um membro *enviando um modo de visualização* quando ocorre uma alteração na participação de membros e o aplicativo é notificado da nova participação – exatamente como falamos sobre um processo enviando uma mensagem *multicast*. Assim como no envio por *multicast*, o envio de um modo de visualização é diferente do recebimento de um modo de visualização. Os protocolos de participação como membro de grupo mantém os modos de visualização propostos em uma fila de retenção até que todos os membros existentes possam concordar com seu envio.

Também falamos sobre um evento ocorrendo *em um modo de visualização* $v(g)$ no processo p, se no momento da ocorrência do evento, p tiver enviado $v(g)$, mas ainda não tiver enviado o próximo modo de visualização $v'(g)$.

Alguns requisitos básicos do envio de modos de visualização são os seguintes:

Ordem: se um processo p envia o modo de visualização $v(g)$ e depois o modo de visualização $v'(g)$, então nenhum outro processo $q \neq p$ envia $v'(g)$ antes de $v(g)$.

Integridade: se o processo p envia o modo de visualização $v(g)$, então $p \in v(g)$.

Não trivialidade: se o processo q entra em um grupo e é, ou se torna acessível, a partir de um processo $p \neq q$, então q estará sempre nos modos de visualização enviados por p. Analogamente, se o grupo sofre um particionamento e assim permanece, então os modos de visualização enviados por uma partição da rede excluirão os processos de outra partição.

Hidden page

Hidden page

O sistema V [Cheriton e Zwaenepoel 1985] foi o primeiro a incluir suporte para grupos de processos. Após o ISIS, foram desenvolvidos grupos de processos com algum tipo de serviço de participação como membro do grupo em vários outros sistemas, incluindo o Horus [van Renesse *et al.* 1996], o Totem [Moser *et al.* 1996] e o Transis [Dolev e Malki 1996].

Variações do sincronismo de modo de visualização foram propostas para serviços de participação como membro de grupo divisível, incluindo suporte para aplicativos com reconhecimento de particionamentos [Babaoglu *et al.* 1998] e sincronismo virtual estendido [Moser *et al.* 1994].

Finalmente, Cristian [1991b] discute um serviço de participação de membro de grupo para sistemas distribuídos síncronos.

Grupos de objetos ◊ Os grupos de objetos fornecem uma estratégia orientada a objetos para computação de grupos. Um grupo de objetos é um conjunto de objetos (normalmente instâncias da mesma classe) que processa o mesmo conjunto de invocações concorrentemente e cada um retorna respostas. Os objetos clientes não precisam saber da replicação. Eles invocam operações sobre um único objeto local, o qual atua como um *proxy* para o grupo. O *proxy* usa um sistema de comunicação em grupo para enviar as invocações para os membros do grupo de objetos.

O Electra [Maffeis 1995] é um sistema compatível com o CORBA que suporta grupos de objetos. Um grupo do Electra pode ter uma interface com qualquer aplicativo compatível com o CORBA. O Electra foi originalmente construído em cima do sistema de comunicação em grupo Horus, o qual utiliza para gerenciar a participação como membro do grupo e para enviar invocações por *multicast*. No modo transparente, o *proxy* local retorna a primeira resposta disponível para um objeto cliente. No modo não transparente, o objeto cliente pode acessar todas as respostas retornadas pelos membros do grupo. O Electra usa uma extensão da interface ORB padrão do CORBA, com funções para criar e destruir grupos de objetos e gerenciar sua participação como membros.

O Eternal [Moser *et al.* 1998] e o Object Group Service [Guerraoui *et al.* 1998] também fornecem suporte compatível com o CORBA para grupos de objetos.

15.3 Serviços tolerantes a falhas

Nesta seção, examinaremos como se faz para fornecer um serviço correto a despeito de até f falhas de processo através da replicação de dados e da funcionalidade dos gerenciadores de réplica. Por simplicidade, supomos que a comunicação permanece confiável e que não ocorrem particionamentos na rede.

É suposto que cada gerenciador de réplica se comporta de acordo com uma especificação da semântica dos objetos que gerencia, quando não tiver falhado. Por exemplo, uma especificação de contas bancárias incluiria a garantia de que os fundos transferidos entre as contas nunca poderia desaparecer, e que apenas os depósitos e os saques afetariam o saldo de uma conta em particular.

Intuitivamente, um serviço baseado em replicação é correto se continua respondendo a despeito das falhas, e se os clientes não conseguem identificar a diferença entre o serviço que obtêm de uma implementação com dados replicados e de um fornecido por um único gerenciador de réplica correto. É preciso cuidado para satisfazer esse critério. Se não forem tomadas precauções, poderão surgir anomalias quando houver vários gerenciadores de réplica – mesmo lembrando que estamos considerando os efeitos de operações individuais e não de transações.

Considere um sistema de replicação simples, no qual dois gerenciadores de réplica nos computadores A e B mantêm, cada um, réplicas de duas contas bancárias x e y. Os clientes lêem e atualizam as contas em seus gerenciadores de réplica locais, mas tentam usar outro gerenciador de réplica, caso esse falhe. Os gerenciadores de réplica propagam as atualizações entre si, em segundo plano (*background*), após responderem ao cliente. As duas contas têm inicialmente um saldo de $ 0.

O cliente 1 atualiza para $1 o saldo de x em seu gerenciador de réplica local B, e depois tenta atualizar para $2 o saldo de y, mas descobre que B falhou. Portanto, em vez disso, o cliente 1 aplica a atualização em A. Agora, o cliente 2 lê os saldos em seu gerenciador de réplica local A e descobre primeiro que y tem $2 e depois que x tem $0 – a atualização de B na conta bancária x não chegou, pois

B falhou. A situação está mostrada a seguir, onde as operações são rotuladas de acordo com o computador em que elas ocorreram primeiro e as operações inferiores acontecem depois:

Cliente 1:	*Cliente* 2:
$setBalance_B(x, 1)$	
$SetBalance_A(y, 2)$	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

Essa execução não corresponde a uma especificação sensata para o comportamento de contas bancárias: o cliente 2 deve ter lido um saldo de $1 para *x*, dado que ele lê o saldo de $2 para *y*, e o saldo de *y* foi atualizado após o de *x*. O comportamento anômalo no caso replicado não poderia ter ocorrido se as contas bancárias tivessem sido implementadas por um único servidor. Podemos construir sistemas que gerenciam objetos replicados sem o comportamento anômalo produzido pelo protocolo simples de nosso exemplo. Primeiramente, precisamos entender o que é considerado um comportamento correto para um sistema replicado.

Capacidade de linearização e consistência seqüencial ◊ Existem vários critérios de correção para objetos replicados. Os sistemas mais rigorosamente corretos podem ser *linearizados* e essa propriedade é chamada de *capacidade de linearização*. Para entender a capacidade de linearização, considere uma implementação de serviço replicado com dois clientes. Seja $o_{i0}, o_{i1}, o_{i2}, \ldots$ a seqüência de operações de leitura e atualização executadas pelo cliente *i* em alguma execução. Cada operação o_{ij} nessas seqüências é especificada pelo tipo de operação, pelos argumentos e pelos valores de retorno, conforme eles ocorreram em tempo de execução. Supomos que toda operação é síncrona. Isto é, os clientes esperam que uma operação termine antes de solicitar a próxima.

Um único servidor gerenciando uma única cópia dos objetos serializaria as operações dos clientes. No caso de uma execução apenas com cliente 1 e cliente 2, essa interposição das operações poderia ser, digamos, $o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, o_{12}, \ldots$. Definimos nossos critérios de correção para objetos replicados fazendo referência a uma interposição *virtual* das operações dos clientes, as quais não ocorrem fisicamente em algum gerenciador de réplica em particular necessariamente, mas que estabelecem a correção da execução.

Diz-se que um serviço de objetos compartilhados replicado pode ser linearizado se, para *qualquer execução*, existe alguma interposição da série de operações executadas por todos os clientes que satisfaz os dois critérios a seguir:

- A seqüência interposta de operações satisfaz a especificação de uma (única) cópia correta dos objetos.
- A ordem das operações na interposição é consistente com os tempos reais em que as operações ocorreram na execução real.

Essa definição capta a idéia de que para qualquer conjunto de operações de cliente existe uma execução canônica virtual – as operações interpostas a que a definição se refere – em preparação a uma imagem virtual única dos objetos compartilhados. E cada cliente enxerga um modo de visualização dos objetos compartilhados consistente com essa imagem única: isto é, os resultados das operações do cliente fazem sentido quando ocorrem dentro da interposição.

O serviço que deu origem à execução dos clientes de conta bancária do exemplo, não pode ser linearizado. Mesmo ignorando o tempo real em que as operações ocorreram, não há nenhuma interposição das operações dos dois clientes que satisfaça qualquer especificação de conta bancária correta: para propósitos de auditoria, se a atualização de uma conta ocorresse após a outra, então a primeira deveria ser observada, caso a segunda também tivesse sido observada.

Note que a capacidade de linearização diz respeito à interposição de operações individuais e não se destina a ser transacional. Uma execução que pode ser linearizada pode violar as noções de consistência específicas da aplicação, caso não seja aplicado controle de concorrência.

O requisito de tempo real na capacidade de linearização é desejável em um mundo ideal, pois ele captura nossa noção de que os clientes devem receber informações atualizadas. Mas, igualmente, a presença do tempo real na definição levanta o problema da exeqüibilidade da capacidade de linearização, pois nem sempre podemos sincronizar os relógios com o grau de precisão exigido. Uma condição de correção menos rigorosa é a *consistência seqüencial*, que captura um requisito fundamental relacionado à ordem em que as requisições são processadas, sem apelar para o tempo real. A definição mantém o primeiro critério da definição de capacidade de linearização, mas modifica o segundo, como segue:

Diz-se que um serviço de objetos compartilhados replicado tem consistência seqüencial se, para qualquer execução, existe alguma interposição da série de operações executadas por todos os clientes que satisfaça os dois critérios a seguir:

- A seqüência interposta de operações satisfaz a especificação de uma (única) cópia correta dos objetos.
- A ordem das operações na interposição é consistente com a ordem do programa no qual cada cliente individual as executou.

Note que o tempo absoluto não aparece nessa definição. Nem nenhuma outra ordem *total* em todas as operações. A única noção de ordem relevante é a ordem dos eventos em cada cliente separado – a ordem do programa. A interposição de operações pode embaralhar, em qualquer ordem, a seqüência de operações de um conjunto de clientes, desde que a ordem de cada cliente não seja violada, e que o resultado de cada operação seja consistente, em termos da especificação dos objetos, com as operações que a precederam. Isso é semelhante a embaralhar vários maços de cartas para que elas se misturem de maneira tal que preservem a ordem original de cada maço.

Todo serviço que pode ser linearizado também tem consistência seqüencial, pois a ordem do tempo real reflete a ordem do programa de cada cliente. O inverso não é verdadeiro. Um exemplo de execução de um serviço que tem consistência seqüencial, mas não pode ser linearizado, aparece a seguir:

Cliente 1:	*Cliente 2:*
$setBalance_B(x, 1)$	
	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$SetBalance_A(y, 2)$	

Essa execução é possível sob uma estratégia de replicação simples, mesmo que nenhum dos computadores A ou B falhe, mas se a atualização de x feita pelo cliente 1 em B não tiver chegado a A quando o cliente 2 a ler. O critério do tempo real para a capacidade de linearização não é satisfeita, pois $getBalance_A(x) \rightarrow 0$ ocorre depois de $setBalance_B(x, 1)$; mas a interposição a seguir satisfaz os dois critérios da consistência seqüencial: $getBalance_A(y) \rightarrow 0$, $getBalance_A(x) \rightarrow 0$, $setBalance_B(x, 1)$, $setBalance_A(y, 2)$.

Lamport concebeu a consistência seqüencial [1979] e a capacidade de linearização [1986] em relação aos registros de memória compartilhada (embora tenha usado o termo atomicidade, em vez de capacidade de linearização). Herlihy e Wing [1990] generalizaram a idéia para incluir objetos compartilhados arbitrários. O Capítulo 18, que examina a memória compartilhada distribuída, definirá e discutirá algumas propriedades menos rigorosas da consistência.

15.3.1 Replicação passiva (*0*)

No modelo *passivo*, ou de *backup primário*, de replicação para tolerância a falhas (Figura 15.4), existe em determinado momento um único gerenciador de réplica primário e um ou mais gerenciadores de réplica secundários – *backups* ou escravos. Na forma pura do modelo, os *front ends* se comunicam

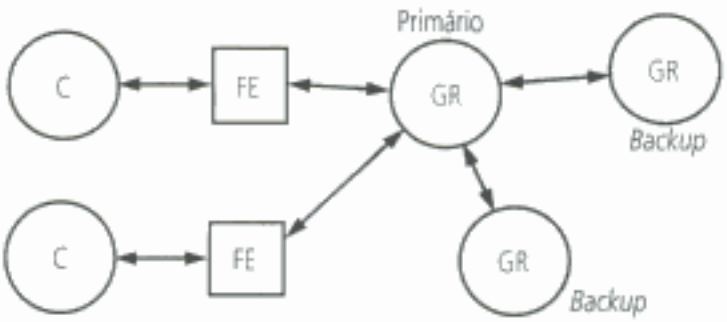


Figura 15.4 O modelo passivo (*backup* primário) de tolerância a falhas.

somente com o gerenciador de réplica primário para obterem o serviço. O gerenciador de réplica primário executa as operações e envia cópias dos dados atualizados para os *backups*. Se o primário falhar, um dos *backups* será promovido para atuar como primário.

A seqüência de eventos, quando um cliente solicita que uma operação seja executada, é a seguinte:

1. *Requisição*: o *front end* emite a requisição, contendo um identificador exclusivo, para o gerenciador de réplica primário.

2. *Coordenação*: o gerenciador primário pega cada requisição atomicamente, na ordem em que a recebe. Ele verifica o identificador exclusivo, para o caso de já a ter executado e, se assim for, simplesmente envia a resposta novamente.

3. *Execução*: o gerenciador primário executa a requisição e armazena a resposta.

4. *Acordo*: se a requisição é uma atualização, então o gerenciador primário envia o estado atualizado, a resposta e o identificador exclusivo para todos os gerenciadores de *backup*. Os gerenciadores de *backup* enviam uma confirmação.

5. *Resposta*: o gerenciador primário responde para o *front end*, o qual envia a resposta de volta para o cliente.

Obviamente, esse sistema implementa a capacidade de linearização se o gerenciador primário estiver correto, pois este coloca em seqüência todas as operações sobre os objetos compartilhados. Se o gerenciador primário falhar, o sistema manterá a capacidade de linearização caso um único gerenciador de *backup* se torne o novo gerenciador primário e a nova configuração do sistema partir exatamente de onde a última estava:

- o gerenciador primário é substituído por um único gerenciador de *backup* (se dois clientes começassem a usar dois gerenciadores de *backup*, o sistema funcionaria incorretamente); e
- os gerenciadores de réplica sobreviventes concordam a respeito das operações que tinham sido executadas no ponto onde o gerenciador primário substituto assume o controle.

Esses dois requisitos são satisfeitos se os gerenciadores de réplica (primário e *backups*) estão organizados como um grupo, e se o gerenciador primário usa comunicação em grupo com modo de visualização síncrono para enviar as atualizações para os gerenciadores de *backup*. Então, o primeiro dos dois requisitos anteriores é facilmente satisfeito. Quando o gerenciador primário falha, o sistema de comunicação envia um novo modo de visualização para os gerenciadores de *backup* sobreviventes, excluindo o gerenciador primário antigo. O gerenciador de *backup* que substitui o gerenciador primário pode ser escolhido por qualquer função desse modo de visualização. Por exemplo, os gerenciadores de *backup* podem escolher o primeiro membro desse modo de visualização como substituto. Esse gerenciador de *backup* pode se registrar como gerenciador primário em um serviço de nomes que os clientes consultam quando suspeitam de que o gerenciador primário falhou (ou assim que exigem o serviço).

O segundo requisito também é satisfeito pela propriedade de ordenação do sincronismo de modo de visualização e pelo uso de identificadores armazenados para detectar requisições repetidas. A semântica do modo de visualização síncrono garante que todos os gerenciadores de *backup*, ou nenhum

deles, enviarão qualquer atualização dada, antes de enviar o novo modo de visualização. Assim, o novo gerenciador primário e os gerenciadores de *backup* sobreviventes concordam sobre se a atualização de qualquer cliente em particular foi processada ou não.

Considere um *front end* que não recebeu uma resposta. O *front end* retransmite a requisição para o gerenciador de *backup* que tiver assumido como gerenciador primário. O gerenciador primário pode ter falhado em qualquer ponto, durante a operação. Se ele falhou antes do estágio de acordo (4), os gerenciadores de réplica sobreviventes não podem ter processado a requisição. Se ele falhou durante o estágio de acordo, eles podem ter processado a requisição. Se ele falhou depois desse estágio, então eles definitivamente o processaram. Mas o novo gerenciador primário não precisa saber em que estágio o gerenciador primário antigo estava quando falhou. Ao receber uma requisição, ele prossegue a partir do estágio 2 anterior. Pelo sincronismo do modo de visualização, nenhuma consulta com os gerenciadores de *backup* é necessária, pois todos processaram o mesmo conjunto de mensagens.

Discussão sobre a replicação passiva ◊ O modelo *backup* primário pode ser usado mesmo onde o gerenciador de réplica primário se comporta de maneira não determinística devido a ser, por exemplo, *multi-threaded*. Como o gerenciador primário comunica o estado atualizado das operações, em vez de uma especificação das operações em si, os gerenciadores de *backup* registram cegamente o estado determinado apenas pelas ações do gerenciador primário.

Para sobreviver a até f falhas de processo, um sistema de replicação passiva exige $f + 1$ gerenciadores de réplica (tal sistema não pode tolerar falhas bizantinas). O *front end* exige pouca funcionalidade para obter tolerância a falhas. Ele precisará pesquisar o novo gerenciador primário, quando o gerenciador primário corrente não responder.

A replicação passiva tem a desvantagem de acarretar sobrecargas relativamente grandes. A comunicação de modo de visualização síncrono exige várias rodadas de comunicação por *multicast* e, se o gerenciador primário falhar, ainda mais latência será acarretada, enquanto o sistema de comunicação em grupo concorda sobre o novo modo de visualização e o distribui.

Em uma variação do modelo apresentado aqui, os clientes podem enviar requisições de leitura para os gerenciadores de *backup*, diminuindo assim o trabalho do gerenciador primário. Com isso, a garantia da capacidade de linearização é perdida, mas os clientes recebem um serviço com consistência seqüencial.

A replicação passiva é usada no sistema de arquivos replicado Harp [Liskov *et al.* 1991]. O *Network Information Service* da Sun (NIS, anteriormente *Yellow Pages*) usa replicação passiva para obter alta disponibilidade e bom desempenho, embora com menos garantias do que a consistência seqüencial. As garantias de consistência mais fracas ainda são satisfatórias para muitos propósitos, como o armazenamento de certos tipos de registros de administração de sistema. Os dados replicados são atualizados em um servidor mestre e propagados de lá para servidores escravos, usando comunicação de um para um (em vez de grupo). Os clientes podem se comunicar com um servidor mestre, ou com um escravo, para recuperar informações. Entretanto, no NIS, os clientes não podem solicitar atualizações: elas são feitas nos arquivos do mestre.

15.3.2 Replicação ativa

No modelo *ativo* de replicação para tolerância a falhas (veja a Figura 15.5), os gerenciadores de réplica são máquinas de estado que desempenham papéis equivalentes e são organizados como um grupo. Os *front ends* enviam suas requisições por *multicast* para o grupo de gerenciadores de réplica, e todos os gerenciadores de réplica processam a requisição independentemente, mas de forma idêntica, e respondem. Se qualquer gerenciador de réplica falhar, isso não tem nenhum impacto sobre o desempenho do serviço, pois os gerenciadores de réplica restantes continuam a responder normalmente. Veremos que a replicação ativa pode tolerar falhas bizantinas, pois o *front end* pode reunir e comparar as respostas que recebe.

Sob a replicação ativa, a seqüência de eventos quando um cliente solicita a execução de uma operação é a seguinte:

1. *Requisição*: o *front end* anexa um identificador exclusivo à requisição e a difunde via *multicast* para o grupo de gerenciadores de réplica, usando uma primitiva de *multicast* totalmente ordenada

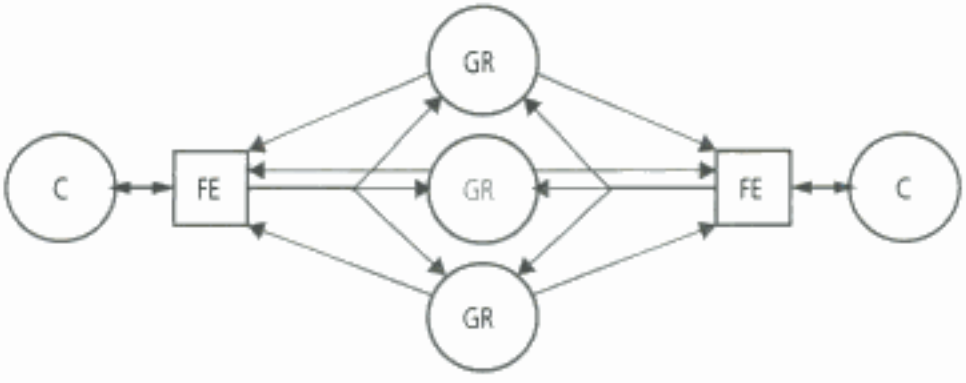


Figura 15.5 Replicação ativa.

e confiável. Supõe-se que o *front end* pode falhar por colapso, na pior das hipóteses. Ele não emite a próxima requisição até que tenha recebido uma resposta.

2. *Coordenação*: o sistema de comunicação em grupo envia a requisição para cada gerenciador de réplica correto na mesma ordem (total).

3. *Execução*: todos os gerenciadores de réplica executam a requisição. Como eles são máquinas de estado, e como as requisições são distribuídas na mesma ordem total, todos os gerenciadores de réplica corretos processam a requisição de forma idêntica. A resposta contém o identificador de requisição exclusiva do cliente.

4. *Acordo*: nenhuma fase de acordo é necessária, devido à semântica da distribuição por *multicast*.

5. *Resposta*: cada gerenciador de réplica envia sua resposta para o *front end*. O número de respostas reunidas pelo *front end* depende das suposições de falha e do algoritmo de *multicast*. Se, por exemplo, o objetivo for tolerar apenas falhas por colapso e o *multicast* satisfizer o acordo uniforme e as propriedades de ordenação, o *front end* passará ao cliente a primeira resposta que chegar e descartará as restantes (ele pode distingui-las das respostas de outras requisições examinando o identificador presente na resposta).

Esse sistema obtém consistência seqüencial. Todos os gerenciadores de réplica corretos processam a mesma seqüência de requisições. A confiabilidade do *multicast* garante que todos os gerenciadores de réplica corretos processam o mesmo conjunto de requisições e a ordem total garante que eles as processam na mesma ordem. Como são máquinas de estado, todos acabam no mesmo estado, após cada requisição. As requisições de cada *front end* são atendidas na ordem FIFO (pois o *front end* espera uma resposta antes de fazer a próxima requisição), que é igual à ordem do programa. Isso garante a consistência seqüencial.

Se os clientes não se comunicam com outros clientes enquanto esperam pelas respostas de suas requisições, então suas requisições são processadas na ordem do antes do acontecido. Se os clientes são *multi-threadeds*, e podem se comunicar uns com os outros enquanto esperam respostas do serviço, então, para garantir o processamento da requisição na ordem antes do acontecido, teríamos que substituir o *multicast* por um que tivesse ordenação causal e total.

O sistema de replicação ativa não obtém capacidade de linearização. Isso porque a ordem total na qual os gerenciadores de réplica processam as requisições não é necessariamente igual à ordem de tempo real na qual os clientes as fizeram. Schneider [1990] descreve como (em um sistema síncrono com relógios aproximadamente sincronizados) a ordem total na qual os gerenciadores de réplica processam requisições pode ser baseada na ordem das indicações de tempo físicas fornecidas pelos *front ends* em suas requisições. Isso não garante a capacidade de linearização, pois as indicações de tempo não são perfeitamente precisas; mas se aproxima disso.

Discussão sobre a replicação ativa ◊ Admitimos uma solução para o *multicast* totalmente ordenado e confiável. Conforme o Capítulo 12 mencionou, solucionar o *multicast* confiável e totalmente

ordenado é equivalente a solucionar o consenso. Por sua vez, solucionar o consenso exige que o sistema seja síncrono, ou, em um sistema assíncrono, se faça uso de uma técnica como o emprego de detectores de falhas para contornar o resultado da impossibilidade de Fischer *et al.* [1985].

Algumas soluções para o consenso, como a de Canetti e Rabin [1993], funcionam mesmo com a suposição de falhas bizantinas. Dada tal solução e, portanto, uma solução para o *multicast* totalmente ordenado e confiável, o sistema de replicação ativa pode mascarar até f falhas bizantinas, desde que o serviço incorpore pelo menos $2f + 1$ gerenciadores de réplica. Cada *front end* espera até ter reunido $f + 1$ respostas idênticas e passa essa resposta para o cliente. Ele descarta as outras respostas para a mesma requisição. Para estarmos rigorosamente seguros de qual resposta é realmente associada a qual requisição (dado o comportamento bizantino), exigimos que os gerenciadores de réplica acrescentem uma assinatura digital em suas respostas.

É possível flexibilizar o sistema que descrevemos. Primeiramente, admitimos que todas as atualizações nos objetos compartilhados replicados devem ocorrer na mesma ordem. Entretanto, na prática, algumas operações podem ser comutativas: isto é, o efeito de duas operações executadas na ordem $o_1;o_2$ é o mesmo da ordem inversa $o_2;o_1$. Por exemplo, quaisquer duas operações somente de leitura (de clientes diferentes) são comutativas; e quaisquer duas operações que não realizam leituras, mas atualizam objetos distintos são comutativas. Um sistema de replicação ativa pode explorar o conhecimento da propriedade comutativa para evitar a despesa de ordenar todas as requisições. Mencionamos, no Capítulo 12, que alguns têm proposto semânticas de ordenação por *multicast* específica ao aplicativo [Cheriton e Skeen 1993, Pedone e Schiper 1999].

Finalmente, os *front ends* podem enviar requisições somente de leitura apenas para gerenciadores de réplica individuais. Ao fazerem isso, eles perdem a tolerância a falhas que acompanha o *multicast* das requisições, mas o serviço continua tendo consistência seqüencial. Além disso, o *front end* pode mascarar facilmente a falha de um gerenciador de réplica; nesse caso, simplesmente enviando a requisição somente de leitura para outro gerenciador de réplica.

15.4 Estudos de caso de serviços de alta disponibilidade: Gossip, Bayou e Coda

Nesta seção, consideraremos como se faz para aplicar técnicas de replicação para tornar os serviços de alta disponibilidade. Nossa ênfase agora é o fornecimento de acesso ao serviço para os clientes – com tempo de resposta razoável – pelo máximo de tempo possível, mesmo que alguns resultados não estejam de acordo com a consistência seqüencial. Por exemplo, o usuário no trem, do início deste capítulo, pode querer aceitar as inconsistências temporárias entre cópias de dados, como as agendas, se puder continuar trabalhando enquanto estiver desconectado e corrigir os problemas posteriormente.

Na Seção 15.3, vimos que os sistemas tolerantes a falhas transmitem as atualizações para os gerenciadores de réplica de maneira "ávida": todos os gerenciadores de réplica corretos recebem as atualizações assim que possível e chegam a um acordo coletivo antes de devolverem o controle para o cliente. Esse comportamento é indesejável para a operação de alta disponibilidade. Em vez disso, o sistema deve fornecer um nível de serviço aceitável, usando um conjunto mínimo de gerenciadores de réplica conectados ao cliente. E ele deve minimizar o tempo durante o qual o cliente fica bloqueado, enquanto os gerenciadores de réplica coordenam suas atividades. Graus de consistência mais fracos geralmente exigem menos acordo e, portanto, permitem que dados compartilhados estejam disponíveis mais facilmente.

Examinaremos agora o projeto de três sistemas que fornecem serviços de alta disponibilidade: a arquitetura Gossip, Bayou e Coda.

15.4.1 A arquitetura Gossip

Ladin *et al.* [1992] desenvolveram o que chamamos de *arquitetura Gossip* (*fofoca*), como uma estrutura para implementar serviços de alta disponibilidade por meio da replicação de dados próximo aos pontos onde os grupos de clientes precisam deles. O nome reflete o fato de que os gerenciadores de

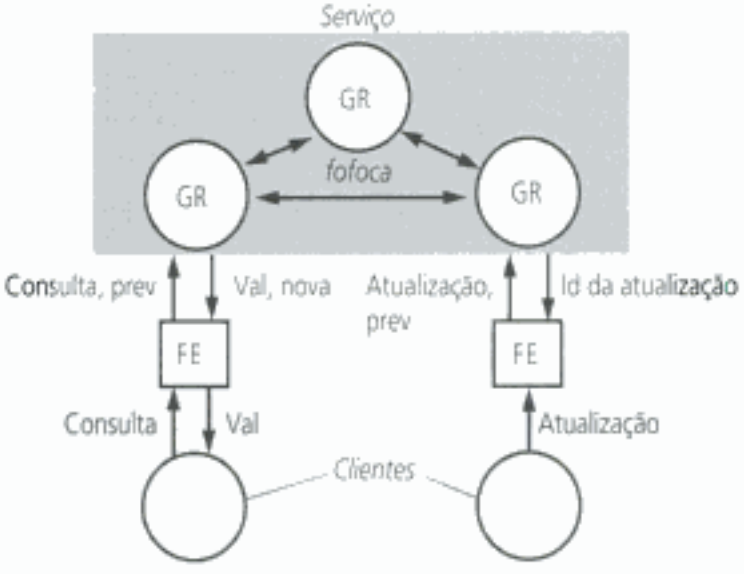


Figura 15.6 Operações de consulta e atualização em um serviço de "fofoca".

réplica trocam mensagens de "fofoca" periodicamente, para transmitir as atualizações que cada um recebeu dos clientes (veja a Figura 15.6). A arquitetura é baseada no trabalho anterior sobre bancos de dados feito por Fischer e Michael [1982] e por Wuu e Bernstein [1984]. Ela pode ser usada, por exemplo, para criar um lista de discussão eletrônica ou um serviço de agenda de alta disponibilidade.

Um serviço de fofoca fornece dois tipos básicos de operação: as consultas são operações somente de leitura e as atualizações modificam, mas não lêem o estado (esta última é uma definição mais restrita do que aquela que estivemos usando). Uma característica importante é que os *front ends* enviam consultas e atualizações para o gerenciador de réplica que escolherem – um que esteja disponível e que possa fornecer um tempo de resposta razoável. O sistema faz duas garantias, mesmo que os gerenciadores de réplica possam estar temporariamente incapazes de se comunicar uns com os outros:

> *Cada cliente obtém um serviço consistente com o passar do tempo*: na resposta a uma consulta, os gerenciadores de réplica só fornecem dados que reflitam pelo menos as atualizações observadas pelo cliente até o momento. Isso acontece mesmo que os clientes possam se comunicar com diferentes gerenciadores de réplica em diferentes momentos – e, portanto, em princípio, poderiam se comunicar com um gerenciador de réplica 'menos avançado' do que o que usaram antes.
>
> *Consistência suavizada entre réplicas*: todos os gerenciadores de réplica acabam por receber todas as atualizações e as aplicam com garantias de ordenação que tornam as réplicas suficientemente semelhantes para atenderem as necessidades do aplicativo. É importante perceber que, embora a arquitetura de fofoca possa ser usada para se obter consistência seqüencial, ela se destina principalmente a apresentar garantias de consistência mais fracas. Dois clientes podem observar réplicas diferentes, mesmo que as réplicas incluam o mesmo conjunto de atualizações; e um cliente pode observar dados antigos.

Para suportar consistência relaxada, a arquitetura de fofoca suporta ordem de atualização causal, conforme definimos na Seção 14.2.1. Ela também suporta garantias de ordenação mais fortes, na forma de ordenação *forçada* (total e causal) e *imediata*. As atualizações com ordenação imediata são aplicadas em uma ordem consistente em relação a *qualquer* outra atualização, em todos os gerenciadores de réplica, seja a ordem destas atualizações especificadas como causal, forçada ou imediata. Além da ordenação forçada é fornecida a ordenação imediata, pois uma atualização de ordenação forçada, ou uma atualização de ordem causal, que não possui uma relação antes do acontecido pode ser aplicada em diferentes ordens em diferentes gerenciadores de réplica.

A escolha da ordenação a ser usada é deixada para o projetista da aplicação e reflete um compromisso entre consistência e custos da operação. As atualizações causais são consideravelmente menos

Hidden page

como os gerenciadores de réplica processam consultas e atualizações. Grande parte do processamento das indicações de tempo vetoriais necessárias para manter atualizações causais é semelhante ao algoritmo de *multicast* causal da Seção 12.4.3.

A indicação de tempo (*timestamp*) de versão do *front end* ◊ Para controlar a ordenação do processamento de operação, cada *front end* mantém uma indicação de tempo vetorial (*vector timestamp*) que reflete a versão mais recente dos valores de dados acessados pelo *front end* (e, portanto, acessados pelo cliente). Essa indicação de tempo, denotada como *prev* na Figura 15.6, contém uma entrada para cada gerenciador de réplica. O *front end* a envia em cada mensagem de requisição para um gerenciador de réplica, junto com uma descrição da operação de consulta ou atualização em si. Quando um gerenciador de réplica retorna um valor como resultado de uma operação de consulta, ele fornece uma nova indicação de tempo vetorial (*nova*, na Figura 15.6), pois as réplicas podem ter sido atualizadas desde a última operação. Analogamente, uma operação de atualização retorna uma indicação de tempo vetorial (*id da atualização*, na Figura 15.6) que é exclusiva da atualização. Cada indicação de tempo retornada é mesclada com a indicação de tempo anterior do *front end*, para registrar a versão dos dados duplicados que foram observados pelo cliente. (Veja na Seção 11.4 uma definição de integração de indicações de tempo vetorial.)

Os clientes trocam dados acessando o mesmo serviço de fofoca e comunicando-se diretamente uns com os outros. Como a comunicação de cliente para cliente também pode levar a relacionamentos causais entre as operações aplicadas ao serviço, ela também ocorre por intermédio dos *front ends* dos clientes. Desse modo, os *front ends* podem levar suas indicações de tempo vetoriais "de carona" nas mensagens para outros clientes. Os destinatários as integram com suas próprias indicações de tempo, para que os relacionamentos causais possam ser deduzidos corretamente. A situação é mostrada na Figura 15.7

Estado do gerenciador de réplica ◊ Independentemente da aplicação, um gerenciador de réplica contém os seguintes componentes principais de estado (Figura 15.8):

Valor: é o valor do estado da aplicação, mantido pelo gerenciador de réplica. Cada gerenciador de réplica é uma máquina de estado, a qual começa com um valor inicial especificado e que daí em diante é apenas o resultado da aplicação de operações de atualização nesse estado.

Indicação de tempo do valor: é a indicação de tempo vetorial que representa as atualizações refletidas no valor. Ela contém uma entrada para cada gerenciador de réplica. É atualizada quando uma operação de atualização é aplicada no valor.

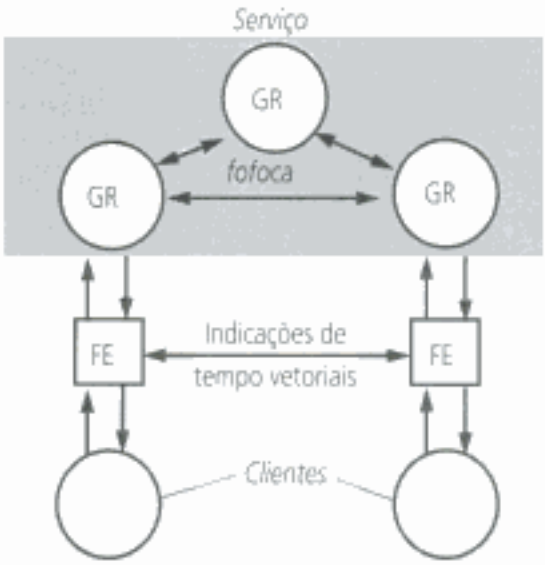


Figura 15.7 Os *front ends* propagam suas indicações de tempo quando os clientes se comunicam diretamente.

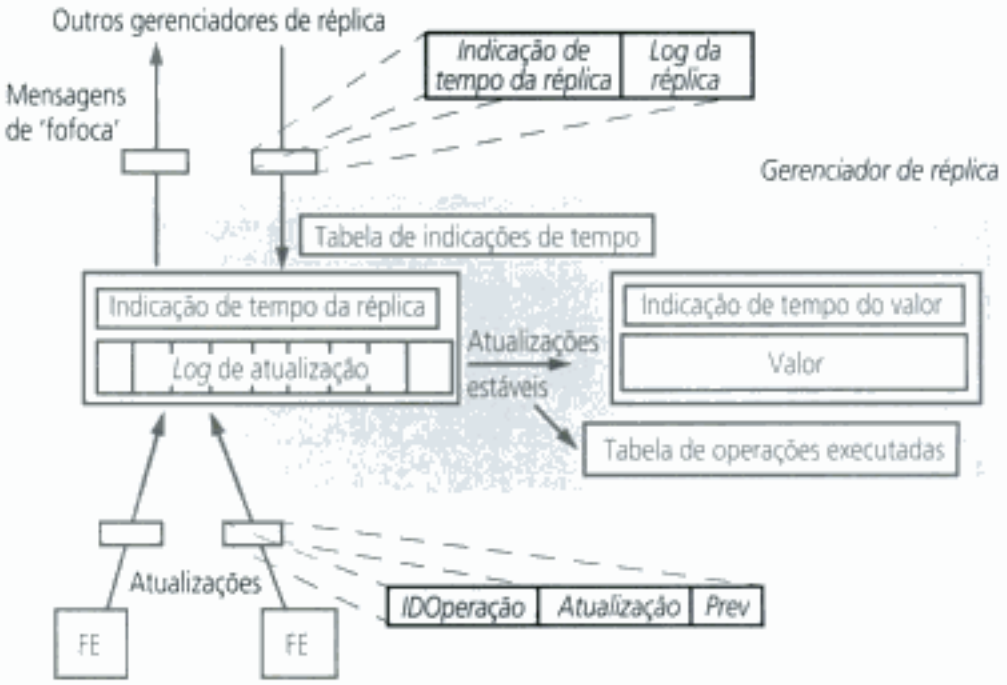


Figura 15.8 Um gerenciador de réplica fofoca mostrando seus principais componentes de estado.

Log de atualização: todas as operações de atualização são registradas nesse *log*, assim que são recebidas. Um gerenciador de réplica mantém as atualizações em um *log* por dois motivos. O primeiro é que o gerenciador de réplica ainda não pode aplicar a atualização, porque ela ainda não é *estável*. Uma atualização estável é aquela que pode ser aplicada consistentemente com suas garantias de ordenação (causal, forçada ou imediata). Uma atualização que ainda não é estável deve ser retida e ainda não processada. O segundo motivo para se manter uma atualização no *log* é que, mesmo que a atualização tenha se tornado estável e aplicada ao valor, o gerenciador de réplica não recebeu confirmação de que essa atualização foi recebida em todos os outros gerenciadores de réplica. Nesse meio-tempo, ele propaga a atualização em mensagens de fofoca.

Indicação de tempo da réplica: esta indicação de tempo vetorial representa as atualizações que foram aceitas pelo gerenciador de réplica – isto é, colocadas no *log* do gerenciador. Ela difere do valor indicação de tempo em geral, é claro, pois nem todas as atualizações do *log* são estáveis.

Tabela de operações executadas: a mesma atualização pode chegar de um *front end* a determinado gerenciador de réplica e em mensagens de fofoca de outros gerenciadores de réplica. Para impedir que uma atualização seja aplicada duas vezes, é mantida a tabela de operações executadas, contendo os identificadores únicos, fornecidos pelo *front end*, das atualizações que foram aplicadas ao valor. Os gerenciadores de réplica consultam essa tabela antes de adicionar uma atualização no *log*.

Tabela de indicações de tempo: essa tabela contém uma indicação de tempo vetorial de todos os outros gerenciadores de réplica, preenchida com indicações de tempo que chegam deles em mensagens de fofoca. Os gerenciadores de réplica usam a tabela para estabelecer quando uma atualização foi aplicada em todos os gerenciadores de réplica.

Os gerenciadores de réplica são numerados como 0, 1, 2, ..., e o i-ésimo elemento de uma indicação de tempo vetorial mantido pelo gerenciador de réplica i corresponde ao número de atualizações recebidas dos *front ends* por i; e o j-ésimo componente ($j \neq i$) é igual ao número de atualizações recebidas por j e propagadas para i em mensagens de fofoca. Assim, por exemplo, em um sistema de fofoca de três gerenciadores, uma indicação de tempo de valor igual a (2,4,5) no gerenciador 0 representaria o fato de que o valor reflete as duas primeiras atualizações aceitas dos *front ends* no gerenciador 0, as quatro primeiras no gerenciador 1 e as cinco primeiras no gerenciador 2. A seguir, veremos com mais detalhes como as indicações de tempo são usadas para impor a ordem.

Hidden page

ao valor. Se essa condição não for satisfeita no momento em que a atualização for enviada, ela será verificada novamente, quando as mensagens de fofoca chegarem. Quando a condição de estabilidade tiver sido satisfeita para um registro de atualização *r*, o gerenciador de réplica aplicará a atualização no valor, acertará a indicação de tempo do valor e a tabela de operações executadas *executed*:

valor:= *apply*(*valor*, *r.u.op*)

valueTS:= *merge*(*valueTS*, *r.ts*)

executed := *executed* ∪ {*r.u.id*}

A primeira dessas três declarações representa a aplicação da atualização no valor. Na segunda declaração, a indicação de tempo da atualização é integrada com a do valor. Na terceira, o identificador de operação da atualização é adicionado ao conjunto de identificadores das operações que foram executadas – o qual é usado para verificar a existência de requisições de operação repetidas.

Operações de atualização forçadas e imediatas ◊ As atualizações forçadas e imediatas exigem tratamento especial. Lembre-se de que as atualizações forçadas são ordenadas totalmente, assim como de forma causal. O método básico para ordenar atualizações forçadas é anexar um número de seqüência exclusivo nas indicações de tempo associadas a elas e processá-las na ordem desse número de seqüência. Conforme o Capítulo 12 explicou, um método geral para gerar números de seqüência é usar um processo seqüenciador exclusivo. Mas a confiança em um processo exclusivo é inadequada no contexto de um serviço de alta disponibilidade. A solução é designar um assim chamado *gerenciador de réplica primário* como seqüenciador, em dado momento, mas garantir que outro gerenciador de réplica possa ser eleito para assumir consistentemente como seqüenciador, caso o gerenciador primário venha a falhar. O que é exigido é que uma maioria de gerenciadores de réplica (incluindo o primário) registre qual atualização é a próxima na seqüência, antes que a operação possa ser aplicada. Então, desde que uma maioria dos gerenciadores de réplica sobreviva à falha, essa decisão sobre a ordenação será honrada por um novo gerenciador primário eleito dentre os gerenciadores de réplica sobreviventes.

As atualizações imediatas são ordenadas com relação às atualizações forçadas, usando o gerenciador de réplica primário para ordená-las nessa seqüência. O gerenciador primário também determina quais atualizações causais são consideradas como precedentes a uma atualização imediata. Ele faz isso se comunicando e sincronizando-se com os outros gerenciadores de réplica em ordem, para chegar a um acordo sobre isso. Mais detalhes são fornecidos em Ladin *et al.* [1992].

Mensagens de fofoca ◊ Os gerenciadores de réplica enviam mensagens de fofoca contendo informações a respeito de uma ou mais atualizações, para que os outros gerenciadores de réplica possam atualizar seus estados. Um gerenciador de réplica usa as entradas de sua tabela de indicação de tempo para fazer uma estimativa de quais atualizações qualquer outro gerenciador de réplica ainda não recebeu (trata-se de uma estimativa, pois agora esse gerenciador de réplica pode ter recebido mais atualizações).

Uma mensagem de fofoca *m* consiste em dois itens enviados pelo gerenciador de réplica de origem: seu log *m.log* e sua indicação de tempo de réplica *m.ts* (veja a Figura 15.8). O gerenciador de réplica que recebe uma mensagem de fofoca tem três tarefas principais:

- Integrar o *log* recebido com o seu próprio (ele pode conter atualizações não vistas anteriormente pelo receptor).
- Aplicar as atualizações que se tornaram estáveis e não foram executadas antes (por sua vez, as atualizações estáveis presentes no *log* recebido podem tornar estáveis as atualizações pendentes).
- Eliminar registros do *log* e entradas na tabela de operações executadas, quando for conhecido que as atualizações foram aplicadas por toda parte e para as quais não há perigo de repetições. Eliminar as entradas redundantes do *log* e da tabela de operações executadas é uma tarefa importante, pois de outro modo elas cresceriam sem limite.

É simples integrar o *log* contido em uma mensagem de fofoca recebida com o *log* do receptor. Seja *replicaTS* a indicação de tempo de réplica do destinatário. Um registro *r* em *m.log* é adicionado no *log* do receptor, a não ser que $r.ts \leq replicaTS$ – no caso em que ele já está no *log* ou foi aplicado ao valor e depois descartado.

O gerenciador de réplica integra a indicação de tempo da mensagem de fofoca recebida com sua própria indicação de tempo de réplica *replicaTS*, para que ela corresponda às adições feitas no *log*:

$replicaTS := merge(replicaTS, m.ts)$

Quando novos registros de atualização tiverem sido integrados no *log*, o gerenciador de réplica reunirá o conjunto S de todas as atualizações no *log*, que agora são estáveis. Elas podem ser aplicadas ao valor, mas é preciso cuidado com a ordem com que elas são aplicadas, para que a relação antes do acontecido seja observada. O gerenciador de réplica ordena as atualizações no conjunto, de acordo com a ordem parcial '$\leq$' entre as indicações de tempo vetoriais. Então, ele aplica as atualizações nessa ordem, com a menor vindo primeiro. Isto é, cada $r \in S$ é aplicado apenas quando não existe nenhum $s \in S$ tal que $s.prev < r.prev$.

Então, o gerenciador de réplica procura registros no *log* que possam ser descartados. Se a mensagem de fofoca foi enviada pelo gerenciador de réplica j e se *tableTS* é a tabela de indicações de tempo de réplica dos gerenciadores de réplica, então o gerenciador de réplica configura

$tableTS[j] := m.ts$

Agora, o gerenciador de réplica pode descartar qualquer registro r do *log* para uma atualização que tenha sido recebida em toda parte. Isto é, se c é o gerenciador de réplica que criou o registro, então exigimos que todos os gerenciadores de réplica i:

$tableTS[i][c] \geq r.ts[c]$

A arquitetura de fofoca também define como os gerenciadores de réplica podem descartar entradas da tabela de operações executadas. É importante não descartar essas entradas cedo demais; caso contrário, uma operação muito atrasada poderia ser aplicada duas vezes por engano. Ladin *et al.* [1992] fornecem detalhes do esquema. Basicamente, os *front ends* emitem sinais de reconhecimento nas respostas de suas atualizações, de modo que os gerenciadores de réplica sabem quando um *front end* parará de enviar a atualização. Eles presumem um atraso de propagação de atualização máximo a partir desse ponto.

Propagação de atualização ◊ A arquitetura de fofoca não especifica quando os gerenciadores de réplica trocam mensagens de fofoca, nem como um gerenciador de réplica escolhe outros gerenciadores de réplica para enviar uma fofoca. É necessária uma estratégia de propagação de atualização robusta, caso todos os gerenciadores de réplica devam receber todas as atualizações em um tempo aceitável.

O tempo que leva para todos os gerenciadores de réplica receberem determinada atualização depende de três fatores:

- A freqüência e duração dos particionamentos da rede.
- A freqüência com que os gerenciadores de réplica enviam mensagens de fofoca.
- A política de escolha de um parceiro com o qual trocar fofoca.

O primeiro fator está fora do controle do sistema, embora os usuários possam determinar, até certo ponto, com que freqüência eles trabalham desconectados.

A freqüência de troca de fofoca desejada pode ser ajustada na própria aplicação. Considere um sistema de lista de discussões eletrônica compartilhada entre vários sites. Parece desnecessário que cada item seja enviado imediatamente para todos os sites. Mas, e se a fofoca for trocada apenas após longos períodos de tempo, digamos, uma vez por dia? Se forem usadas apenas atualizações causais, então será bem possível que os clientes em cada site tenham seus próprios debates consistentes pela mesma lista de discussão, isolados das discussões que ocorrem em outros sites. Então, digamos, à meia-noite, todos os debates serão integrados; mas os debates sobre o mesmo assunto provavelmente serão incongruentes, quando teria sido preferível que eles levassem em conta uns aos outros. Um período de troca de fofoca de alguns minutos, ou mesmo horas, parece mais apropriado nesse caso.

Existem vários tipos de política de escolha de parceiro. Golding e Long [1993] consideram as políticas *aleatória*, *determinista* e *topológica* para seu protocolo anti-entropia com indicação de tempo, o qual usa um esquema de propagação de atualização estilo fofoca.

As políticas aleatórias escolhem um parceiro ao acaso, mas com probabilidades ponderadas, de modo a favorecer alguns parceiros em detrimento de outros como, por exemplo, escolher os parceiros próximos. Golding e Long descobriram que tal política funciona surpreendentemente bem sob simulações. As políticas deterministas utilizam uma função simples do estado do gerenciador de réplica para fazer a escolha do parceiro. Por exemplo, um gerenciador de réplica poderia examinar sua tabela de indicação de tempo e escolher o gerenciador de réplica que pareça estar mais atrasado nas atualizações recebidas.

As políticas topológicas organizam os gerenciadores de réplica em um grafo fixo. Uma possibilidade é uma malha: os gerenciadores de réplica enviam mensagens de fofoca para os quatro gerenciadores de réplica com que está diretamente conectado. Outra é organizar os gerenciadores de réplica em um anel, com cada um passando a fofoca apenas para seu vizinho (digamos, no sentido horário), de modo que as atualizações de qualquer gerenciador de réplica finalmente percorram o anel todo. Existem muitas outras topologias possíveis, incluindo as árvores.

Diferentes políticas de escolha de parceiro como essas ponderam a quantidade de comunicação em relação às latências de transmissão mais altas e a possibilidade de que uma única falha afete outros gerenciadores de réplica. Na prática, a escolha depende da importância relativa desses fatores. Por exemplo, a topologia em anel produz relativamente pouca comunicação, mas está sujeita às altas latências de transmissão, pois a fofoca geralmente precisa passar por vários gerenciadores de réplica. Além disso, se um gerenciador de réplica falhar, o anel não funcionará e precisará ser reconfigurado. Em contraste, a política de escolha aleatória não é suscetível às falhas, mas pode produzir tempos de propagação de atualização mais variáveis.

Discussão sobre a arquitetura de fofoca ◊ A arquitetura de fofoca tem como objetivo obter alta disponibilidade para os serviços. A seu favor está o fato de que os clientes podem continuar a obter um serviço mesmo quando são separados do resto da rede, desde que pelo menos um gerenciador de réplica continue a funcionar na partição. Mas esse tipo de disponibilidade é obtido à custa da imposição de garantias de consistência suavizadas. Para objetos como contas bancárias, onde a consistência seqüencial é obrigatória, uma arquitetura de fofoca não é melhor do que os sistemas tolerantes a falhas estudados na Seção 15.3 e fornece o serviço apenas em uma partição onde haja maioria.

A estratégia "preguiçosa" de propagação de atualização torna um sistema baseado em fofoca inadequado para atualizar réplicas em um tempo próximo ao real, como quando os usuários tomam parte de uma conferência em tempo real e atualizam um documento compartilhado. Um sistema baseado em *multicast* seria mais apropriado para esse caso.

A escalabilidade de um sistema de fofoca é outro problema. À medida que o número de gerenciadores de réplica aumenta, também aumenta o número de mensagens de fofoca que precisam ser transmitidas e o tamanho das indicações de tempo usadas. Se um cliente faz uma consulta, normalmente isso exige duas mensagens (entre o *front end* e o gerenciador de réplica). Se um cliente faz uma operação de atualização causal, e se cada um dos R gerenciadores de réplica normalmente reúne G atualizações em uma mensagem de fofoca, então o número de mensagens trocadas é de $2 + (R - 1)/G$. O primeiro termo representa a comunicação entre o *front end* e o gerenciador de réplica e o segundo é a parte da atualização de uma mensagem de fofoca enviada para os outros gerenciadores de réplica. Aumentar G melhora o número de mensagens; mas piora as latências de envio, pois o gerenciador de réplica espera que mais atualizações cheguem, antes de propagá-las.

Uma estratégia para fazer com que os serviços baseados em fofoca possam mudar de escala é tornar a maioria das réplicas somente de leitura. Em outras palavras, essas réplicas são atualizadas pelas mensagens de fofoca, mas não recebem atualizações diretamente dos *front ends*. Esse arranjo é potencialmente útil onde a relação *atualização/consulta* é pequena. As réplicas somente de leitura podem estar situadas próximas aos grupos de clientes e as atualizações podem ser atendidas por relativamente poucos gerenciadores de réplica centralizados. O tráfego de fofoca é reduzido, pois as réplicas somente de leitura não têm nenhuma fofoca para propagar. E as indicações de tempo vetoriais só precisam conter entradas para as réplicas que podem ser atualizadas.

15.4.2 Bayou e a estratégia da transformação operacional

O sistema Bayou [Terry *et al.* 1995, Petersen *et al.* 1997] fornece replicação de dados de alta disponibilidade com garantias menores do que a consistência seqüencial, como a arquitetura de fofoca e o protocolo anti-entropia com indicação de tempo. Assim como nesses sistemas, os gerenciadores de réplica Bayou suportam conectividade variável, trocando atualizações em pares, nas quais os projetistas também designam um protocolo anti-entropia. Mas o Bayou adota uma estratégia marcadamente diferente, pois permite a ocorrência de detecção de conflitos específica do domínio e solução de conflitos.

Considere o usuário que precisa atualizar uma agenda, enquanto trabalha de forma desconectada. Se for exigida consistência restrita, então na arquitetura de fofoca as atualizações seriam realizadas usando-se uma operação forçada (totalmente ordenada). Mas, então, apenas os usuários de uma divisão onde houvesse maioria poderia atualizar a agenda. Assim, o acesso dos usuários à agenda pode ser limitado – independentemente de precisarem, de fato, fazer atualizações que violariam a integridade da agenda. Os usuários que quiserem marcar um compromisso não conflitante serão tratados de forma igual aos usuários que podem ter marcado uma repartição de hora duas vezes, inconscientemente.

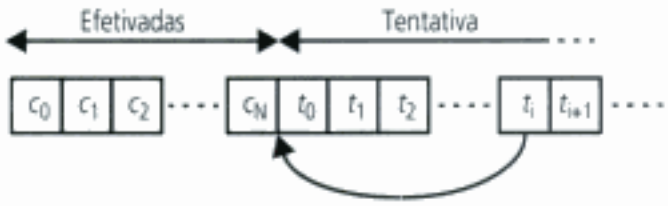
No Bayou, em contraste, os usuários que estão no trem e no trabalho podem fazer as atualizações que desejarem. Todas elas serão aplicadas e registradas no gerenciador de réplica que encontrarem. Entretanto, quando as atualizações recebidas em quaisquer dois gerenciadores de réplica são integradas, durante uma troca anti-entropia, os gerenciadores de réplica detectam e solucionam os conflitos. Qualquer critério de conflito específico do domínio entre operações pode ser aplicado. Por exemplo, se um executivo e sua secretária tiverem adicionado compromissos na mesma repartição de hora, um sistema Bayou detectará isso, depois que o executivo tiver reconectado seu laptop. Além disso, ele soluciona o conflito de acordo com uma política específica do domínio. Nesse caso, ele poderia, por exemplo, confirmar o compromisso do executivo e remover o da secretária na repartição de hora. Tal efeito, no qual uma ou mais operações de um conjunto conflitante são desfeitas, ou alteradas, para solucioná-las, é chamado de *transformação operacional*.

O estado que o Bayou replica é mantido na forma de um banco de dados, suportando consultas e atualizações (que podem inserir, modificar ou excluir itens no banco de dados). Embora não nos concentremos nesse aspecto aqui, uma atualização do Bayou é um caso especial de transação. Ela consiste em uma única operação, uma invocação de um procedimento armazenado, o qual afeta vários objetos dentro de cada gerenciador de réplica, mas que é executada com as garantias ACID. O Bayou pode desfazer e refazer as atualizações no banco de dados, à medida que a execução prossegue.

A garantia do Bayou é que, finalmente, cada gerenciador de réplica recebe o mesmo conjunto de atualizações e finalmente aplica essas atualizações de maneira tal que os bancos de dados dos gerenciadores de réplica sejam idênticos. Na prática, pode haver um fluxo contínuo de atualizações e os bancos de dados podem nunca se tornar idênticos; mas eles se tornariam idênticos se as atualizações cessassem.

Atualizações efetivadas e de tentativa ◊ As atualizações são marcadas como *de tentativa* quando são aplicadas a um banco de dados pela primeira vez. O Bayou faz com que as atualizações de tentativa sejam finalmente colocadas em uma ordem canônica e marcadas como *efetivadas*. Enquanto as atualizações são de tentativa, o sistema pode desfazê-las e reaplicá-las, à medida que produz um estado consistente. Uma vez efetivadas, elas permanecem aplicadas em sua ordem definida. Na prática, a ordem efetivada pode ser obtida designando-se algum gerenciador de réplica como gerenciador de réplica *primário*. Como sempre, ele decide a ordem efetivada como aquela na qual recebe as atualizações de tentativa e propaga essa informação de ordenação para os outros gerenciadores de réplica. Como gerenciador primário, os usuários podem escolher, por exemplo, uma máquina rápida que normalmente esteja disponível; igualmente, ele poderia ser o gerenciador de réplica no laptop do executivo, caso as atualizações desse usuário tivessem prioridade.

A qualquer momento, o estado de uma réplica de banco de dados é derivado de uma seqüência (possivelmente vazia) de atualizações efetivadas, seguida de uma seqüência (possivelmente vazia) de atualizações de tentativa. Se a próxima atualização efetivada chegar, ou se uma das atualizações de tentativa aplicada se tornar a próxima atualização efetivada, deverá ocorrer então uma reordenação das atualizações. Na Figura 15.9, t_j se tornou efetivada. Todas as atualizações de tentativa após c_N precisam ser desfeitas; então, t_i é aplicada após c_N e t_0 a t_{i-1} e t_{i+1} etc., são reaplicadas após t_i.



A atualização de tentativa t_i se torna a próxima atualização efetivada e é inserida após a última atualização efetivada c_N.

Figura 15.9 Atualizações efetivadas e de tentativa no Bayou.

Verificações de dependência e procedimentos de integração ◊ Uma atualização pode entrar em conflito com alguma outra operação que já tenha sido aplicada. Devido a essa possibilidade, toda atualização do Bayou contém uma *verificação de dependência* e um *procedimento de integração* além da especificação da operação (o tipo e os parâmetros da operação). Todos esses componentes de uma atualização são específicos do domínio.

Um gerenciador de réplica ativa o procedimento de verificação de dependência, antes de aplicar a operação. Ele verifica se ocorreria um conflito caso a atualização fosse aplicada e, para fazer isso, pode examinar qualquer parte do banco de dados. Por exemplo, considere o caso do registro de um compromisso em uma agenda. A verificação de dependência poderia, mais simplesmente, testar a existência de um conflito de *escrita-escrita*: isto é, se outro cliente preencheu a repartição de hora exigida. Mas a verificação de dependência também poderia testar a existência de um conflito de *leitura-escrita*. Por exemplo, ela poderia testar se a repartição de hora desejada está vazia e se o número de compromissos nesse dia é menor do que seis.

Se a verificação de dependência indicar um conflito, então o Bayou ativará o procedimento de integração da operação. Esse procedimento altera a operação que será aplicada, de modo que ela obtenha algo semelhante ao efeito pretendido, mas evite um conflito. Por exemplo, no caso da agenda, o procedimento de integração poderia escolher outra repartição de hora próxima ou, conforme mencionado anteriormente, poderia usar um esquema de prioridade simples para decidir qual compromisso é mais importante e impô-lo. O procedimento de integração pode não encontrar uma alteração conveniente para a operação, no caso em que o sistema indicará um erro. Entretanto, o efeito de um procedimento de integração é determinista – os gerenciadores de réplica Bayou são máquinas de estado.

Discussão ◊ O Bayou difere dos outros esquemas de replicação que consideramos, pois torna a replicação não transparente para a aplicação. Ele explora o conhecimento da semântica da aplicação para aumentar a disponibilidade dos dados, enquanto mantém um estado replicado, que é o que poderíamos chamar de *consistência seqüencial final*.

As desvantagens dessa estratégia são, primeiramente, a maior complexidade para o programador da aplicação que precisa fornecer verificações de dependência e procedimentos de integração. Ambos podem ser complexos de produzir, dado o grande número de possíveis conflitos que precisam ser detectados e solucionados. A segunda desvantagem é a maior complexidade para o usuário. Não apenas se espera que os usuários lidem com dados que são lidos enquanto ainda são de tentativa, como também com o fato de que a operação especificada por um usuário pode ser alterada. Por exemplo, o usuário marcou uma repartição de hora em uma agenda, apenas para descobrir posteriormente que a marcação 'pulou' para uma repartição de hora próxima. É muito importante que o usuário receba uma indicação clara de quais dados são de tentativa e quais são efetivados.

A estratégia da transformação operacional usada pelo Bayou aparece particularmente nos sistemas para suportar trabalho cooperativo apoiado por computador (CSCW, do nome em inglês, *Computer Supported Cooperative Working*), onde podem ocorrer atualizações conflitantes entre usuários separados geograficamente [Kindberg *et al.* 1996], [Sun e Ellis 1998]. Na prática, a estratégia é limitada às aplicações onde os conflitos são relativamente raros, onde a semântica dos dados subjacentes é relativamente simples e onde os usuários podem aceitar informações de tentativa.

Hidden page

Normalmente, o acesso a arquivo do Coda ocorre de maneira semelhante ao do AFS, com cópias dos arquivos armazenadas na cache sendo fornecidas para os computadores clientes por qualquer um dos servidores no AVSG corrente. Assim como no AFS, os clientes são notificados das alterações por meio de um mecanismo de *promessa de callback*, mas isso agora depende de um mecanismo adicional para a distribuição de atualizações para cada réplica. No *fechamento de um arquivo* (*close*), as cópias dos arquivos modificados são transmitidas em paralelo para todos os servidores no AVSG.

No Coda, diz-se que a operação desconectada ocorre quando o AVSG está vazio. Isso pode ser devido a falhas de rede ou servidor, ou pode ser uma conseqüência da desconexão deliberada do computador cliente, como no caso de um laptop. A operação desconectada efetiva conta com a presença, na cache do computador cliente, de *todos* os arquivos exigidos para que o trabalho do usuário prossiga. Para conseguir isso, o usuário precisa colaborar com o Coda na geração de uma lista de arquivos que devem ser colocados na cache. É fornecida uma ferramenta que registra, em uma lista, o histórico da utilização dos arquivos enquanto está na operação conectada e isso serve como base para prever a utilização enquanto está na operação desconectada.

Um princípio de projeto do Coda é o fato de que as cópias dos arquivos residentes nos servidores sejam mais confiáveis do que aquelas que residem nas caches dos computadores clientes. Embora seja possível construir logicamente um sistema de arquivos que conte inteiramente com cópias de arquivos armazenadas na cache em computadores clientes, é improvável que seja obtida uma qualidade de serviço satisfatória. Os servidores Coda existem para fornecer a qualidade de serviço necessária. As cópias dos arquivos residentes nas caches do computador cliente são consideradas úteis somente enquanto sua validade puder ser periodicamente revalidada em relação às cópias residentes nos servidores. No caso da operação desconectada, a revalidação ocorre quando a operação desconectada cessa e os arquivos armazenados na cache são reintegrados com os dos servidores. No pior caso, isso pode exigir alguma intervenção manual para resolver inconsistências ou conflitos.

A estratégia da replicação ◊ A estratégia de replicação do Coda é otimista – ela permite que a modificação dos arquivos prossiga quando a rede é particionada ou durante a operação desconectada. Ela conta com a anexação, em cada versão de um arquivo, de um *vetor de versão Coda* (CVV, do original em inglês, *Coda Version Vector*). Um CVV é uma indicação de tempo vetorial com um elemento para cada servidor no VSG relevante. Cada elemento do CVV é uma estimativa do número de modificações realizadas na versão do arquivo mantida no servidor correspondente. O objetivo dos CVVs é fornecer informações suficientes sobre o histórico de atualização de cada réplica de arquivo para permitir que conflitos em potencial possam ser detectados e submetidos à intervenção manual e para que as réplicas antigas sejam atualizadas automaticamente.

Se o CVV de um dos sites for maior ou igual a todos os CVVs correspondentes nos outros sites (a Seção 11.4 definiu o significado de $v_1 \geq v_2$ para indicações de tempo vetoriais v_1 e v_2), então não existe nenhum conflito. As réplicas mais antigas (com indicações de tempo rigorosamente menores) incluem todas as atualizações em uma réplica mais recente e podem ser atualizadas automaticamente com ela.

Quando isso não acontece, isto é, quando nem $v_1 \geq v_2$, nem $v_2 \geq v_1$, valem para dois CVVs, então existe um conflito: cada réplica reflete pelo menos uma atualização que a outra não reflete. Em geral, o Coda não resolve conflitos automaticamente. O arquivo é marcado como "inoperante" e o proprietário do arquivo é informado sobre o conflito.

Quando um arquivo modificado é fechado, cada site presente no AVSG corrente recebe uma mensagem de atualização enviada pelo processo Venus no cliente, contendo o CVV corrente e o novo conteúdo do arquivo. O processo Vice em cada site verifica o CVV e, se ele for maior do que o correntemente mantido, armazena o novo conteúdo do arquivo e retorna um sinal de reconhecimento positivo. Então, o processo Venus calcula um novo CVV com contagens de modificação aumentadas para os servidores que responderam positivamente à mensagem de atualização e distribui o novo CVV para os membros do AVSG.

Como a mensagem é enviada apenas para os membros do AVSG, e não do VSG, os servidores que não estão no AVSG corrente não recebem o novo CVV. Portanto, qualquer CVV sempre conterá uma contagem de modificação precisa do servidor local, mas as contagens dos servidores não locais, em geral, serão menores, pois só serão atualizadas quando o servidor receber uma mensagem de atualização.

O quadro a seguir contém um exemplo ilustrando o uso de CVVs para gerenciar a atualização de um arquivo replicado em três sites. Mais detalhes sobre o uso de CVVs para o gerenciamento de atualizações podem ser encontrados em Satyanarayanan *et al.* [1990]. Os CVVs são baseados nas técnicas de replicação usadas no sistema Locus [Popek e Walker 1985].

Na operação normal, o comportamento do Coda parece semelhante ao AFS. Uma perda de cache é transparente para os usuários e apenas impõe uma penalidade no desempenho. As vantagens derivadas da replicação de alguns, ou de todos, volumes de arquivo em vários servidores são:

- Os arquivos em um volume replicado permanecem acessíveis para qualquer cliente que possa acessar pelo menos uma das réplicas.
- O desempenho do sistema pode ser melhorado pelo compartilhamento, entre todos os servidores que contêm réplicas, de parte do trabalho do atendimento às requisições de cliente em um volume replicado.

Na operação desconectada (quando nenhum dos servidores de um volume pode ser acessado pelo cliente) uma perda de cache impede qualquer avanço e a computação é suspensa até que a conexão seja restabelecida, ou o usuário cancele o processo. Portanto, é importante carregar a cache antes que a operação desconectada comece, para que as perdas possam ser evitadas.

Em resumo, comparado com o AFS, o Coda melhora a disponibilidade tanto pela replicação de arquivos nos servidores como pela capacidade dos clientes operarem inteiramente fora de suas caches. Os dois métodos dependem do uso de uma estratégia otimista para a detecção de conflitos de atualização na presença de particionamentos da rede. Os mecanismos são complementares e independentes entre si. Por exemplo, um usuário pode explorar os benefícios da operação desconectada, mesmo que os volumes de arquivo exigidos estejam armazenados em um único servidor.

Exemplo: considere uma seqüência de modificações em um arquivo F, em um volume que é replicado em 3 servidores, S_1, S_2 e S_3. O VSG de F é $\{S_1, S_2, S_3\}$. F é modificado praticamente ao mesmo tempo por dois clientes C_1, C_2. Devido a uma falha da rede, C_1 só pode acessar S_1 e S_2 (o AVSG de C_1 é $\{S_1, S_2\}$) e C_2 só pode acessar S_3 (o AVSG de C_2 é $\{S_3\}$).

1. Inicialmente, os CVVs de F em todos os 3 servidores são iguais, digamos [1,1,1].
2. C_1 executa um processo que abre F, o modifica e depois fecha. O processo Venus em C_1 transmite uma mensagem de atualização para seu AVSG, $\{S_1, S_2\}$, resultando finalmente nas novas versões de F e um CVV [2,2,1] em S_1 e S_2, mas não muda em S_3.
3. Nesse meio-tempo, C_2 executa dois processos, cada um dos quais abre F, o modifica e depois fecha. O processo Venus em C_2 transmite uma mensagem de atualização para seu AVSG, $\{S_3\}$, após cada modificação, resultando finalmente em uma nova versão de F e um CVV [1,1,3] em S_3.
4. Algum tempo depois, a falha da rede é reparada e C_2 realiza uma verificação de rotina para ver se os membros inacessíveis do VSG se tornaram acessíveis (o processo por meio do qual tais verificações são feitas será descrito posteriormente) e descobre que S_1 e S_2 agora estão acessíveis. Ele modifica seu AVSG para $\{S_1, S_2, S_3\}$, para o volume que contém F, e solicita os CVVs de F de todos os membros do novo AVSG. Quando eles chegam, C_2 descobre que S_1 e S_2 tem, cada um, os CVVs [2,2,1], enquanto S_3 tem [1,1,3]. Isso representa um *conflito* exigindo intervenção manual para atualizar F, de maneira que minimize a perda de informações de atualização.

Por outro lado, considere um cenário semelhante, porém, mais simples, que segue a mesma seqüência de eventos acima, mas omitindo o item (3), de modo que F não é modificado por C_2. Portanto, o CVV em S_3 permanece inalterado como [1,1,1] e, quando a falha da rede é reparada, C_2 descobre que os CVVs em S_1 e S_2 ([2,2,1]) *predominam* sobre o de S_3. A versão do arquivo em S_1 ou S_2 deve substituir a que está em S_3.

Semântica de atualização ◊ As garantias de vigência oferecidas pelo Coda quando um cliente abre um arquivo são menores do que as do AFS, refletindo a estratégia de atualização otimista. O servidor único S, referenciado nas garantias de vigência do AFS, é substituído por um conjunto de servidores $\bar{S}$ (o VSG do arquivo) e o cliente C pode acessar um subconjunto de servidores $\bar{s}$ (o AVSG do arquivo visto por C).

Informalmente, a garantia oferecida por uma operação de *abertura de arquivo* (*open*) bem-sucedida no Coda é que ela fornece a cópia mais recente de F do AVSG corrente e, se nenhum servidor estiver acessível, uma cópia de F armazenada na cache local será usada, caso haja uma disponível. Uma operação de *fechamento* (*close*) bem-sucedida garante que o arquivo foi propagado para o conjunto de servidores correntemente acessíveis ou, se nenhum servidor estiver disponível, que o arquivo foi marcado para propagação na próxima oportunidade.

Uma definição mais precisa dessas garantias, levando em conta o efeito de *callbacks* perdidos, pode ser dada usando-se uma extensão da notação usada pelo AFS. Em cada definição, exceto a última, existem dois casos: o primeiro, começando com $\bar{s} \neq \emptyset$, se refere a todas as situações em que o AVSG não está vazio; e o segundo trata da operação desconectada:

após uma operação de *abertura* bem-sucedida:	$(\bar{s} \neq \emptyset$ *e* $(latest(F, \bar{s}, 0)$
	ou $(latest(F, \bar{s}, T)$ *e* $lostCallback(\bar{s}, T)$ *e*
	$inCache(F))))$
	ou $(\bar{s} = \emptyset$ *e* $inCache(F))$
após uma operação de *abertura* malsucedida:	$(\bar{s} \neq \emptyset$ *e* $conflict(F, \bar{s}))$
	ou $(\bar{s} = \emptyset$ *e* $\neg\, inCache(F))$
após uma operação de *fechamento* bem-sucedida:	$(\bar{s} \neq \emptyset$ *e* $updated(F, \bar{s}))$
	ou $(\bar{s} = \emptyset)$
após uma operação de *fechamento* malsucedida:	$\bar{s} \neq \emptyset$ *e* $conflict(F, \bar{s})$

Esse modelo presume um sistema síncrono: T é o tempo mais longo durante o qual um cliente pode permanecer sem saber de uma atualização em qualquer parte de um arquivo que está em sua cache; $latest(F, \bar{s}, T)$ denota o fato de que o valor corrente de F, em C, era o mais recente em todos os servidores em $\bar{s}$, em algum instante nos últimos T segundos, e que não havia nenhum conflito entre as cópias de F nesse instante; $lostCallback(\bar{s}\ T)$ significa que um *callback* foi enviado por algum membro de $\bar{s}$ nos últimos T segundos e que não foi recebido em C; e $conflict(F, \bar{s})$ significa que os valores de F em alguns servidores em $\bar{s}$ estão correntemente em conflito.

Acessando réplicas ◊ A estratégia usada nas operações de *abertura* e *fechamento* para acessar as réplicas de um arquivo é uma variante da estratégia *um lê/todos escrevem* (*read-one/write-all*). Na *abertura*, se uma cópia do arquivo não estiver presente na cache local, o cliente identificará um servidor preferido do AVSG do arquivo. O servidor preferido pode ser escolhido aleatoriamente, ou de acordo com critérios de desempenho, como a proximidade física ou a carga no servidor. O cliente solicita uma cópia dos atributos e do conteúdo do arquivo do servidor preferido e, ao recebê-la, consulta todos os outros membros do AVSG para verificar se a cópia é a versão mais recente disponível. Se não for, um membro do AVSG com a versão mais recente se torna o site preferido, o conteúdo do arquivo é novamente buscado e os membros do AVSG são notificados de que alguns membros possuem réplicas antigas. Quando a busca termina, uma promessa de *callback* é estabelecida no servidor preferido.

Quando um arquivo é fechado em um cliente, após a modificação, seu conteúdo e seus atributos são transmitidos em paralelo para todos os membros do AVSG, usando um protocolo de chamada remota de procedimento por *multicast*. Isso maximiza a probabilidade de que todo site de replicação de um arquivo tenha a versão corrente o tempo todo. Não há garantia disso, pois o AVSG não inclui necessariamente todos os membros do VSG. Esse processo minimiza a carga do servidor, passando aos clientes a responsabilidade de propagar as alterações nos sites de replicação no caso normal (os servidores são envolvidos apenas quando uma réplica antiga é descoberta na operação de *abertura*).

Como seria dispendioso manter o estado de *callback* em todos os membros de um AVSG, a promessa de *callback* é mantida apenas no servidor preferido. Mas isso introduz um novo problema: o

servidor preferido de um cliente não precisa estar no AVSG de outro cliente. Se esse for o caso, uma atualização feita pelo segundo cliente não causará um *callback* para o primeiro cliente. A solução adotada para esse problema será discutida na próxima subseção.

Coerência de cache ◊ As garantias de vigência do Coda mencionadas anteriormente significam que o processo Venus em cada cliente deve detectar os seguintes eventos dentro de T segundos a partir de sua ocorrência:

- ampliação de um AVSG (devido à acessibilidade de um servidor anteriormente inacessível);
- redução de um AVSG (devido ao fato de um servidor se tornar inacessível);
- um evento de *callback* perdido.

Para conseguir isso, a cada T segundos, o processo Venus envia uma mensagem de verificação (*probe*) para todos os servidores nos VSGs dos arquivos que possui em sua cache. Serão recebidas respostas apenas dos servidores acessíveis. Se o processo Venus receber uma resposta de um servidor anteriormente inacessível, ele ampliará o AVSG correspondente e eliminará do volume relevante as promessas de *callback* em todos os arquivos que as contém. Isso é feito porque a cópia armazenada na cache pode não ser mais a versão mais recente disponível no novo AVSG.

Se deixar de receber uma resposta de um servidor anteriormente acessível, o processo Venus reduzirá o AVSG correspondente. Nenhuma alteração de *callback* é exigida, a não ser que a diminuição seja causada pela perda de um servidor preferido, no caso em que todas as promessas de *callback* desse servidor deverão ser eliminadas. Se uma resposta indicar que uma mensagem de *callback* foi enviada, mas não recebida, a promessa de *callback* no arquivo correspondente será eliminada.

Agora, ficamos com o problema mencionado anteriormente, das atualizações perdidas por um servidor por que ele não está no AVSG de um cliente diferente que realiza uma atualização. Para tratar desse caso, o processo Venus recebe um *vetor de versão de volume* (*CVV de volume*) em resposta a cada mensagem de verificação. O CVV de volume contém um resumo dos CVVs de todos os arquivos presentes no volume. Se o processo Venus detecta qualquer diferença entre os CVVs de volume, então alguns membros do AVSG devem ter algumas versões de arquivo que não estão atualizadas. Embora os arquivos desatualizados possam não ser aqueles que estão em sua cache local, o processo Venus faz uma suposição pessimista e elimina do volume relevante as promessas de *callback* em todos os arquivos que as contém.

Note que o processo Venus só verifica os servidores nos VSGs dos arquivos para os quais contém cópias na cache e que uma única mensagem de verificação serve para atualizar os AVSGs e verificar os *callbacks* de todos os arquivos de um volume. Isso, combinado com um valor relativamente grande de T (na ordem de 10 minutos na implementação experimental), significa que as verificações não são um obstáculo para a escalabilidade do Coda para grandes quantidades de servidores e de redes de longa distância.

Operação desconectada ◊ Durante as desconexões breves, como aquelas que podem ocorrer devido a interrupções inesperadas do serviço, a política de substituição da cache usada menos recentemente, normalmente adotada pelo processo Venus, pode ser suficiente para evitar perdas na cache nos volumes desconectados. Mas é improvável que um cliente possa operar no modo desconectado por longos períodos, sem gerar referências para arquivos ou diretórios que não estão na cache, a não ser que uma política diferente seja adotada.

Portanto, o Coda permite que os usuários especifiquem uma lista de prioridade dos arquivos e diretórios que o processo Venus deve se esforçar para manter na cache. Os objetos que estão no nível mais alto são identificados como *aderentes* e devem ser mantidos na cache o tempo todo. Se o disco local for grande o suficiente para acomodar todos eles, o usuário será assegurado de que eles permanecerão acessíveis. Como freqüentemente é difícil saber exatamente quais acessos de arquivo são gerados por qualquer seqüência de ações do usuário, é fornecida uma ferramenta que permite ao usuário definir uma seqüência de ações; o processo Venus nota as referências de arquivo geradas pela seqüência e as marca com a prioridade dada.

Quando a operação desconectada termina, começa um processo de *reintegração*. Para cada arquivo ou diretório em cache que foi modificado, criado ou excluído durante a operação desconectada, o

processo Venus executa uma seqüência de operações de atualização para tornar as réplicas do AVSG idênticas à cópia armazenada em cache. A reintegração ocorre de cima para baixo, a partir da raiz de cada volume colocado na cache.

Podem ser detectados conflitos durante a reintegração, resultantes das atualizações nas réplicas do AVSG feitas por outros clientes. Quando isso ocorre, a cópia armazenada na cache é posta em um local temporário no servidor e o usuário que iniciou a reintegração é informado. Essa estratégia é baseada na filosofia de projeto adotada no Coda, que atribui prioridade maior às réplicas baseadas no servidor do que às cópias colocadas na cache. As cópias temporárias são armazenadas em um *co-volume*, que é associado a cada volume em um servidor. Os co-volumes são parecidos com os diretórios *lost+found* encontrados nos sistemas UNIX convencionais. Eles espelham apenas as partes da estrutura de diretório de arquivos necessárias para conter os dados temporários. Pouco armazenamento adicional é exigido, pois os co-volumes são quase vazios.

Desempenho ◊ Satyanarayanan *et al.* [1990] comparam o desempenho do Coda com o AFS com *benchmarks* projetados para simular populações de usuários variando de cinco a 50 usuários típicos do AFS.

Sem nenhuma replicação, há pouca diferença significativa entre o desempenho do AFS e do Coda. Com replicação tripla, o tempo para o Coda executar um *benchmark* que simula a carga equivalente a 5 usuários típicos ultrapassou a do AFS sem duplicação em apenas 5%. Entretanto, com replicação tripla e uma carga equivalente a 50 usuários, o tempo para o *benchmark* aumentou em 70%, enquanto o do AFS sem replicação aumentou em apenas 16%. Essa diferença é atribuída apenas em parte às sobrecargas associadas à replicação – as diferenças na otimização da implementação são responsáveis por parte da diferença no desempenho.

Discussão ◊ Mencionamos anteriormente que o Coda é semelhante ao Bayou porque também emprega uma estratégia otimista para obter alta disponibilidade (embora eles sejam diferentes em vários outros aspectos e não apenas porque um gerencia arquivos e o outro bancos de dados). Também descrevemos o modo como o Coda usa CVVs para verificar a existência de conflitos, sem considerar a semântica dos dados armazenados nos arquivos. A estratégia pode detectar conflitos de escrita-escrita em potencial, mas não conflitos de leitura-escrita. Existe o potencial de conflitos de escrita-escrita porque, no nível da semântica da aplicação, não pode haver nenhum conflito real: os clientes podem ter atualizado compativelmente objetos diferentes do arquivo e uma simples integração automática seria possível.

A estratégia global do Coda de detecção de conflito livre de semântica e solução manual é sensata em muitos casos, especialmente em aplicações que exigem julgamento humano ou em sistemas sem conhecimento da semântica dos dados.

Os diretórios são um caso especial do Coda. Às vezes é possível manter automaticamente a integridade desses objetos importantes por meio da solução de conflitos, pois sua semântica é relativamente simples: as únicas alterações que podem ser feitas nos diretórios são a inserção ou a exclusão de entradas de diretório. O Coda incorpora seu próprio método para solucionar diretórios. Ele tem o mesmo efeito da estratégia de transformação operacional do Bayou, mas o Coda integra diretamente o estado de diretórios conflitantes, pois não tem nenhum registro das operações executadas pelos clientes.

15.5 Transações em dados replicados

Até aqui, neste capítulo, consideramos sistemas nos quais os clientes solicitam uma única operação por vez em conjuntos de objetos replicados. Os Capítulos 13 e 14 explicaram que as transações são *seqüências* de uma ou mais operações, aplicadas de maneira a impor as propriedades ACID. Assim como acontece nos sistemas da Seção 15.4, nos sistemas transacionais os objetos podem ser replicados para aumentar a disponibilidade e o desempenho.

Do ponto de vista do cliente, uma transação sobre objetos replicados deve ser idêntica à dos objetos não replicados. Em um sistema não replicado, as transações parecem ser executadas uma por

vez, em alguma ordem. Isso é conseguido garantindo-se uma interposição equivalente em série das transações dos clientes. O efeito das transações executadas pelos clientes sobre objetos replicados deve ser igual ao que se eles tivessem efetuado uma por vez, sobre um único conjunto de objetos. Essa propriedade é chamada de *capacidade de serialização de uma cópia*. Ela é semelhante à consistência seqüencial, mas não deve ser confundida com esta. A consistência seqüencial considera execuções válidas, sem nenhuma noção de agregação das operações do cliente nas transações.

Cada gerenciador de réplica fornece controle de concorrência e recuperação de seus próprios objetos. Nesta seção, presumimos que o bloqueio de duas fases é usado para controle de concorrência.

A recuperação é complicada pelo fato de que um gerenciador de réplica falho é membro de um conjunto, e que os outros membros continuam a fornecer um serviço durante o tempo em que ele está indisponível. Quando um gerenciador de réplica se recupera de uma falha, ele usa informações obtidas dos outros gerenciadores de réplica para restaurar seus objetos com seus valores correntes, levando em conta todas as alterações ocorridas durante o tempo em que estava indisponível.

Esta seção apresenta primeiro a arquitetura das transações com dados replicados. As principais questões são: se a requisição de um cliente pode ser tratada em qualquer um dos gerenciadores de réplica; como muitos gerenciadores de réplica são exigidos para a conclusão bem-sucedida de uma operação; se o gerenciador de réplica contatado por um cliente pode adiar o encaminhamento das requisições até que uma transação seja efetivada; e como executar um protocolo de efetivação de duas fases.

A implementação da capacidade de serialização de uma cópia é ilustrada pela estratégia *um lê/todos escrevem* – um esquema de replicação simples no qual as operações *de leitura* são executadas por um único gerenciador de réplica e as operações *de escrita* são realizadas por todos.

Em seguida, a seção discutirá os problemas da implementação de esquemas de replicação na presença de falhas e recuperação de servidores. Será apresentada a replicação de cópias disponíveis – uma variante do esquema de replicação um lê/todos escrevem, na qual as operações *de leitura* são realizadas por um único gerenciador de réplica e as operações *de escrita* são efetuadas por todos aqueles que estiverem disponíveis.

Finalmente, a seção apresentará três esquemas de replicação que funcionam corretamente quando o conjunto de gerenciadores de réplica é dividido em subgrupos por um particionamento da rede:

- *Cópias disponíveis com validação*: a replicação de cópias disponíveis é aplicada em cada partição e quando o particionamento for reparado, é aplicado um procedimento de validação e todas as inconsistências são tratadas.
- *Consenso de quorum*: um subgrupo deve ter *quorum* (significando que ele tem membros suficientes) para poder continuar fornecendo um serviço na ocorrência de um particionamento. Quando um particionamento é reparado (e quando um gerenciador de réplica é reiniciado, após uma falha), os gerenciadores de réplica atualizam seus objetos por intermédio de procedimentos de recuperação.
- *Partição virtual*: uma combinação do consenso de *quorum* e das cópias disponíveis. Se uma partição virtual tem *quorum*, ele pode usar replicação de cópias disponíveis.

15.5.1 Arquiteturas para transações replicadas

Assim como na variedade de sistemas que já consideramos nas seções anteriores, um *front end* pode difundir as requisições do cliente através de *multicast* para os grupos de gerenciadores de réplica ou pode enviar cada requisição para um único gerenciador de réplica, o qual fica então responsável por processar a requisição e responder para o cliente. Wiesmann *et al.* [2000] e Schiper e Raynal [1996] consideram o caso do *multicast* das requisições e não vamos tratar disso aqui. Daqui por diante, supomos que um *front end* envia requisições do cliente para um dos gerenciadores do grupo de gerenciadores de réplica de um objeto lógico. Na estratégia da *cópia primária*, todos os *front ends* se comunicam com um gerenciador de réplica primário diferenciado, para executar uma operação, e esse gerenciador de réplica mantém os *backups* atualizados. Como alternativa, os *front ends* podem se comunicar com qualquer gerenciador de réplica para executar uma operação – mas a coordenação entre os gerenciadores de réplica é, conseqüentemente, mais complexa.

O gerenciador de réplica que recebe uma requisição para executar uma operação sobre um objeto em particular fica responsável por obter a cooperação dos outros gerenciadores de réplica do grupo que tenham cópias desse objeto. Diferentes esquemas de replicação têm diferentes regras a respeito de quantos gerenciadores de réplica de um grupo são exigidos para a conclusão bem-sucedida de uma operação. Por exemplo, no esquema um lê/todos escrevem, um pedido *de leitura* pode ser executado por um único gerenciador de réplica, enquanto um pedido *de escrita* deve ser executado por todos os gerenciadores de réplica do grupo, como mostrado na Figura 15.10 (pode haver diferentes números de réplicas dos vários objetos). Os esquemas de consenso de *quorum* são projetados para reduzir o número de gerenciadores de réplica que devem efetuar operações de atualização, mas ao custo de um maior número de gerenciadores de réplica para executar operações *somente de leitura*.

Outra questão é se o gerenciador de réplica contatado por um *front end* deve adiar o encaminhamento das requisições de atualização para outros gerenciadores de réplica do grupo, até que uma transação seja efetivada – a assim chamada estratégia *preguiçosa* de propagação de atualização – ou, inversamente, se os gerenciadores de réplica devem encaminhar cada requisição de atualização para todos os gerenciadores de réplica necessários dentro da transação e antes de efetivar – a estratégia *ávida*. A estratégia "preguiçosa" é uma alternativa atraente, pois reduz a quantidade de comunicação entre os gerenciadores de réplica que ocorre antes da resposta ao cliente que está fazendo a atualização. Entretanto, o controle de concorrência também deve ser considerado. Às vezes, a estratégia "preguiçosa" é usada na replicação de cópia primária (veja a seguir), onde um único gerenciador de réplica primário dispõe as transações em série. Mas se várias transações diferentes puderem tentar acessar os mesmos objetos em gerenciadores de réplica diferentes de um grupo, então, para garantir que as transações sejam dispostas em série corretamente em todos os gerenciadores de réplica do grupo, cada gerenciador de réplica precisa saber a respeito das requisições feitas pelos outros. A estratégia "ávida" é a única disponível, nesse caso.

O protocolo de efetivação de duas fases ◊ O protocolo de efetivação de duas fases se torna um protocolo de efetivação de duas fases com dois níveis de aninhamento. Como antes, o coordenador de uma transação se comunica com os operários. Mas se o coordenador, ou um operário, for um gerenciador de réplica, ele se comunicará com os outros gerenciadores de réplica para os quais passou requisições durante a transação.

Isto é, na primeira fase, o coordenador envia a requisição de *canCommit* para os operários, os quais a passam para os outros gerenciadores de réplica e reúnem suas respostas, antes de responder para o coordenador. Na segunda fase, o coordenador envia a requisição de *doCommit* ou *doAbort*, a qual é passada para os membros dos grupos de gerenciadores de réplica.

Replicação da cópia primária ◊ A replicação da cópia primária pode ser usada no contexto das transações. Nesse esquema, todas as requisições de cliente (sejam elas somente de leitura ou não) são direcionadas para um único gerenciador de réplica primário (veja a Figura 15.4). Para a replicação

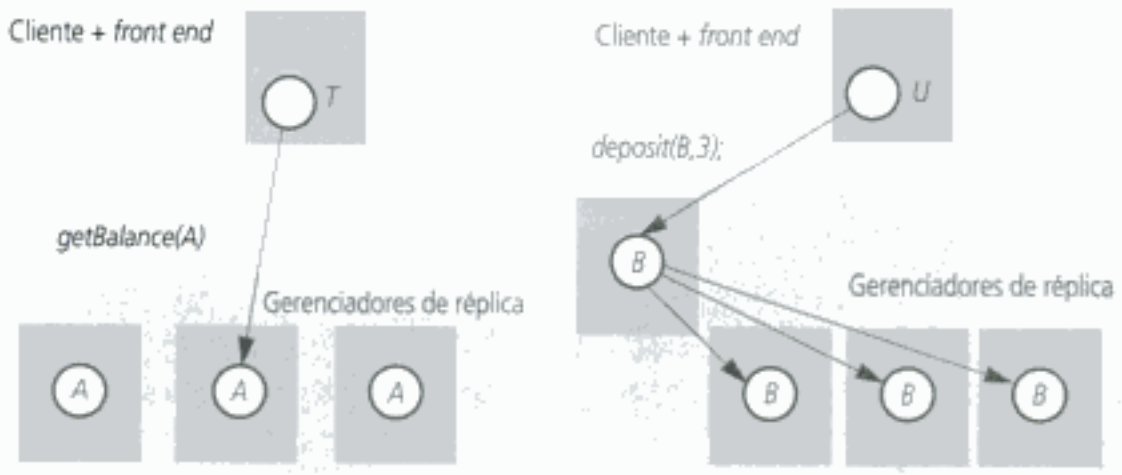


Figura 15.10 Transações sobre dados replicados.

da cópia primária, o controle de concorrência é aplicado no gerenciador primário. Para efetivar uma transação, o gerenciador primário se comunica com os gerenciadores de réplica de *backup* e, então, na estratégia "ávida", responde para o cliente. Essa forma de replicação permite que um gerenciador de réplica de *backup* assuma o controle consistentemente, caso o gerenciador primário venha a falhar. Na alternativa "preguiçosa", o gerenciador primário responde para os *front ends* antes de ter atualizado seus *backups*. Nesse caso, um gerenciador de *backup* que venha a substituir um *front end* falho não terá necessariamente o estado mais recente do banco de dados.

Um lê/todos escrevem (*read-one/write-all*) ◊ Usamos esse esquema de replicação simples para ilustrar como o bloqueio de duas fases em cada gerenciador de réplica pode ser usado para obter capacidade de serialização de uma cópia, onde os *front ends* podem se comunicar com qualquer gerenciador de réplica. Toda operação *de escrita* deve ser executada em todos os gerenciadores de réplica, cada um dos quais configura uma trava de escrita sobre o objeto afetado pela operação. Cada operação *de leitura* é executada por um único gerenciador de réplica, o qual configura uma trava de leitura sobre o objeto afetado pela operação.

Considere pares de operações de diferentes transações sobre o mesmo objeto: qualquer par de operações *de escrita* exigirá travas conflitantes em todos os gerenciadores de réplica; uma operação *de leitura* e uma operação *de escrita* exigirão travas conflitantes em um único gerenciador de réplica. Assim, a capacidade de serialização de uma cópia é obtida.

15.5.2 Replicação de cópias disponíveis

A replicação um lê/todos escrevem simples não é um esquema realista, pois não pode ser realizado se alguns dos gerenciadores de réplica estiverem indisponíveis, ou porque falharam, ou devido a uma falha de comunicação. O esquema das cópias disponíveis é projetado para possibilitar que alguns gerenciadores de réplica estejam temporariamente indisponíveis. A estratégia usa o fato de que a requisição *de leitura* de um cliente sobre um objeto lógico pode ser executada por qualquer gerenciador de réplica disponível, mas a requisição de atualização de um cliente deve ser executada por todos os gerenciadores de réplica disponíveis no grupo que contêm cópias do objeto. A idéia de 'membros disponíveis de um grupo de gerenciadores de réplica' é semelhante ao grupo de armazenamento de volume disponível do Coda, descrito na Seção 15.4.3.

No caso normal, as requisições do cliente são recebidas e executadas por um gerenciador de réplica que esteja funcionando. As requisições *de leitura* podem ser executadas pelo gerenciador de réplica que as receber. As requisições *de escrita* são executadas pelo gerenciador de réplica receptor e por todos os outros gerenciadores de réplica disponíveis no grupo. Por exemplo, na Figura 15.11, a operação *getBalance* da transação *T* é efetuada por *X*, enquanto sua operação *deposit* é executada por *M*, *N* e *P*. O controle de concorrência em cada gerenciador de réplica afeta as operações executadas

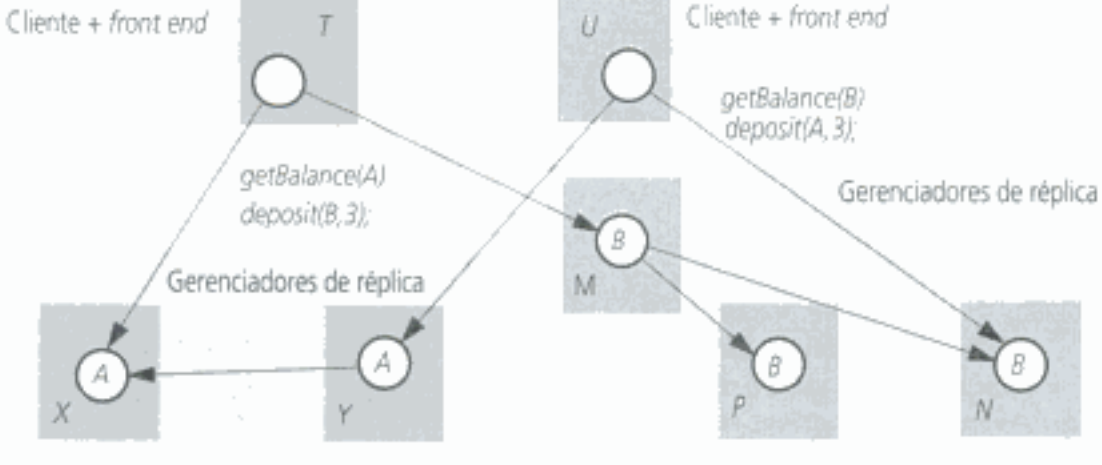


Figura 15.11 Cópias disponíveis.

de forma local. Por exemplo, em X, a transação T leu A e, portanto, a transação U não pode atualizar A com a operação *deposit* até que a transação T tenha terminado. Desde que o conjunto de gerenciadores de réplica disponíveis não mude, o controle de concorrência local obtém a capacidade de serialização de uma cópia da mesma maneira que na replicação um lê/todos escrevem. Infelizmente, isso não acontece se um gerenciador de réplica falha ou se recupera durante o andamento das transações conflitantes.

Falha do gerenciador de réplica ◊ Supomos que os gerenciadores de réplica falham benignamente por colapso. Entretanto, um gerenciador de réplica que apresenta falha por colapso é substituído por um novo processo, o qual recupera o estado efetivado dos objetos a partir de um arquivo de recuperação. Os *front ends* usam tempos limites para decidir se um gerenciador de réplica não está correntemente disponível. Quando um cliente faz uma requisição para um gerenciador de réplica que falhou, o *front end* atinge o tempo limite e tenta novamente, em outro gerenciador de réplica do grupo. Se a requisição for recebida por um gerenciador de réplica no qual o objeto está desatualizado, porque o gerenciador não se recuperou completamente da falha, ele rejeita a requisição e o *front end* tenta novamente em outro gerenciador de réplica do grupo.

A serialização de uma cópia exige que as falhas e recuperações sejam dispostas em série com relação às transações. De acordo com o fato de poder acessar um objeto ou não, uma transação observa se uma falha ocorre depois de ter terminado ou antes de ter começado. A serialização de uma cópia não é obtida quando transações diferentes fazem observações de falha conflitantes.

Considere o caso da Figura 15.11, onde o gerenciador de réplica X falha imediatamente após T ter executado *getBalance* e o gerenciador de réplica N falha imediatamente após U ter executado *getBalance*. Suponha que esses dois gerenciadores de réplica falhem antes de T e U terem executado suas operações *deposit*. Isso implica que a operação *deposit* de T será executada nos gerenciadores de réplica M e P, e que a operação *deposit* de U será executada no gerenciador de réplica Y. Infelizmente, o controle de concorrência de A no gerenciador de réplica X não impede que a transação U atualize A no gerenciador de réplica Y. Nem o controle de concorrência de B no gerenciador de réplica N impede que a transação T atualize B nos gerenciadores de réplica M e P.

Isso se contrapõe ao requisito da capacidade de serialização de uma cópia. Se essas operações fossem executadas em cópias únicas dos objetos, elas seriam dispostas em série, com a transação T antes de U, ou com a transação U antes de T. Isso garante que uma das transações lerá o valor configurado pela outra. O controle de concorrência local sobre cópias de objetos não é suficiente para garantir a capacidade de serialização de uma cópia no esquema de replicação de cópias disponíveis.

Quando as operações *de escrita* são direcionadas para todas as cópias disponíveis, o controle de concorrência local garante que as gravações conflitantes em um objeto sejam dispostas em série. Em contraste, uma operação *de leitura* de uma transação e uma operação *de escrita* de outra não afetam necessariamente a mesma cópia de um objeto. Portanto, o esquema exige controle de concorrência adicional para evitar que as dependências entre uma operação *de leitura* de uma transação e uma operação *de escrita* de outra transação formem um ciclo. Tais dependências não podem surgir se as falhas e recuperações de réplicas de objetos estiverem dispostas em série com relação às transações.

Validação local ◊ Nos referimos ao procedimento de controle de concorrência adicional como validação local. O procedimento de validação local é projetado para garantir que qualquer evento de falha ou recuperação não se manifeste durante o andamento de uma transação. Em nosso exemplo, como T leu de um objeto em X, a falha de X deve ser após T. Analogamente, como T observa a falha de N, quando ela tenta atualizar o objeto, a falha de N deve ser antes de T. Ou seja:

N falha $\rightarrow$ T lê o objeto A em X; T escreve o objeto B em M e $P \rightarrow T$ é efetivada $\rightarrow$ X falha

Também pode ser argumentado, para a transação U, que:

X falha $\rightarrow$ U lê o objeto B em N; U escreve o objeto A em $Y \rightarrow U$ é efetivada $\rightarrow$ N falha

O procedimento de validação local garante que não podem ocorrer duas dessas seqüências incompatíveis. Antes que uma transação seja efetivada, ela verifica a existência de falhas (e recuperações) de gerenciadores de réplica dos objetos que acessou. No exemplo, a transação T verificaria se N ainda está indisponível e se X, M e P ainda estão disponíveis. Se esse fosse o caso, T poderia ser efetivada.

Isso implica que *X* falha após *T* ser validada e antes de *U* ser validada. Em outras palavras, a validação de *U* ocorre após a validação de *T*. A validação de *U* falha porque *N* já falhou.

Quando uma transação tiver observado uma falha, o procedimento de validação local tentará se comunicar com os gerenciadores de réplica falhos para garantir que eles ainda não se recuperaram. A outra parte da validação local, que é testar se os gerenciadores de réplica não falharam desde que os objetos foram acessados, pode ser combinada com o protocolo de efetivação de duas fases.

Os algoritmos de cópias disponíveis não podem ser usados em ambientes nos quais gerenciadores de réplica que estão funcionando não conseguem se comunicar uns com os outros.

15.5.3 Particionamento da rede

Os esquemas de replicação precisam levar em conta a possibilidade de particionamento da rede. Um particionamento da rede separa um grupo de gerenciadores de réplica em dois ou mais subgrupos, de maneira tal que os membros de um subgrupo podem se comunicar uns com os outros, mas os membros de subgrupos diferentes não podem se comunicar. Por exemplo, na Figura 15.12, os gerenciadores de réplica que estão recebendo a requisição *deposit* não podem enviá-la para os gerenciadores de réplica que estão recebendo a requisição *withdraw*.

Os esquemas de replicação são projetados segundo a suposição de que o particionamento será reparado em algum momento. Portanto, os gerenciadores de réplica dentro de uma única partição devem garantir que as requisições que executarem durante o particionamento não tornarão o conjunto de réplicas inconsistente quando o mesmo for reparado.

Davidson *et al.* [1985] discutem muitas estratégias diferentes, as quais classificam como otimistas ou pessimistas com relação à probabilidade de ocorrência de inconsistências. Os esquemas otimistas não limitam a disponibilidade durante um particionamento, enquanto os esquemas pessimistas o fazem.

As estratégias otimistas permitem atualizações em todas as partições – isso pode levar a inconsistências entre as partições, as quais deverão ser resolvidas quando o particionamento for reparado. Um exemplo dessa estratégia é uma variante do algoritmo de cópias disponíveis na qual são permitidas atualizações nas partições e, após o particionamento ter sido reparado, as atualizações são validadas – as atualizações que violam o critério da capacidade de serialização de uma cópia são canceladas.

A estratégia pessimista limita a disponibilidade, mesmo quando não existem particionamentos, mas impede a ocorrência de inconsistências durante o particionamento. Quando um particionamento é reparado, basta atualizar as cópias dos objetos. A estratégia do consenso de *quorum* é pessimista. Ela permite atualizações em uma partição que tenha a maioria dos gerenciadores de réplica e propaga as atualizações para os outros gerenciadores de réplica quando o particionamento é reparado.

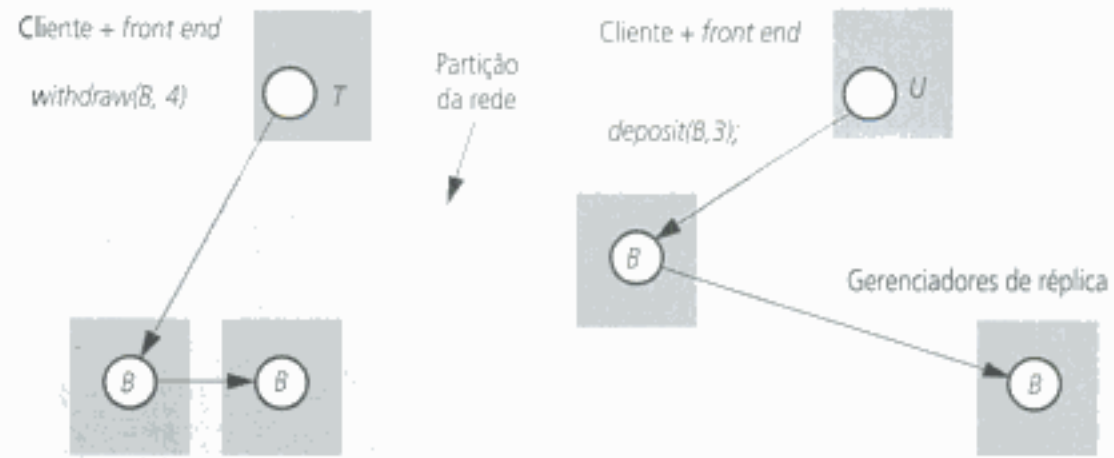


Figura 15.12 Particionamento da rede.

15.5.4 Cópias disponíveis com validação

O algoritmo de cópias disponíveis é aplicado dentro de cada partição. Essa estratégia otimista mantém o nível normal de disponibilidade para operações *de leitura*, mesmo durante um particionamento. Quando um particionamento é reparado, as transações possivelmente conflitantes que ocorreram nas partições separadas são validadas. Se a validação falhar, alguns passos devem ser dados para superar as inconsistências. Se não houvesse particionamento, uma de duas transações com operações conflitantes teria sido retardada ou cancelada. Infelizmente, como houve o particionamento, os pares de transações conflitantes puderam ser efetivados nas diferentes partições. A única opção após o evento é cancelar uma delas. Isso exige fazer alterações nos objetos e, em alguns casos, compensar os efeitos no mundo real, como tratar com contas bancárias sem fundo. A estratégia otimista só é possível em aplicações onde tais ações de compensação podem ser executadas.

Vetores de versão podem ser usados para validar conflitos entre pares de operações *de escrita*. Eles são usados no sistema de arquivos Coda e foram descritos na Seção 15.4.3. Essa estratégia não consegue detectar conflitos de leitura-escrita, mas funciona bem nos sistemas de arquivos onde as transações tendem a acessar um único arquivo e os conflitos de leitura-escrita não são importantes. Ela não é conveniente para aplicações como nosso exemplo de transações bancárias, onde os conflitos de leitura-escrita são importantes.

Davidson [1984] usou *grafos de precedência* para detectar inconsistências entre partições. Cada divisão mantém um *log* dos objetos afetados pelas operações *de leitura* e *de escrita* das transações. Esse *log* é usado para construir um grafo de precedência cujos nós são transações e cujas setas representam conflitos entre as operações *de leitura* e *de escrita* das transações. Tal grafo não conterá ciclos, pois foi aplicado controle de concorrência dentro da partição. O procedimento de validação pega os grafos de precedência das partições e adiciona setas, representando conflitos, entre transações nas diferentes partições. Se o grafo resultante contiver ciclos, a validação falhará.

15.5.5 Métodos de consenso de quorum

Uma maneira de evitar que transações em diferentes partições produzam resultados inconsistentes é estabelecer a regra de que as operações só podem ser executadas dentro de uma das partições. Como os gerenciadores de réplica nas diferentes partições não podem se comunicar, o subgrupo de gerenciadores de réplica dentro de cada partição deve ser capaz de decidir independentemente se podem executar as operações. Um *quorum* é um subgrupo de gerenciadores de réplica cujo tamanho proporciona a ele o direito de executar operações. Por exemplo, se o critério fosse ter a maioria, um subgrupo que tivesse a maioria dos membros de um grupo formaria um *quorum*, pois nenhum outro subgrupo poderia ter a maioria.

Nos esquemas de replicação por consenso de *quorum*, uma operação de atualização sobre um objeto lógico pode ser concluída com êxito por um subgrupo de seu grupo de gerenciadores de réplica. Portanto, os outros membros do grupo terão cópias desatualizadas do objeto. Para determinar se as cópias estão atualizadas, podem ser usados números de versão ou indicações de tempo. Se forem usadas versões, o estado inicial de um objeto é a primeira versão e, após cada alteração, temos uma nova versão. Cada cópia de um objeto tem um número de versão, mas apenas as cópias atualizadas têm o número de versão corrente, enquanto as cópias desatualizadas têm números de versão anteriores. As operações devem ser aplicadas apenas nas cópias com o número de versão corrente.

Gifford [1979] desenvolveu um esquema de replicação de arquivos no qual um número de 'votos' é atribuído a cada cópia física de um único arquivo lógico em um gerenciador de réplica. Um voto pode ser considerado como um peso relacionado ao desejo de usar uma cópia em particular. Cada operação *de leitura* deve primeiro obter *quorum* de leitura de R votos, antes de poder ler qualquer cópia atualizada, e cada operação *de escrita* deve obter *quorum* de gravação de W votos, antes de poder prosseguir com uma operação de atualização. R e W são configurados para um grupo de gerenciadores de réplica de modo que

$W >$ metade do total de votos

$R + W >$ número total de votos do grupo

Hidden page

Hidden page

Hidden page

Figura 15.14 Partição virtual.

modo que Y não pode se comunicar com Z, mas os dois grupos V, X, Y e V, X, Z podem se comunicar entre si. Então, existe o perigo da criação de duas partições virtuais sobrepostas, como V_1 e V_2 mostradas na Figura 15.15.

Considere o efeito de executar transações diferentes nas duas partições virtuais. A operação *de leitura* da transação em V, X, Y poderia ser aplicada no gerenciador de réplica Y, no caso em que sua trava de leitura não entraria em conflito com as travas de escritas configuradas por uma operação *de escrita* e uma transação na outra partição virtual. As partições virtuais sobrepostas são contrárias à capacidade de serialização de uma cópia.

O objetivo do protocolo é criar novas partições virtuais consistentemente, mesmo que ocorram partições reais durante o processo. O protocolo para criar uma nova partição virtual tem duas fases, como mostra a Figura 15.16.

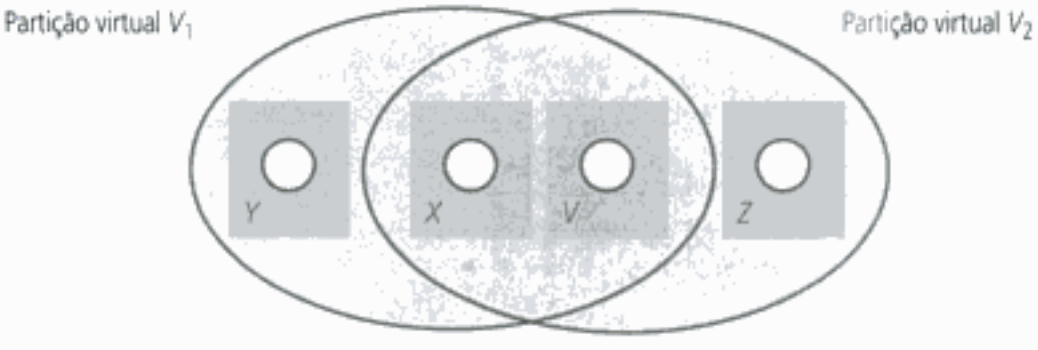


Figura 15.15 Duas partições virtuais sobrepostas.

Fase 1:

- O iniciador envia uma requisição de *Join* para cada membro em potencial. O argumento de *Join* é uma indicação de tempo lógica proposta para a nova partição virtual.
- Quando um gerenciador de réplica recebe uma requisição de *Join*, ele compara a indicação de tempo lógico proposta com a de sua partição virtual corrente.
  - Se a indicação de tempo lógica proposta for maior, ele concorda em juntar e responde *Sim*;
  - Se for menor, ele recusa a se juntar e responde *Não*.

Fase 2:

- Se o iniciador tiver recebido respostas *Sim* suficientes para ter *quora de leitura* e *de escrita*, ele pode completar a criação da nova partição virtual, enviando uma mensagem de *Confirmation* para os sites que concordaram em se juntar. A indicação de tempo da criação e a lista de membros reais são enviadas como argumentos.
- Os gerenciadores de réplica que receberam a mensagem de *Confirmation* entram na nova partição virtual e gravam sua indicação de tempo de criação e sua lista de membros reais.

Figura 15.16 Criando uma partição virtual.

Um gerenciador de réplica que responde *Sim* na Fase 1 não pertence a uma partição virtual até receber a mensagem de *Confirmation* correspondente na Fase 2.

Em nosso exemplo anterior, os gerenciadores de réplica *Y* e *Z* mostrados na Figura 15.13 tentam cada um criar uma partição virtual e qual tiver a indicação de tempo lógica mais alta será usado no final.

Esse é um método eficiente quando o particionamento não é uma ocorrência comum. Cada transação usa o algoritmo de cópias disponíveis dentro de uma partição virtual.

15.6 Resumo

A replicação de objetos é uma maneira importante de obter serviços com bom desempenho, alta disponibilidade e tolerância a falhas em um sistema distribuído. Descrevemos arquiteturas para serviços nas quais os gerenciadores de réplica contêm réplicas dos objetos e nas quais os *front ends* tornam essa replicação transparente. Os clientes, os *front ends* e os gerenciadores de réplica podem ser processos separados ou existir no mesmo espaço de endereçamento.

Este capítulo começou descrevendo um modelo de sistema no qual cada objeto lógico é implementado por um conjunto de réplicas físicas. Freqüentemente, as atualizações nessas réplicas podem ser feitas convenientemente pela comunicação em grupo. Expandimos nossa narrativa sobre a comunicação em grupo para incluir serviços de participação como membro do grupo e comunicação de modo de visualização síncrono.

Definimos a capacidade de linearização e a consistência seqüencial como critérios da correção para serviços tolerantes a falhas. Esses critérios expressam como os serviços devem fornecer o equivalente a uma única imagem do conjunto de objetos lógicos, mesmo que esses objetos sejam replicados. Praticamente falando, o mais significativo dos critérios é a consistência seqüencial.

Na replicação passiva (*backup* primário), a tolerância a falhas é obtida pelo direcionamento de todas as requisições por intermédio de um gerenciador de réplica distinto e fazendo-se com que um gerenciador de réplica de *backup* assuma, caso o primeiro falhe. Na replicação ativa, todos os gerenciadores de réplica processam todas as requisições independentemente. As duas formas de replicação podem ser convenientemente implementadas usando-se comunicação em grupo.

Em seguida, consideramos os serviços com alta disponibilidade. As arquiteturas de fofoca e Bayou permitem que os clientes façam atualizações em réplicas locais, enquanto estão particionados. Em cada sistema, os gerenciadores de réplica trocam atualizações entre si, quando são reconectados. A arquitetura de fofoca fornece alta disponibilidade à custa de uma consistência causal relaxada. A arquitetura Bayou fornece garantias de consistência eventual mais fortes, empregando detecção automática de conflitos e a técnica da transformação operacional para solucionar conflitos. O Coda é um sistema de arquivos de alta disponibilidade que usa vetores de versão para detectar atualizações potencialmente conflitantes.

Finalmente, consideramos o desempenho das transações com dados replicados. Para esse caso, existem tanto arquiteturas de *backup* primário como arquiteturas nas quais os *front ends* podem se comunicar com qualquer gerenciador de réplica. Discutimos como os sistemas transacionais permitem falhas do gerenciador de réplica e particionamentos da rede. As técnicas de cópias disponíveis, consenso de *quorum* e partições virtuais permitem o progresso das operações dentro das transações, mesmo em algumas circunstâncias nas quais nem todas as réplicas estão acessíveis.

Exercícios

15.1 Três computadores juntos fornecem um serviço replicado. Os fabricantes dizem que cada computador tem um tempo médio entre falhas de cinco dias; normalmente, uma falha demora quatro horas para ser corrigida. Qual é a disponibilidade do serviço replicado? *página 521*

15.2 Explique por que um servidor *multi-threaded* poderia não ser qualificado como uma máquina de estado. *página 523–524*

15.3 Em um jogo multiusuário, os jogadores movem figuras em uma cena comum. O estado do jogo é replicado nas estações de trabalho dos jogadores e em um servidor, o qual contém serviços controlando o jogo global, como a detecção de colisão. As atualizações são enviadas por *multicast* para todas as réplicas.

(i) As figuras podem lançar projéteis umas nas outras e um acerto debilita o infeliz receptor por um tempo limitado. Que tipo de ordem de atualização é exigido aqui? Dica: considere os eventos de disparo, colisão e reanimação.

(ii) O jogo incorpora dispositivos mágicos que podem ser pegos por um jogador para ajudá-lo. Que tipo de ordenação deve ser aplicado na operação "pegar dispositivo"? *página 524–525*

15.4 Um roteador que separa o processo p de dois outros, q e r, falha imediatamente após p iniciar o *multicast* da mensagem m. Se o sistema de comunicação em grupo é de modo de visualização síncrono, explique o que acontece com p em seguida. *página 527–528*

15.5 Você recebe um sistema de comunicação em grupo com uma operação de *multicast* totalmente ordenada e um detector de falha. É possível construir comunicação em grupo com modo de visualização síncrono a partir apenas desses componentes? *página 527–528*

15.6 Uma operação de *multicast* com *ordenação de sincronismo* é aquela cuja semântica de ordem de distribuição é igual a da distribuição de modos de visualização em um sistema de comunicação em grupo com modo de visualização síncrono. Em um serviço *thingumajig*, as operações sobre *thingumajigs* têm ordem causal. O serviço suporta listas de usuários capazes de efetuar operações sobre cada *thingumajig* em particular. Explique por que a remoção de um usuário de uma lista deve ser uma operação com ordenação de sincronismo. *página 527–528*

15.7 Qual é o problema de consistência levantado pela transferência de estado? *página 528–529*

15.8 Uma operação X sobre um objeto o faz com que o invoque uma operação sobre outro objeto o'. Agora, é proposto que ela replique o, mas não o'. Explique a dificuldade que isso acarreta com relação às invocações sobre o' e sugira uma solução. *página 529–530*

15.9 Explique a diferença entre capacidade de linearização e consistência seqüencial, e porque, em geral, é mais prático implementar esta última. *página 531–532*

15.10 Explique por que permitir que gerenciadores de *backup* processem operações de leitura leva a execuções com consistência seqüencial, em vez de execuções com capacidade de linearização em um sistema de replicação passiva. *página 533–534*

15.11 A arquitetura de fofoca poderia ser usada para um jogo de computador distribuído, conforme descrito no Exercício 15.3? *página 536–537*

15.12 Na arquitetura de fofoca, por que um gerenciador de réplica precisa manter uma indicação de tempo da réplica e uma indicação de tempo do 'valor'? *página 539–540*

15.13 Em um sistema de fofoca, um *front end* tem a indicação de tempo vetorial (3, 5, 7), representando os dados que recebeu dos membros de um grupo de três gerenciadores de réplica. Os três gerenciadores de réplica têm as indicações de tempo vetoriais (5, 2, 8), (4, 5, 6) e (4, 5, 8) respectivamente. Qual gerenciador (ou gerenciadores) de réplica poderia atender imediatamente uma consulta do *front end* e qual é a indicação de tempo resultante do *front end*? Qual poderia incorporar uma atualização do *front end* imediatamente? *página 540–541*

15.14 Explique por que tornar alguns gerenciadores de réplica somente de leitura pode melhorar o desempenho de um sistema de fofoca. *página 544*

15.15 Escreva um pseudocódigo para verificações de dependência e procedimentos de integração (conforme os usados no Bayou), convenientes para uma aplicação simples de marcação de quartos. *página 545–546*

15.16 No sistema de arquivos Coda, por que às vezes é necessário que os usuários intervenham manualmente no processo de atualização das cópias de um arquivo em vários servidores? *página 551–552*

15.17 Projete um esquema para integrar duas réplicas de um diretório de sistema de arquivos que passaram por atualizações separadas durante a operação desconectada. Use a estratégia da transformação operacional do Bayou ou forneça uma solução para o Coda. *página 552–553*

15.18 A replicação de cópias disponíveis é aplicada nos itens de dados A e B, com réplicas A_x, A_y e B_w, B_z. As transações T e U são definidas como:

T: *Read*(A); *Write*(B, 44). U: *Read*(B); *Write*(A, 55).

Mostre uma interposição de T e U, supondo que são aplicadas travas de duas fases nas réplicas. Explique por que apenas as travas não podem garantir capacidade de serialização de uma cópia, caso uma das réplicas falhe durante o andamento de T e U. Explique, com referência a este exemplo, como a validação local garante a capacidade de serialização de uma cópia. *página 555*

15.19 A replicação de consenso de *quorum* de Gifford está em uso nos servidores X, Y e Z, todos os quais contêm réplicas dos itens de dados A e B. Os valores iniciais de todas as réplicas de A e B são 100 e os votos de A e B são 1 cada, em X, Y e Z. Além disso, $R = W = 2$ para A e para B. Um cliente lê o valor de A e depois o escreve em B.

(i) No momento em que o cliente executa essas operações, um particionamento separa os servidores X e Y do servidor Z. Descreva os *quora* obtidos e as operações que ocorrerão se o cliente puder acessar os servidores X e Y.

(ii) Descreva os *quora* obtidos e as operações que ocorrerão se o cliente puder acessar apenas o servidor Z.

(iii) O particionamento é reparado e, em seguida, outro particionamento ocorre, de modo que X e Z ficam separados de Y. Descreva os *quora* obtidos e as operações que ocorrerão se o cliente puder acessar os servidores X e Z. *página 559–560*

16 Computação Móvel e Ubíqua

Este capítulo estuda os campos da computação móvel e ubíqua, os quais surgiram devido à miniaturização dos dispositivos e à conectividade sem fio. Falando de modo geral, a computação móvel ocupa-se da exploração da conexão de equipamentos portáteis; a computação ubíqua diz respeito à exploração da integração cada vez maior dos dispositivos de computação com nosso mundo físico cotidiano.

Este capítulo apresenta um modelo de sistema comum que dá ênfase à volatilidade dos sistemas móveis e ubíquos: o conjunto de usuários, dispositivos e componentes de software, em qualquer ambiente, está sujeito a mudar freqüentemente. Em seguida, o capítulo examina algumas das principais áreas de pesquisa que surgiram por causa da volatilidade e de suas bases físicas, incluindo: como os componentes de software podem associar-se e interagir quando as entidades mudam, falham ou aparecem espontaneamente; como os sistemas são integrados no mundo físico, por meio da percepção e do reconhecimento de contexto; os problemas da segurança e privacidade que surgem nos sistemas voláteis e fisicamente integrados; e as técnicas para se adaptar à falta de recursos computacionais e de E/S dos dispositivos portáteis. O capítulo termina com um estudo de caso do projeto Cooltown, que planejou uma arquitetura orientada para seres humanos, baseada na web, para computação móvel e ubíqua.

16.1 Introdução

A computação móvel e ubíqua surgiu devido à miniaturização dos dispositivos e da conectividade sem fio. De modo geral, a computação móvel ocupa-se da exploração da conexão de dispositivos que se movimentam no mundo físico cotidiano; a computação ubíqua diz respeito à exploração da integração cada vez maior dos dispositivos de computação com nosso mundo físico. À medida que os equipamentos se tornam menores, fica mais fácil levá-los conosco ou vesti-los, e podemos incorporá-los em muitas partes do mundo físico – e não apenas no já comum *desktop* ou no *rack* de um servidor. E, à medida que a conectividade sem fio se torna predominante, podemos conectar melhor esses novos e pequenos dispositivos uns com os outros, com computadores pessoais e com servidores convencionais.

Este capítulo examina os aspectos da computação móvel (um assunto já mencionado no tratamento da operação desconectada do Capítulo 15) e da computação ubíqua. O capítulo se concentra nas propriedades comuns e nas diferenças que compartilham com os sistemas distribuídos mais convencionais. Dado o progresso feito até agora, o capítulo fala mais sobre os problemas em aberto do que sobre soluções.

Primeiramente, este capítulo descreve em linhas gerais os princípios da computação móvel e ubíqua e apresenta as subáreas conhecidas como computação acoplada ao corpo (*wearable*), de mão (*handheld*) e com reconhecimento de contexto (*context-aware computing*). Depois disso, ele descreve um modelo de sistema que abrange todas essas áreas e subáreas por meio de sua volatilidade – o conjunto de usuários, dispositivos e componentes de software, em determinado ambiente, é sujeito a mudanças freqüentes. Em seguida, o capítulo examina algumas das principais áreas de pesquisa que surgiram por causa da volatilidade e de suas bases físicas, incluindo: como os componentes de software se associam e interagem uns com os outros quando as entidades se movem, falham ou aparecem espontaneamente nos ambientes; como os sistemas se tornam integrados com o mundo físico, por meio da percepção e do reconhecimento de contexto; os problemas da segurança e privacidade que surgem nos sistemas voláteis e fisicamente integrados; e as técnicas para se adaptar à falta de recursos computacionais e de E/S dos dispositivos portáteis. O capítulo termina com um estudo de caso do projeto Cooltown, que planejou uma arquitetura orientada para seres humanos, baseada na web, para computação móvel e ubíqua.

Computação móvel ◊ A computação móvel surgiu como um paradigma no qual os usuários poderiam carregar seus computadores pessoais e manter certa conectividade com outras máquinas. Por volta de 1980, tornou-se possível construir computadores pessoais leves o suficiente para serem carregados e que podiam ser conectados a outros computadores por meio de linhas telefônicas, usando um modem. A evolução tecnológica levou a mais ou menos a mesma idéia, mas com funcionalidade e desempenho muito melhores: o equivalente atual é um *laptop*, ou o menor tipo de computador *notebook*, com combinações de conectividade sem fio, incluindo as tecnologias de telecomunicações com sinal infravermelho, WiFi, Bluetooth e GPRS ou 3G.

Um caminho diferente da evolução tecnológica levou à *computação de mão* (*handheld computing*): o uso de aparelhos que cabem na mão, incluindo os assistentes digitais pessoais (PDAs – *Personal Digital Assistants*), os telefones móveis e outros equipamentos mais especializados operados manualmente. Os PDAs são computadores de propósito geral, capazes de executar muitos tipos diferentes de aplicações; mas, comparados aos *laptops* e *notebooks*, tem menor tamanho e capacidade da bateria, possuem um poder de processamento correspondentemente limitado, uma tela menor e outras restrições de recursos. Cada vez mais, os fabricantes equipam os PDAs com a mesma variedade de conectividade sem fio que os *laptops* e *notebooks* possuem.

Uma tendência interessante na computação de mão tem sido a confusão na distinção entre PDAs, telefones móveis e equipamentos de mão de finalidade específica, como as câmaras digitais. Vários tipos de telefone móvel têm funcionalidade de computação do tipo do PDA, em virtude de executarem os sistemas operacionais Linux, Symbian ou Microsoft Smartphone. Os PDAs e os telefones móveis podem ser equipados com câmaras, leitores de código de barras e outros tipos de acessórios especializados, tornado-os uma alternativa aos equipamentos de mão de finalidade específica. Por exemplo, um usuário que queira tirar fotografias digitais pode usar uma câmara de finalidade específica, um PDA com uma câmara incorporada ou um telefone com câmara. Todos eles vêm (ou podem ser adquiridos) com uma forma de conectividade sem fio de curto ou longo alcance.

Stojmenovic [2002] aborda os princípios e protocolos da comunicação sem fio, incluindo uma abordagem dos dois maiores problemas da camada de rede que precisam ser resolvidos para os sistemas estudados neste capítulo. O primeiro problema é como fornecer conectividade contínua para dispositivos móveis que entram e saem do alcance das *estações de base*, que são os componentes de infra-estrutura que fornecem regiões de cobertura sem fio. O segundo problema é como permitir que conjuntos de dispositivos se comuniquem sem fio uns com os outros, em lugares onde não existe infra-estrutura (veja o breve tratamento das *redes ad hoc*, na Seção 16.4.2). Os dois problemas surgem porque, freqüentemente, a conectividade sem fio direta não está disponível entre quaisquer dois dispositivos. Então, a comunicação precisa ser obtida por meio de vários segmentos de rede com ou sem fio. Dois fatores principais levam a essa cobertura sem fio subdividida. Primeiro, quanto maior o alcance de uma rede sem fio, mais dispositivos competirão por sua largura de banda limitada. Segundo, considerações sobre a energia se aplicam: a energia necessária para transmitir um sinal sem fio é proporcional ao quadrado de seu alcance; mas muitos dos dispositivos que consideraremos têm capacidade de energia limitada.

Computação ubíqua ◊ Mark Weiser cunhou o termo computação ubíqua, em 1988 [Weiser 1991]. Às vezes, a computação ubíqua também é conhecida como computação pervasiva e os dois termos normalmente são considerados sinônimos. "Ubíquo" significa "em toda parte". Weiser percebeu a predominância cada vez maior dos dispositivos de computação levando a mudanças revolucionárias na maneira como usaríamos os computadores.

Primeiro, cada pessoa no mundo utilizaria muitos computadores. Podemos comparar isso à revolução da computação pessoal anterior a essa, que viu um computador para cada pessoa. Embora pareça simples, essa mudança teve um efeito dramático sobre a maneira como usamos os computadores, em comparação à era anterior dos computadores de grande porte, quando havia apenas um computador para muitas pessoas. A idéia de Weiser de "uma pessoa, muitos computadores" significa algo muito diferente da situação comum, na qual cada um de nós tem vários computadores mais ou menos parecidos – um no trabalho, um em casa, um *laptop* e, talvez, um PDA que levamos conosco. Em vez disso, na computação ubíqua, os computadores se multiplicam na forma e na função e não apenas no número, para atender a diferentes tarefas.

Por exemplo, suponha que todos os meios de exibição e escrita fixos de uma sala – quadros, livros, folhas de papel, notas adesivas, etc. – fossem substituídos por dezenas ou centenas de computadores individuais com telas eletrônicas. Os quadros poderiam ajudar as pessoas a desenhar, organizar e arquivar suas idéias; os livros poderiam se tornar equipamentos que permitissem aos leitores pesquisarem seu texto, procurarem o significado de palavras, buscarem idéias relacionadas na web e verem conteúdo multimídia vinculado. Agora, incorpore funcionalidade de computação em todas as ferramentas de escrita. Por exemplo, as canetas e os marcadores se tornariam capazes de armazenar o que o usuário escreveu e desenhou, e de reunir, copiar e mover conteúdo multimídia entre os muitos computadores situados nas imediações. Esse cenário levanta questões de capacidade de utilização e econômicos e toca apenas em uma pequena parte de nossas vidas, mas nos dá uma idéia do que a "computação por toda parte" poderia ser.

A segunda mudança que Weiser previu foi que os computadores "desapareceriam" – que eles "se incorporariam em utensílios e objetos do dia a dia, até se tornarem indistinguíveis". Essa é principalmente uma noção psicológica, comparável ao modo como as pessoas aceitam normalmente a mobília de uma casa e mal a notam. Ela reflete a idéia de que a computação estará incorporada ao que consideramos itens do cotidiano – aqueles que normalmente não achamos que tenham recursos computacionais, nada diferente do que pensamos a respeito das máquinas de lavar, ou dos veículos como "equipamentos de computação", mesmo tendo o controle de microprocessadores incorporados – cerca de 100 microprocessadores, no caso de alguns carros.

Embora a invisibilidade de certos dispositivos seja apropriada – nos casos como os sistemas de computadores incorporados em um carro –, isso não vale para todos os equipamentos que vamos considerar, particularmente para os dispositivos que os usuários móveis normalmente carregam. Por exemplo, os telefones móveis eram uns dos dispositivos de maior penetração, quando este livro foi escrito, mas sua capacidade computacional dificilmente era visível e, com certeza, nem deveria ser.

Computação acoplada ao corpo (*wearable computing*) ◊ Os usuários levam equipamentos de computação acoplados a si mesmos, presos no tecido de suas roupas ou dentro dele, ou transportados em seus próprios corpos, como relógios, jóias ou óculos. Ao contrário dos equipamentos de mão mencionados anteriormente, esses dispositivos freqüentemente funcionam sem que o usuário precise manipulá-los. Normalmente, eles possuem funcionalidade especializada. Um exemplo primitivo é o "crachá ativo", um pequeno equipamento de computação preso ao usuário, que transmite regularmente a identidade do crachá (associada a um usuário) por intermédio de um transmissor de sinal infravermelho [Want *et al.* 1992; Harter e Hopper 1994]. A idéia do crachá é que os dispositivos presentes no ambiente respondam às suas transmissões e, assim, respondam à presença de um usuário; as transmissões de sinal infravermelho têm um alcance limitado e, portanto, serão captadas apenas se o usuário estiver nas proximidades. Por exemplo, uma tela eletrônica poderia ser adaptada à presença de um usuário, personalizando-se seu comportamento de acordo com as preferências desse usuário, como a cor do desenho e a espessura de linha padrão (Figura 16.1). Uma sala poderia ser adaptada para ajustes de ar condicionado e iluminação, de acordo com a pessoa que estivesse dentro dela.

Computação com reconhecimento de contexto (*context-aware computing*) ◊ O crachá ativo – ou melhor, as reações de outros dispositivos à sua presença – exemplifica a computação com reconhecimento de contexto, que é uma subárea importante da computação móvel e ubíqua. É onde os sistemas de computadores adaptam seu comportamento automaticamente, de acordo com as circunstâncias físicas. Essas circunstâncias podem, em princípio, ser algo fisicamente medido ou detectado, como a presença de um usuário, a hora do dia ou as condições atmosféricas. Algumas das condições dependentes são relativamente simples de determinar, como o fato de ser noite (a partir da hora, do dia do ano e da posição geográfica). Mas outras exigem processamento sofisticado para sua detecção. Por exemplo, considere um telefone móvel com reconhecimento de contexto, que só deve tocar quando for apropriado. Em particular, ele deve trocar automaticamente para o modo "vibrar", em vez de "tocar", quando estiver no cinema. Mas não é simples detectar que o usuário está assistindo a um filme dentro de um cinema e não parado no saguão, dadas às imprecisões das medidas do sensor de posição. A Seção 16.4 examinará o contexto com mais detalhes.

16.1.1 Sistemas voláteis

Do ponto de vista dos sistemas distribuídos, não há nenhuma diferença básica entre computação móvel e ubíqua, ou as subáreas que apresentamos (ou, na verdade, as subáreas que omitimos, como a computação tangível [Ishii e Ullmer 1997] e a realidade ampliada, exemplificada pela escrivaninha digital de Wellner [Wellner 1991]). Nesta subseção, mostraremos um modelo do que chamamos de *sistemas voláteis*, que abrangem os recursos dos sistemas distribuídos básicos de todos eles.

Chamamos os sistemas descritos neste capítulo de voláteis porque, ao contrário da maioria dos sistemas descritos nas outras partes deste livro, certas mudanças são comuns, em vez de excepcionais. O conjunto de usuários, hardware e software nos sistemas móveis e ubíquos é altamente dinâmico e muda de maneira imprevisível. Outra palavra que às vezes usaremos para esses sistemas é *espontâneo*,

Figura 16.1 Uma sala respondendo a um usuário portando um crachá ativo.

que aparece na literatura, na frase *interligação em rede espontânea*. As formas relevantes de volatilidade incluem:

- falhas de dispositivos e enlaces de comunicação;
- mudanças nas características da comunicação, como a largura de banda;
- a criação e destruição de *associações* – relacionamentos de comunicação lógicos – entre os componentes de software residentes nos dispositivos.

Aqui, o termo "componente" abrange qualquer unidade de software, como objetos ou processos, independentemente de interagir como cliente, servidor ou *peer*.

O Capítulo 15 já mostrou as maneiras de tratar com algumas dessas mudanças, a saber, falhas de processamento e a operação desconectada. Mas as soluções lá mostradas foram dadas considerando-se as falhas de processamento e comunicação como sendo a exceção e não a regra, e levando em conta a existência de recursos de processamento redundantes. Os sistemas voláteis não apenas violam essas suposições, como também acrescentam ainda mais fenômenos de alteração, notadamente às mudanças freqüentes nas associações entre os componentes.

É importante esclarecermos um possível mal-entendido, antes de prosseguirmos. A volatilidade não é uma propriedade *que define* os sistemas móveis e ubíquos: existem outros tipos de sistema que demonstram uma ou mais formas de volatilidade, mas que não são móveis, nem ubíquos. Um bom exemplo é a computação *peer-to-peer*, como os aplicativos de compartilhamento de arquivo (Capítulo 10), na qual o conjunto de processos participantes e as associações entre eles estão sujeitos a altas taxas de mudança. O que é diferente na computação móvel e ubíqua é que elas exibem *todas* as formas anteriores de volatilidade, devido à maneira como são integradas com o mundo físico. Temos muito a dizer sobre essa integração física e como ela causa volatilidade. Mas a integração física em si não é uma propriedade dos sistemas distribuídos, enquanto a volatilidade, sim. Daí vem o termo que adotamos.

No restante desta seção, descreveremos os espaços inteligentes (*smart spaces*), que são os ambientes dentro dos quais os sistemas voláteis subsistem; em seguida, caracterizaremos os dispositivos móveis e ubíquos, sua conectividade física e lógica e as conseqüências da menor confiança e privacidade.

Espaços inteligentes (*smart spaces*) ◊ Os espaços físicos são importantes, pois eles formam a base da computação móvel e ubíqua. Ocorre mobilidade entre os espaços físicos; a computação ubíqua é incorporada em espaços físicos. Um *espaço inteligente* é qualquer local físico com serviços incorporados – isto é, serviços fornecidos apenas ou principalmente dentro desse espaço físico. É possível introduzir dispositivos de computação ao acaso, onde não existe nenhuma infra-estrutura, para executar uma aplicação, como o monitoramento ambiental. Mas, normalmente, os dispositivos móveis e sistemas ubíquos existem em dado momento em uma peça com melhorias computacionais do ambiente construído ou veicular, como uma sala, um prédio, quarteirão ou vagão de trem. Nesses casos, o espaço inteligente normalmente contém uma infra-estrutura de computação relativamente estável, a qual pode incluir computadores servidores convencionais, dispositivos como impressoras e telas, sensores e uma infra-estrutura de rede sem fio, incluindo uma conexão com a Internet.

Existem vários tipos de movimentação ou "aparecimento e desaparecimento" que podem ocorrer nos espaços inteligentes. Primeiro, existe a *mobilidade física*. Os espaços inteligentes atuam como ambientes para os dispositivos que os visitam e saem. Os usuários entram e saem com equipamentos que carregam, ou vestem, dispositivos de robótica podem até se mover sozinhos, entrando e saindo do espaço. Segundo, existe a *mobilidade lógica*. Um processo, ou agente móvel, pode entrar ou sair de um espaço inteligente ou do dispositivo pessoal de um usuário. Além disso, a movimentação física de um dispositivo pode causar a movimentação lógica dos componentes dentro dele. Entretanto, tenha um componente se movido ou não, devido à movimentação física de seu dispositivo, a mobilidade lógica não ocorreu de nenhuma maneira interessante, a não ser que, como resultado, o componente tenha alterado alguma de suas associações com outros componentes. Terceiro, os usuários podem adicionar dispositivos relativamente estáticos no espaço, como reprodutores de mídia, como acréscimos de mais longo prazo, e correspondentemente retirar dele dispositivos mais antigos. Considere,

por exemplo, a evolução de uma *casa inteligente*, cujos ocupantes variam o conjunto de dispositivos dentro dela [Edwards e Grinter 2001] de uma maneira relativamente não planejada com o passar do tempo. Quarto e último, os dispositivos podem falhar e, assim, "desaparecer" de um espaço.

Da perspectiva dos sistemas distribuídos, alguns desses fenômenos parecem semelhantes. Em cada caso, um componente de software *aparece* em um espaço inteligente previamente existente e, se resultar algo interessante, ele torna-se integrado a esse espaço, pelo menos temporariamente; ou, então, um componente *desaparece* do espaço por causa da mobilidade ou porque é simplesmente desligado ou porque falha. Pode ou não ser possível que qualquer componente em particular faça distinção entre dispositivos da "infra-estrutura" e dispositivos "visitantes".

Entretanto, existem distinções significativas a serem deduzidas no projeto de um sistema. Uma diferença importante que pode surgir entre sistemas voláteis é a taxa de alteração. Algoritmos que precisam aceitar vários componentes aparecendo ou desaparecendo por dia (por exemplo, em uma casa inteligente) podem ser projetados de forma muito diferente daqueles para os quais há pelo menos uma dessas mudanças ocorrendo em dado momento (por exemplo, um sistema implementado usando comunicação Bluetooth entre telefones móveis em uma cidade densamente povoada). Além disso, embora todos os fenômenos de aparecimento e desaparecimento anteriores pareçam semelhantes em uma primeira aproximação, é claro que existem diferenças importantes. Por exemplo, do ponto de vista da segurança, uma coisa é o dispositivo de um usuário entrar em um espaço inteligente, e outra é um componente de software externo se mover em um dispositivo de infra-estrutura pertencente ao espaço.

Modelo de dispositivo ◊ Com o surgimento da computação móvel e ubíqua, uma nova classe de dispositivos de computação está se tornando parte dos sistemas distribuídos. Esses dispositivos são limitados em sua fonte de energia e em seus recursos de computação; e eles podem ter maneiras próprias para fazer a interface com o mundo físico: sensores, como os detectores de luz, e/ou controladores, como um meio de movimentação programável.

Energia limitada: um dispositivo portátil ou incorporado no mundo físico normalmente precisa funcionar com baterias, e quanto menor e mais leve o dispositivo precisa ser, menor será a capacidade de sua bateria. Substituir ou recarregar essas baterias provavelmente será inconveniente, em termos de tempo (pode haver centenas desses dispositivos por usuário) e acesso físico. Computação, acesso à memória e outras formas de armazenamento, tudo isso consome energia preciosa. A comunicação sem fio é particularmente dispendiosa em termos de energia. Além disso, a energia consumida pela recepção de uma mensagem pode significar uma fração substancial daquela exigida para transmiti-la; mesmo o modo de "espera", no qual uma interface de rede está pronta para receber uma mensagem, pode exigir um consumo de energia considerável [Shih *et al.* 2002]. Assim, se um equipamento precisa permanecer, enquanto for possível, com determinado nível de carga na bateria, os algoritmos precisam ser sensíveis à energia que eles consomem, especialmente em termos de sua complexidade de mensagem. Mas, em última análise, a probabilidade de falha de dispositivo aumenta devido à descarga da bateria.

Restrições de recurso: os dispositivos móveis e ubíquos têm recursos computacionais limitados, em termos de velocidade do processador, capacidade de armazenamento e largura de banda de rede. Isso se dá, em parte, porque o consumo de energia aumenta quando melhoramos essas características. Mas também acontece porque tornar os dispositivos portáteis, ou incorporá-los nos objetos físicos cotidianos, significa torná-los fisicamente pequenos, o que, dadas às limitações impostas pelos processos de fabricação, restringe o número de transistores nos microprocessadores e controladores. Isso resulta em dois problemas: como projetar algoritmos que possam ser executados em um tempo razoável, apesar dessas limitações, e como aumentar os recursos escassos usando os recursos do próprio ambiente.

Sensores e controladores: para permitir sua integração com o mundo físico – em particular, para fazê-los ter reconhecimento de contexto –, os dispositivos são equipados com *sensores* e *controladores*. Os sensores são dispositivos que medem parâmetros físicos e fornecem seus valores para o software. Inversamente, os controladores são dispositivos controlados pelo software, que afetam o mundo físico. Existe uma grande variedade de cada tipo de componente. No lado dos

sensores, por exemplo, existem os que medem posição, orientação, carga e níveis de luz e som. Os controladores incluem aqueles para ar condicionado programável e motores. Um problema importante para os sensores é a precisão, que é bastante limitada e, portanto, pode levar a um comportamento espúrio, como uma resposta inadequada para o que se revela ser o lugar errado. A imprecisão provavelmente continuará sendo uma característica dos dispositivos que são baratos o suficiente para serem distribuídos por toda parte.

Os dispositivos descritos anteriormente parecem um tanto exóticos. Entretanto, eles não apenas estão disponíveis comercialmente, como até são produzidos em massa. Dois exemplos são as partículas (*motes*, em inglês) e os telefones com câmara.

Partículas (motes): as partículas [Hill *et al.* 2000; www.xbow.com] são dispositivos destinados à operação autônoma em aplicações como a percepção ambiental. Eles são projetados para serem incorporados em um ambiente, programados de modo a descobrirem um ao outro, sem fio, e transferir entre si os valores percebidos. Se, por exemplo, houver um incêndio em uma floresta, então uma ou mais partículas espalhadas pela floresta poderão perceber temperaturas anormalmente altas e comunicá-las, por intermédio de seus pares, a um dispositivo mais potente, capaz de repassar a situação para os serviços de emergência. A forma mais básica de partícula tem um processador de baixa potência (um microcontrolador) que executa o sistema operacional TinyOS [Culler *et al.* 2001] em uma memória flash interna, uma memória para registro de dados e um transceptor de sinais de rádio bidirecional ISM (*Industrial, Scientific and Medical*), de curto alcance. Uma variedade de módulos sensores pode ser adicionada. As partículas também são conhecidas como "pó inteligente", refletindo o tamanho minúsculo pretendido, em última análise, para esses dispositivos, embora seu tamanho, quando este livro estava sendo escrito, fosse em torno de 6 × 3 × 1 cm, excluindo-se o compartimento de bateria e os sensores. A Smart-its fornece funcionalidade similar para partículas com um fator de forma semelhante [www.smart-its.org]. A Seção 16.4.2 discutirá os usos de dispositivos do tipo partícula em redes de sensores sem fio.

Telefones com câmara: os telefones com câmara são exemplos bem diferentes de dispositivos nos sistemas que estamos considerando. Suas funções principais são a comunicação humana e a geração de imagens. Mas, executando um sistema operacional como o Symbian, eles podem ser programados para uma ampla variedade de aplicações. Além de sua conectividade de dados remota, eles freqüentemente têm interfaces de rede sem fio infravermelho (IrDA), ou Bluetooth, de curto alcance, que permite conectarem-se uns aos outros, com PCs e com dispositivos sensores, como o GPS ou outras unidades de navegação via satélite, para determinar sua localização. Além disso, eles podem executar software para reconhecimento de símbolos, como códigos de barra, a partir das imagens de suas câmaras, tornando-os sensores de "valores codificados" em objetos físicos, como produtos, que podem ser usados para acessar serviços associados. Por exemplo, um usuário poderia usar seu telefone com câmara para saber as especificações de um produto em uma loja, a partir do código de barras existente em sua caixa [Kindberg 2002].

Conectividade volátil ◊ Todos os dispositivos de interesse deste capítulo têm alguma forma de conectividade sem fio, e podem ter várias. As tecnologias de conexão (seja Bluetooth, WiFi, GPRS, etc.) variam em sua largura de banda e latência nominais, em seus custos de energia e no fato de existirem ou não custos financeiros na comunicação. Mas a volatilidade da conectividade – a variabilidade, em tempo de execução, do estado da conexão ou desconexão entre os dispositivos e a qualidade do serviço entre eles – também tem forte impacto sobre as propriedades do sistema.

Desconexão: as desconexões sem fio são bem mais prováveis de acontecer do que as desconexões em redes com fio. Muitos dos dispositivos que descrevemos são móveis e, portanto, podem ultrapassar sua distância operacional em relação aos outros dispositivos, ou encontrar obstruções de sinal de rádio entre eles, por exemplo, por causa de prédios. Mesmo quando os dispositivos são estáticos, podem existir usuários e veículos se movendo, os quais causam desconexão por obstrução. Também há a questão do roteamento de múltiplos *hops* (saltos) sem fio entre dispositivos. No *roteamento ad hoc*, um conjunto de dispositivos se comunica sem confiar em nenhum outro dispositivo: eles cooperam para direcionar todos os pacotes entre si. Usando nosso exemplo de partículas em uma floresta, uma partícula poderia continuar a se comunicar com todas as outras

partículas em uma faixa de rádio imediata, mas deixar de comunicar sua leitura de temperatura alta para os serviços de emergência, devido à falha de partículas mais distantes, pelas quais todos os pacotes teriam que passar.

Largura de banda e latência variáveis: os fatores que podem levar a uma completa desconexão também podem levar a uma largura de banda e latência altamente variáveis, pois acarretam taxas de erro variáveis. À medida que a taxa de erro aumenta, cada vez mais pacotes são perdidos. Isso leva intrinsecamente a baixas taxas de desempenho de saída (*throughput*). Mas a situação pode ser agravada por tempos limites (*timeout*) nos protocolos de camadas mais altas. Os valores de tempo limite são difíceis de adaptar em condições dramaticamente variáveis. Se eles forem grandes demais, comparados com as condições de erro correntes, a latência e o desempenho de saída sofrerão. Se eles forem pequenos demais, poderão aumentar o congestionamento e desperdiçar energia.

Interação espontânea ◊ Em um sistema volátil, os componentes rotineiramente mudam o conjunto de componentes com que se comunicam à medida que se movem ou que outros componentes aparecem em seu ambiente. Usamos o termo *associação* para o relacionamento lógico formado quando pelo menos um componente de determinado par de componentes se comunica com o outro, durante um período de tempo bem definido, e *interação* para suas interações durante a associação. Note que associação é diferente de conectividade: dois componentes (por exemplo, um cliente de e-mail, em um *laptop*, e um servidor de e-mail) podem estar correntemente desconectados, enquanto permanecem associados.

Em um espaço inteligente, as associações mudam porque os componentes tiram proveito de oportunidades para interagir com componentes locais. Um exemplo simples de tal oportunidade é quando um dispositivo utiliza uma impressora local onde quer que esteja no momento. Analogamente, talvez um dispositivo queira oferecer serviços para os clientes em seu ambiente local – como um "servidor pessoal" [Want *et al.* 2002] que o usuário veste (por exemplo, em seu cinto), o qual fornece atributos personalizados sobre o usuário para uma unidade de ar condicionado. Essa unidade, então, é capaz de ajustar as condições da sala de acordo com as preferências do usuário. É claro que certas associações estáticas ainda fazem sentido, mesmo em um sistema volátil; demos o exemplo de um computador *laptop* que viaja com seu proprietário pelo mundo, mas que só se comunica com um servidor de e-mail fixo.

Para colocar esse tipo de associação em um contexto maior de serviços na Internet, a Figura 16.2 mostra exemplos de três tipos de associação espontânea (à direita), comparadas com associações previamente configuradas (à esquerda).

As associações previamente configuradas são orientadas a serviços; isto é, os clientes têm uma necessidade a longo prazo de usar um serviço específico e, portanto, elas são previamente configuradas para serem associadas a ele. O trabalho de configuração dos clientes (incluindo configurá-los com o endereço do serviço exigido) é pequeno, comparado com o valor a longo prazo de usar o serviço, em particular.

Previamente configurada	*Espontânea*
	Orientada a seres humanos: *navegador web e servidores web*
Orientada a serviços: *cliente e servidor de e-mail*	Orientada a dados: *aplicativos de compartilhamento de arquivo P2P*
	Fisicamente orientada: *sistemas móveis e ubíquos*

Figura 16.2 Exemplos de associação previamente configurada *versus* espontânea.

No lado direito da figura estão os tipos de associação que variam rotineiramente, orientados por um operador humano, pela necessidade de dados específicos ou pelas circunstâncias físicas variáveis. Podemos considerar as associações entre um navegador web e serviços web como espontâneos e *orientados a seres humanos*: o usuário faz escolhas dinâmicas e (do ponto de vista do sistema) imprevisíveis de *links* para clicar e, assim, da instância de serviço a acessar. A web é um sistema verdadeiramente volátil, e o fato de a mudança nas associações normalmente envolver um trabalho desprezível é importante para seu sucesso – os autores das páginas web já fizeram o trabalho de configuração.

As aplicações *peer-to-peer* na Internet, como os programas de compartilhamento de arquivo, também são sistemas voláteis, mas elas são principalmente *orientadas a dados*. Esses dados freqüentemente se originam do ser humano (por exemplo, o nome do conteúdo a ser buscado), mas é o valor dos dados fornecidos que faz com que um par estabeleça associações com outro, com o qual pode nunca ter se associado e cujo endereço não foi armazenado por ele anteriormente, por meio de um algoritmo de descoberta distribuído, baseado em dados.

Os sistemas móveis e ubíquos deste capítulo são diferenciados por exibirem espontaneidade de associações *fisicamente orientadas*. As associações são estabelecidas e desfeitas – às vezes por seres humanos –, de acordo com as circunstâncias físicas correntes dos componentes, em particular, a sua proximidade.

Menor confiança e privacidade ◊ Conforme foi explicado no Capítulo 7, em última análise, a segurança nos sistemas distribuídos é baseada em hardware e software confiáveis – a base da computação confiável. Mas nos sistemas voláteis a confiança é problemática, devido à interação espontânea. Que base de confiança pode haver entre componentes que são capazes de se associar espontaneamente? Os componentes que se movem entre espaços inteligentes podem pertencer a indivíduos ou organizações distintos, e têm pouco ou nenhum conhecimento anterior uns dos outros ou de um terceiro participante confiável.

A privacidade é um problema importante para os usuários, que podem desconfiar dos sistemas por causa de seus recursos de percepção. A presença de sensores nos espaços inteligentes significa que se torna possível rastrear os usuários eletronicamente, em uma escala potencialmente maciça e jamais vista. Tirando proveito dos serviços de reconhecimento de contexto – como no exemplo das salas que configuram o ar condicionado de acordo com as preferências dos usuários que estão dentro delas –, os usuários podem permitir que outros saibam onde eles estavam e o que estavam fazendo lá. Para piorar as coisas, eles podem nem sempre saber que estavam sendo seguidos. Mesmo que o usuário não revele sua identidade, é possível que outros a saibam e, assim, descubram o que um indivíduo em particular faz – por exemplo, observando as viagens regulares entre uma casa e um local de trabalho e correlacionando-os com o uso de um cartão de crédito em algum lugar entre esses dois pontos.

16.2 Associação

Conforme explicado anteriormente, os dispositivos estão sujeitos a aparecer e desaparecer nos espaços inteligentes de maneira imprevisível. Apesar disso, os componentes voláteis precisam interagir – preferivelmente sem intervenção do usuário. Em outras palavras, um dispositivo que aparece em um espaço inteligente precisa conseguir se inicializar na rede local para possibilitar a comunicação com outros dispositivos e se associar apropriadamente no espaço inteligente:

- Inicialização na rede. Normalmente, a comunicação ocorre por meio de uma rede local. O dispositivo deve primeiro adquirir um endereço na rede local (ou registrar um endereço previamente existente, como um endereço IP móvel); ele também pode adquirir ou registrar um nome.
- Associação. Os componentes do dispositivo se associam aos serviços no espaço inteligente ou fornecem serviços para componentes em qualquer parte do espaço inteligente (ou ambos).

Inicialização na rede ◊ Existem soluções bem estabelecidas para o problema da integração de um dispositivo na rede. Algumas dessas soluções contam com servidores acessíveis dentro do espaço

Hidden page

determinados dinamicamente como uma função do contexto do cliente – neste caso, o espaço inteligente em particular onde as consultas ocorrem. Segundo, pode não haver infra-estrutura no espaço inteligente para conter um servidor de diretório. Terceiro, os serviços registrados no diretório podem desaparecer espontaneamente. Quarto, os protocolos usados para acessar o diretório precisam ser sensíveis ao consumo de energia e largura de banda.

Existem tanto serviços de *descoberta de dispositivo* como de *descoberta de serviço*; o Bluetooth inclui ambos. Na descoberta de dispositivo, os clientes descobrem os nomes e endereços de dispositivos presentes no mesmo local. Normalmente, eles escolhem um dispositivo individual com base em uma informação externa ao sistema (como a seleção por um ser humano) e o consultam para saber os serviços que ele oferece. Por outro lado, um serviço de descoberta de serviço é usado onde os clientes não estão preocupados com qual dispositivo fornece o serviço que precisam, mas somente com os atributos do serviço. Esta descrição se concentrará nos serviços de descoberta de serviço e, a não ser que seja declarado de outra forma, é a isso que nos referiremos daqui por diante como serviços de descoberta.

Um serviço de descoberta tem uma interface para automaticamente registrar e anular o registro dos serviços que estão disponíveis para associação, assim como uma interface para os clientes pesquisarem serviços que estão correntemente disponíveis a partir desses, para fazerem a associação com um serviço apropriado. A Figura 16.3 dá um exemplo fictício e simplificado dessas interfaces. Primeiro, existem chamadas para registrar a disponibilidade de um serviço com determinado endereço e atributos e, subseqüentemente, para gerenciar seu registro. Em seguida, existe uma chamada para pesquisar os serviços que correspondem a uma especificação de atributos exigidos. Zero ou mais serviços podem corresponder à especificação; cada um é retornado com seu endereço e seus atributos. Note que, por si só, um serviço de descoberta não permite associação: também é exigida a *seleção do serviço* – a escolha de um serviço no conjunto retornado. Isso pode ocorrer por meio de um programa ou pela listagem dos serviços correspondentes, para um usuário escolher.

Os desenvolvimentos nos serviços de descoberta incluem o serviço de descoberta Jini (veja a seguir), o protocolo de localização de serviço [Guttman 1999], o *Intentional Naming System* [Adjie-Winoto *et al.* 1999], o protocolo de descoberta de serviço simples, que é o cerne da iniciativa Universal *Plug and Play* [www.upnp.org] e o *Secure Service Discovery Service* [Czerwinski *et al.* 1999]. Também existem serviços de descoberta na camada de enlace, como o do Bluetooth.

Os problemas a serem tratados no projeto de um serviço de descoberta são os seguintes:

- Pouco esforço, associação apropriada. De preferência, as associações apropriadas seriam feitas sem qualquer trabalho humano. Primeiro, o conjunto de serviços retornado pela operação *consulta* (Figura 16.3) seria apropriado – eles seriam precisamente os serviços existentes no espaço inteligente que corresponderam à consulta. Segundo, a seleção do serviço poderia ser feita por programa, ou com entrada humana mínima, para atender as necessidades dos usuários.

Métodos para registro/cancelamento de registro de serviço	Explicação
arrendamento := registrar(endereço, atributos)	Registra o serviço no endereço dado, com os atributos dados; é retornado um arrendamento (*lease*)
atualizar(arrendamento)	Atualiza o arrendamento retornado no registro
cancelarRegistro(arrendamento)	Remove o registro do serviço registrado sob o arrendamento dado
Método invocado para pesquisar um serviço	
conjuntoServiço := consultar(especificaçãoAtributo)	Retorna um conjunto de serviços registrados cujos atributos correspondem à especificação dada

Figura 16.3 Interface para um serviço de descoberta.

Hidden page

Hidden page

Jini ◊ O Jini [Waldo 1999; Arnold *et al.* 1999] é um sistema projetado para ser usado por sistemas móveis e ubíquos. Ele é totalmente baseado em Java – ele presume que máquinas virtuais Java são executadas em todos os computadores, permitindo que eles se comuniquem uns com os outros por meio de RMI, ou eventos (veja o Capítulo 5), e façam o *download* de código, conforme for necessário. Descreveremos aqui o sistema de descoberta do Jini.

Os componentes relacionados à descoberta em um sistema Jini são serviços de *pesquisa* (*lookup*), serviços Jini e clientes Jini (veja a Figura 16.4). O serviço de *pesquisa* implementa o que chamamos de serviço de descoberta, embora o Jini use o termo "descoberta" apenas para descobrir o serviço de pesquisa em si. O serviço de pesquisa permite que os serviços Jini registrem as tarefas que oferecem e os clientes Jini solicitam os serviços que correspondem aos seus requisitos. Um serviço Jini, como um serviço de impressão, pode ser registrado em um ou mais serviços de pesquisa. Um serviço Jini fornece (e os serviços de pesquisa armazenam) um objeto que providencia o serviço, assim como os atributos do serviço. Os clientes Jini consultam os serviços de pesquisa para encontrar serviços Jini que correspondam aos seus requisitos; caso uma correspondência seja encontrada, eles fazem o *download* de um objeto que dá acesso ao serviço, a partir do serviço de pesquisa. A correspondência que o serviço oferece as requisições dos clientes pode ser baseada em atributos ou em tipos Java, por exemplo, permitindo que um cliente solicite uma impressora colorida para a qual possui a interface Java correspondente.

Quando um cliente ou serviço Jini inicia, ele envia uma requisição para um endereço *multicast* IP conhecido. Qualquer serviço de pesquisa que receba essa requisição, que possa responder, envia seu endereço, permitindo que o solicitante realize uma invocação remota para pesquisar ou registrar um serviço nele (no Jini, o registro é chamado de *união – joining*, em inglês). Os serviços de pesquisa também anunciam sua existência em datagramas enviados para o mesmo endereço *multicast*. Os clientes e serviços Jini também podem captar o endereço *multicast* para que saibam da existência de novos serviços de pesquisa.

Pode haver várias instâncias do serviço de pesquisa acessíveis por comunicação *multicast* a partir de determinado cliente ou serviço Jini. Toda instância de tal serviço é configurada com um ou mais nomes de *grupo*, como *admin*, *financeiro* e *vendas*, os quais atuam como rótulos de escopo. A Figura 16.4 mostra um cliente Jini descobrindo e usando um serviço de impressão. O cliente solicita um serviço de pesquisa no grupo financeiro; portanto, ele envia em *multicast* uma requisição contendo esse nome de grupo (mensagem 1 na figura). Somente um serviço de pesquisa está ligado ao grupo financeiro (o serviço que também está ligado ao grupo admin) e esse serviço responde (2). A resposta do serviço de pesquisa inclui seu endereço e o cliente se comunica diretamente com ele, por RMI, para

Figura 16.4 **Descoberta de serviço no Jini.**

localizar todos os serviços de tipo "impressão" (3). Somente um serviço de impressão se registrou nesse serviço de pesquisa sob o grupo financeiro e é retornado um objeto para acessar esse serviço em particular. Então, o cliente usa o serviço de impressão diretamente, utilizando o objeto retornado (4). A figura também mostra outro serviço de impressão, que está no grupo admin. Também há um serviço de informações corporativas, que não está ligado a nenhum grupo em particular (e que pode ser registrado em todos os serviços de pesquisa).

Discussão sobre os serviços de descoberta de rede ◊ Os serviços de descoberta baseados apenas no alcance da rede que acabamos de descrever – serviços de descoberta de rede – avançam até certo ponto na solução do problema da associação. Existem implementações de diretório eficientes, incluindo aquelas que não contam com uma infra-estrutura. Em muitos casos, o número de clientes e serviços que podem ser atingidos por meio de uma sub-rede é administrável em termos de custos de computação e rede; portanto, a escala freqüentemente não é um problema. Descrevemos medidas para suportar a volatilidade dos sistemas.

Mas os serviços de descoberta de rede criam duas dificuldades quando vistos da perspectiva do princípio do limite: o uso de uma sub-rede e falta de adequação na maneira como os serviços são descritos.

A sub-rede pode ser uma aproximação insatisfatória de um espaço inteligente. Primeiro, a descoberta de rede pode incluir, por engano, serviços que não estão no espaço inteligente. Considere um quarto de hotel, por exemplo. As transmissões baseadas em sinais de radiofreqüência (RF), como 802.11 ou Bluetooth, normalmente penetram nas paredes dos quartos de outros hóspedes. Seguindo o exemplo do Jini, os serviços poderiam ser divididos logicamente por grupos – um grupo por quarto de hotel. Mas isso levanta a questão de como o quarto de hotel do usuário vai se tornar um parâmetro para o serviço de descoberta. Segundo, a descoberta de rede pode, por engano, não levar em conta serviços que estão "no" espaço inteligente, no sentido de serem qualificados para descoberta, mas que são colocados fora de sua sub-rede. O estudo de caso do Cooltown (veja a Seção 16.7.1) ilustra o modo como entidades não eletrônicas, como os documentos impressos em um espaço inteligente, podem ser associadas a serviços contidos fora do espaço inteligente.

Além disso, os serviços de descoberta de rede nem sempre estabelecem associações apropriadas, pois a linguagem usada para descrever os serviços pode ser inadequada em dois aspectos. Primeiro, a descoberta pode ser *frágil*: mesmo ligeiras variações no vocabulário da descrição do serviço usado por organizações distintas poderiam fazer com que ela falhasse. Por exemplo, o quarto de hotel tem um serviço chamado "Imprimir", enquanto o *laptop* do hóspede procura "Impressão". Variações no vocabulário da linguagem humana tendem a piorar esse problema. Segundo, pode haver *oportunidades perdidas* de acesso ao serviço. Por exemplo, existe uma "tela de fotografia digital" na parede do quarto de hotel, a qual exibirá instantâneos de férias no formato JPEG. A câmara do hóspede tem uma conexão sem fio e produz imagens nesse formato, mas ela não tem nenhuma descrição para o serviço – ela não foi atualizada com esse desenvolvimento relativamente recente. Portanto, a câmara é incapaz de tirar proveito dele.

16.2.2 Associação física

As deficiências dos sistemas de descoberta de rede podem ser resolvidas até certo ponto usando-se meios físicos, embora as soluções freqüentemente exijam um maior grau de envolvimento humano. Foram desenvolvidas as técnicas a seguir.

Interação humana na descoberta de escopo: ◊ Este caso é onde um ser humano fornece entrada para o dispositivo para configurar a abrangência da descoberta. Um exemplo simples disso seria digitar, ou selecionar, o identificador do espaço inteligente, como o número do quarto, no caso do hóspede de hotel. O dispositivo pode então usar o identificador como um atributo de grupo extra do serviço (como no Jini).

Percepção e canais fisicamente restritos para descoberta de escopo ◊ Uma possibilidade menos trabalhosa é o usuário utilizar um sensor em seu dispositivo. Por exemplo, o espaço inteligente po-

deria ter um identificador apresentado em símbolos de codificação de identificador, chamados *glyfos*, em documentos e superfícies no espaço – por exemplo, exibidos na tela da TV no quarto do hóspede de um hotel. O hóspede usa seu telefone com câmara, ou outro dispositivo de geração de imagens, para decodificar tal símbolo e o dispositivo usa o identificador resultante da maneira que descrevemos para entrada via intervenção humana. Outra possibilidade, para uso em espaços inteligentes onde estão disponíveis sinais de navegação via satélite, é utilizar um sensor para obter a posição do espaço inteligente em coordenadas de latitude e longitude, e enviar essas coordenadas para um serviço remoto conhecido, o qual retorna o endereço do serviço de descoberta local. Entretanto, dadas às imprecisões na navegação via satélite, esse método pode ser menos preciso na identificação do espaço inteligente, caso existam outros espaços próximos.

Outra técnica que evita a entrada humana é usar um *canal fisicamente restrito* (veja também a Seção 16.5.2) – um canal de comunicação que, até certo grau de aproximação, penetra apenas a extensão física do espaço inteligente. Por exemplo, no quarto do hóspede, a TV poderia estar tocando música de fundo em volume baixo, com uma codificação digital do identificador do quarto sobreposta como uma perturbação inaudível do sinal [Madhavapeddy *et al.* 2003]; ou poderia ser um transmissor infravermelho (uma baliza, ou, em inglês, *beacon*) no quarto que propagasse o identificador [Kindberg *et al.* 2002a]. Esses dois canais são significativamente atenuados pelos materiais nos limites do quarto (supondo que as portas e janelas estejam fechadas).

Associação direta ◊ O último conjunto de técnicas que vamos considerar aqui é o ser humano usando um mecanismo físico para associar dois dispositivos diretamente, sem usar um serviço de descoberta. Normalmente, é aí que os dispositivos envolvidos oferecem apenas um ou um pequeno conjunto de serviços selecionados pelo ser humano. Em cada uma das técnicas a seguir, o ser humano ativa o dispositivo que está transportando para conhecer o endereço de rede (por exemplo, Bluetooth ou endereço IP) de um dispositivo de destino.

Percepção de endereço: usar um dispositivo para perceber (sentir) diretamente o endereço de rede do dispositivo de destino. As possibilidades incluem: ler um *glyfo* no dispositivo, que codifica seu endereço de rede; ou trazer um dispositivo para muito próximo do outro e usar um canal sem fio de curto alcance para ler seu endereço. Dois exemplos desses canais de curto alcance são (1) *Near Field Communication* [www.nfc-forum.org] – um padrão para comunicação bidirecional via rádio que abrange várias faixas curtas, mas tem uma variante para apenas até cerca de 3 centímetros; ou (2) transmissões de sinal infravermelho de alcance muito curto.

Estímulo físico: usar um estímulo físico para fazer o dispositivo de destino enviar seu endereço. Um exemplo aqui é irradiar um raio laser com modulação digital (outro canal fisicamente restrito) para o dispositivo de destino [Patel e Abowd 2003], transmitindo assim seu endereço para o destino, o qual responde com o seu endereço.

Correlação temporal ou física: usar estímulos correlacionados temporal ou fisicamente para associar dispositivos. A especificação SWAP-CA [SWAP-CA 2002] para rede sem fio em um ambiente doméstico introduziu um protocolo, às vezes referido como protocolo de dois botões, para os seres humanos associarem dois dispositivos sem fio entre si. Cada dispositivo capta em um endereço *multicast* conhecido. Os usuários pressionam botões em seus respectivos dispositivos, mais ou menos simultaneamente, e os dispositivos enviam seus endereços de rede para o endereço *multicast*. É improvável que outra rodada desse protocolo ocorra ao mesmo tempo na mesma sub-rede. Portanto, os dispositivos são associados usando o endereço que chegar dentro de um pequeno intervalo de pressionamento dos botões. Existe um correlato físico interessante, se bem que raramente prático, para essa estratégia, no qual um usuário segura dois dispositivos na mesma mão e os sacode juntos [Holmquist *et al.* 2001]. Cada dispositivo tem um acelerômetro para captar seu estado de movimento. O dispositivo grava o padrão da vibração, calcula um identificador a partir dela e envia esse identificador por *multicast*, junto com seu endereço único, para um endereço *multicast* conhecido. Somente os dois dispositivos que experimentam exatamente esse padrão de aceleração – e dentro da faixa de comunicação direta – reconhecerão o identificador um do outro e, portanto, conhecerão o endereço um do outro.

16.2.3 Resumo e perspectiva

Esta seção descreveu o problema da associação de componentes em sistemas voláteis e algumas tentativas de resolver esse problema, variando da descoberta de rede até técnicas mais supervisionadas por seres humanos. Os sistemas móveis e ubíquos apresentam dificuldades únicas, pois eles são integrados ao nosso desorganizado mundo físico cotidiano, de espaços como ambientes doméstico e escritórios, tornando difícil encontrar soluções. Os seres humanos tendem a ter em mente fortes considerações territoriais e administrativas, quando julgam o que está em um espaço inteligente em particular e o que está fora dele. O princípio do limite diz que as soluções para o problema da associação precisam corresponder aos espaços físicos subjacentes até um grau que seja aceitável para os seres humanos. Vimos que, freqüentemente, certo grau de supervisão humana está envolvido, devido às deficiências dos sistemas de descoberta de rede. O estudo de caso do Cooltown (Seção 16.7) descreverá um modelo específico de envolvimento humano.

Ignoramos a escala como fator nas soluções do problema da associação, tendo por base que o mundo está dividido em espaços inteligentes, os quais, normalmente, têm tamanho administrável. Entretanto, existe pesquisa sendo feita sobre serviços de descoberta com escalabilidade – afinal, algumas aplicações poderiam considerar o planeta inteiro como um espaço inteligente. Um exemplo é o INS/Twine [Balazinska *et al.* 2002], que divide dados de diretório entre um conjunto de resolvedores *peer-to-peer*.

16.3 Interoperabilidade

Descrevemos as maneiras pelas quais dois ou mais componentes em um sistema volátil se associam e, agora, veremos a questão de como eles interagem. Os componentes se associam com base em certos atributos, ou dados, que um ou ambos possuem. Mas isso deixa as perguntas de qual protocolo eles usam para se comunicar e, em um nível mais alto, qual modelo de programação é mais conveniente para a interação entre eles. Esta seção trata dessas questões.

Os Capítulos 4 e 5 descreveram modelos de interação, incluindo várias formas de comunicação entre processos, invocação a métodos remotos e chamada de procedimentos remotos. Uma suposição implícita a alguns desses modelos é que os componentes que estão interagindo são feitos para trabalhar juntos em um sistema ou aplicativo específico e que alterações nesses componentes são um problema de configuração a longo prazo, ou uma condição de erro em tempo de execução a ser tratada ocasionalmente. Mas essas suposições não são válidas nos sistemas móveis e ubíquos. Felizmente, conforme explicaremos nesta seção, alguns dos métodos de interação dos Capítulos 4 e 5, além de alguns métodos novos, são mais convenientes para esses sistemas voláteis.

De preferência, um componente em um sistema móvel ou ubíquo poderia se associar com classes de serviços variadas e não apenas com um conjunto variável de instâncias da mesma classe de serviço. Isto é, é melhor evitar o problema da "oportunidade perdida", descrito na seção anterior, onde, por exemplo, uma câmara digital é incapaz de enviar suas imagens para uma tela de fotografia digital porque não pode interagir com o serviço de envio de fotos da tela.

Em outras palavras, um objetivo da computação móvel e ubíqua é que um componente deve ter uma chance razoável de interagir com um componente funcionalmente compatível, mesmo que este último esteja em um tipo de espaço inteligente diferente daquele para o qual foi originalmente desenvolvido. Isso exige algum acordo global entre os desenvolvedores de software. Dado o trabalho necessário para se chegar a um acordo, é melhor minimizar o que precisa ser concordado.

A principal dificuldade que a interação volátil enfrenta é a incompatibilidade da interface de software. Se, por exemplo, uma câmara digital espera invocar uma operação *pushImage*, e não existe tal operação na interface da tela de fotografia digital, então elas não podem interagir – pelo menos, não diretamente.

Existem duas estratégias principais para enfrentar esse problema. A primeira é permitir que as interfaces sejam heterogêneas, mas adaptá-las uma a outra. Por exemplo, se a tela de fotografia digital tivesse uma operação *sendImage* com os mesmos parâmetros e a mesma semântica de *pushImage*, então seria simples construir um componente que atuasse como *proxy* para a tela de fotografia digital, convertendo a ativação *pushImage* da câmara em uma ativação *sendImage* da tela.

Entretanto, é muito difícil realizar essa estratégia. Freqüentemente, a semântica das operações pode variar, assim como a sintaxe, e superar a incompatibilidade semântica é uma atividade difícil e propensa a erros, de modo geral. Então, há um problema de escala: se existirem N interfaces, então potencialmente N^2 adaptadores precisam ser escritos – e cada vez mais interfaces serão criadas com o passar do tempo. Além disso, há a questão de como os componentes vão adquirir adaptadores de interface convenientes à medida que são novamente associados em um sistema volátil. Os componentes (ou os dispositivos que os contêm) não podem vir previamente carregados com todos os N^2 adaptadores possíveis; portanto, o adaptador correto precisa ser determinado e carregado em tempo de execução. Apesar de todas essas dificuldades, existe pesquisa sendo feita sobre como tornar prática a adaptação da interface. Veja, por exemplo, Ponnekanti e Fox [2004].

Outra estratégia para a interação é obrigar as interfaces a terem sintaxe idêntica na mais ampla classe de componentes possível. Isso pode parecer irreal inicialmente, mas na verdade tem sido praticado amplamente e com êxito há várias décadas. O exemplo mais simples são os *pipes* do UNIX. Um *pipe* tem apenas duas operações, *leitura* e *escrita*, para o transporte de dados entre componentes (processos) em suas duas extremidades. Com o passar dos anos, os programadores de UNIX criaram muitos programas que podem ler e/ou escrever dados em um *pipe*. Devido as suas interfaces padronizadas, e às funcionalidades de processamento de texto genéricas, a saída de qualquer um desses programas pode alimentar a entrada de outro; usuários e programadores têm descoberto muitas maneiras úteis de combinar programas dessa forma – programas que foram escritos independentemente, sem conhecimento da funcionalidade específica dos outros programas.

Outro exemplo de mais sucesso ainda, de sistema que obtém um alto grau de interação por meio de uma interface fixa, é a web. O conjunto de métodos definidos pela especificação HTTP (veja a Seção 4.4) é pequeno e fixo; normalmente, um cliente web usa apenas as operações GET e POST para acessar um servidor web. A conseqüência das interfaces fixas é que um software relativamente estável – freqüentemente, o navegador – é capaz de interagir com um conjunto de serviços em expansão. O que muda entre os serviços é o tipo e os valores do conteúdo trocado e a semântica de processamento do servidor. Mas toda interação ainda é uma operação GET ou POST.

16.3.1 Programação orientada a dados para sistemas voláteis

Chamamos os sistemas que usam uma interface fixa de serviço, como os *pipes* do UNIX e a web, de *orientados a dados* (ou, equivalentemente, *orientados a conteúdo*). O termo foi escolhido para fazer distinção de *orientados a objetos*. Um componente em um sistema orientado a dados pode ser invocado por qualquer outro componente que conheça a interface fixa. Por outro lado, um objeto, ou um conjunto de procedimentos, tem uma interface de um conjunto amplo e variado de interfaces possíveis e só pode ser invocado pelos componentes que conheçam sua interface em particular. Distribuir e explorar um número desconhecido de definições de interface especializadas é uma possibilidade muito mais problemática do que publicar e usar uma especificação de interface, como a especificação HTTP. Isso ajuda a explicar porque o sistema distribuído mais usado e heterogêneo que conhecemos é a web, em vez de um conjunto de, digamos, objetos CORBA.

A flexibilidade dos sistemas orientados a dados tem um compromisso com a robusteza. Nem sempre faz sentido dois componentes em particular interagirem, e há pouca base para os programas verificarem a compatibilidade. Em um sistema orientado a objetos, ou a procedimentos, os programas podem pelo menos verificar se suas assinaturas de interface correspondem. Mas um componente orientado a dados só pode impor a compatibilidade verificando o tipo de dados enviado a ele. Ele deve fazer isso por intermédio de descritores de tipo de dados padronizados fornecidos como metadados (como os tipos MIME de conteúdo web) ou verificando os valores de dados passados a ele; por exemplo, os dados JPEG começam com informações de cabeçalho reconhecíveis.

Examinaremos agora alguns modelos de programação que têm sido usados para sistemas voláteis por causa de seus recursos de interação orientados a dados. Começaremos com dois modelos de interação entre componentes associados indiretamente: sistemas de eventos e espaços de tuplas. Vamos descrever dois projetos de interação entre dispositivos associados diretamente: *JetSend* e *Speakeasy*.

Hidden page

Hidden page

um espaço de tuplas, entretanto, pode durar mais que o componente lá colocado – e do que qualquer componente que a leia (em oposição a consumir destrutivamente). Essa persistência pode ser uma vantagem; por exemplo, as baterias da câmara de um usuário podem falhar após ele ter feito o *upload* de fotos no espaço de tuplas de um quarto de hotel, mas antes de tê-las transferido para outro dispositivo. Ao mesmo tempo, a persistência pode ser uma desvantagem: e se um conjunto de dispositivos puser tuplas em um espaço, mas elas nunca forem consumidas porque os componentes que as deveriam consumir se desconectaram? O conjunto de tuplas em tal espaço poderia crescer incontrolavelmente. Sem conhecimento global de um conjunto de componentes volátil seria impossível determinar quais tuplas seriam lixo.

Os projetistas do *event heap* reconheceram o problema da persistência dos iRooms. Eles optaram por permitir que as tuplas expirem (isto é, sejam coletadas como lixo), após estarem em um conjunto de eventos por um tempo especificado, o qual normalmente é escolhido de forma a corresponder ao período de tempo de uma interação humana. Isso evita, por exemplo, que um evento de "pausa", não consumido de um controle remoto, cause um incômodo quando um usuário tentar ver um vídeo no dia seguinte.

Interação direta com o dispositivo ◊ Os modelos de programação anteriores eram para a interação entre componentes associados indiretamente. O *JetSend* e o *Speakeasy* são sistemas projetados para interação entre dois dispositivos que um ser humano trouxe para associação direta.

JetSend: o protocolo *JetSend* [Williams 1998] foi projetado para interação entre aparelhos como câmaras, impressoras, scanners e TVs. O *JetSend* foi explicitamente projetado para ser orientado a dados, para que nenhum aparelho tivesse que ser carregado com *drivers* especializados, de acordo com os dispositivos específicos com os quais interagiria. Por exemplo, uma câmara *JetSend* pode enviar uma imagem para um dispositivo de consumo de imagem *JetSend*, como uma impressora ou TV, independentemente da funcionalidade específica do consumidor. A operação genérica central entre dispositivos *JetSend* conectados é *sincronizar* o estado que um apresenta com o estado do outro. Isso significa transferir o estado em um formato sobre o qual os dispositivos negociam. Por exemplo, um dispositivo de produção de imagem, como um scanner, poderia ser sincronizado com um dispositivo de consumo de imagem, como uma tela digital, consumindo uma imagem do produtor no formato JPEG, escolhido dentre vários formatos de imagem que o produtor poderia oferecer. O mesmo scanner poderia igualmente ser sincronizado com uma TV, talvez usando um formato de imagem diferente.

Os projetistas do *JetSend* reconheceram que sua operação de sincronização se beneficiava apenas de uma interação – basicamente, a transferência de dados – entre dispositivos heterogêneos. Isso levanta a questão de como obter interações mais complexas entre dispositivos específicos. Por exemplo, como deve ser feita a escolha entre o modo monocromático e colorido ao se transferir uma imagem para ser impressa? Por suposição, o dispositivo de origem não tem nenhum *driver* para uma impressora específica. E não é possível que esse dispositivo seja programado *a priori* com a semântica de qualquer dispositivo arbitrário (incluindo dispositivos que ainda serão inventados) com que ele possa se conectar. A resposta do *JetSend* para esse problema foi contar com o ser humano para selecionar as funções específicas do dispositivo de destino (digamos, uma impressora), usando uma interface com o usuário especificada por este, mas traduzidas em seu dispositivo de origem (digamos, uma câmara). É assim que a interação acontece rotineiramente na web, quando os usuários interagem com serviços altamente heterogêneos por intermédio de seus navegadores: cada serviço com que eles interagem envia sua interface, na forma de script de marcação, para o navegador, o qual o traduz para o usuário como um conjunto genérico de elementos de tela, e sem conhecimento da semântica específica do serviço. Os serviços web (veja o Capítulo 19) são uma tentativa de substituir o ser humano por programas, mesmo para interações complexas.

Speakeasy: o projeto *Speakeasy* [Edwards *et al.* 2002] aplicou posteriormente os mesmos princípios de projeto do *JetSend* na interação entre dispositivos, mas com uma diferença: utilizou código móvel. Existem dois motivos para se usar código móvel. O primeiro é que um dispositivo, como uma impressora, pode enviar qualquer interface com o usuário para um usuário de outro dispositivo, como um PDA. Uma implementação de código móvel de uma interface com o usuário pode realizar processamento local, como validação de dados de entradas, e pode fornecer modos de interação não disponíveis nas interfaces com o usuário que precisam ser especificadas em uma linguagem de marcação.

Hidden page

1999], mas, falando de modo geral, os dados fornecem dicas sobre o estado do sistema (porém, eles podem ser antigos e, portanto, não devem ser considerados precisos); e mais importante, os dados que compõem o *soft state* são atualizados automaticamente. Alguns sistemas de descoberta (Seção 16.2) exemplificam o uso de *soft state* para gerenciar o conjunto de entradas de registro do serviço. Primeiro, as entradas são apenas palpites – pode haver uma entrada para um serviço que desapareceu. Segundo, as entradas são atualizadas automaticamente um anúncio *multicast* dos serviços – para adicionar novas entradas e manter as já existentes atualizadas.

16.3.3 Resumo e perspectivas

Esta seção descreveu modelos de interação entre componentes em sistemas voláteis. Se cada espaço inteligente desenvolvesse sua própria interface de programação, as vantagens da mobilidade seriam limitadas. Se um componente não se originasse em determinado espaço inteligente, mas se movesse para lá, a única maneira pela qual ele poderia interagir com os serviços dentro do espaço inteligente seria por intermédio de um modo de adaptar sua interface espontaneamente à de sua vizinhança. Obter isso exigiria um suporte em tempo de execução muito sofisticado, que ainda não é realizado fora de uns poucos exemplos em atividades de pesquisa.

Uma estratégia diferente, descrita anteriormente por meio de vários exemplos, é a programação orientada a dados. Por um lado, a web tem mostrado a capacidade de ampliação e aplicação em massa desse paradigma. Por outro, não há nenhuma panacéia que resolva todos os problemas da interação de sistemas voláteis. Os sistemas orientados a dados trocam o acordo sobre o conjunto de funções em uma interface pelo acordo sobre os tipos de dados passados como argumentos para essas funções. Embora a XML (veja a Seção 4.3.3) às vezes seja apregoada como uma maneira de facilitar a interação de dados, permitindo que eles sejam "autodescritivos", na verdade ela apenas fornece uma estrutura para expressar estrutura e vocabulário. Em si mesma, a XML nada tem para contribuir com o que é um problema de semântica. Alguns autores consideram a "web semântica" [www.w3.org XX] como uma possibilidade de solução futura.

16.4 Percepção e reconhecimento de contexto

As seções anteriores se concentraram nos aspectos de volatilidade dos sistemas móveis e ubíquos. Esta seção se concentrará em outra característica importante desses sistemas: o fato de serem integrados com o mundo físico. Especificamente, ela considerará as arquiteturas para processamento de dados coletados a partir de sensores e os sistemas de reconhecimento de contexto que podem responder às suas circunstâncias físicas (percebidas). O sensoriamento ou percepção do local, um importante parâmetro físico, será examinado com mais detalhes.

Como os usuários e os dispositivos que estamos considerando freqüentemente são móveis, e como o mundo físico apresenta diferentes oportunidades de interações de locais em diferentes tempos, suas circunstâncias físicas são freqüentemente relevantes como determinantes para o comportamento do sistema. O sistema *Active Badge* fornece um exemplo histórico: a localização de um usuário – isto é, a localização do crachá que ele portava – foi usada, antes do surgimento dos telefones móveis, para identificar para qual telefone suas ligações deveriam ser direcionadas [Want *et al.* 1992]. O sistema de frenagem com reconhecimento de contexto de um carro poderia ajustar seu comportamento de acordo com a condição da estrada ser escorregadia ou não. Um dispositivo pessoal poderia utilizar recursos detectados automaticamente em seu ambiente, como uma tela para exibição.

O **contexto** de uma entidade (pessoa, lugar ou coisa, seja eletrônico ou não) é um aspecto de circunstâncias físicas, de relevância para o comportamento do sistema. Isso inclui valores relativamente simples, como a localização, a hora, a temperatura, a identidade de um usuário associado, por exemplo, operando um dispositivo, ou a presença e o estado de um objeto como uma tela de exibição. O contexto pode ser codificado e influenciado por meio de regras, como "Se o usuário for Fred e ele estiver em uma sala de reunião do IQ Labs, e se houver uma tela de exibição a 1m de distância, então mostre as informações do dispositivo na tela – a não ser que um funcionário que não seja do IQ Labs

esteja presente". O contexto também é usado para incluir atributos mais complexos, como a atividade do usuário. Por exemplo, um telefone com reconhecimento de contexto que precisa decidir se vai tocar, exige respostas para perguntas como: o usuário está em um cinema assistindo a um filme ou está falando com seus amigos antes da exibição?

16.4.1 Sensores

A determinação de um valor contextual começa com sensores, que são combinações de hardware e/ou software usadas para medir valores contextuais. Alguns exemplos são:

Localização, velocidade e orientação: unidades de navegação por satélite (por exemplo, o GPS) para fornecer coordenadas e velocidades globais; acelerômetros para detectar movimento; magnetômetros e giroscópios para fornecer dados de orientação.

Condições do ambiente: termômetros; sensores que medem a intensidade da luz; microfones de intensidade sonora.

Presença: sensores que medem a carga física, por exemplo, para detectar a presença de uma pessoa em uma cadeira ou andando em um pavimento; dispositivos que lêem identificadores eletrônicos em etiquetas colocadas próximas a eles, como os leitores *RFID* (*Radio Frequency IDentification*) [Want 2004] ou leitores de sinais infravermelhos, como aqueles usados para perceber etiquetas ativas; software usado para detectar pressionamentos de tecla em um computador.

As categorias anteriores são apenas exemplos de uso de sensores para propósitos em particular. Determinado sensor pode ser usado para diversos propósitos. Por exemplo, a presença de seres humanos pode ser detectada com microfones em uma sala de reunião; a localização de um objeto pode ser determinada pela detecção da presença de sua etiqueta ativa em um local conhecido.

Um aspecto importante de um sensor é o seu modelo de erro. Todos os sensores produzem valores com certo grau de erro. Alguns sensores, por exemplo, os termômetros, podem ser fabricados de modo que os erros caiam dentro de limites de tolerância bem conhecidos e com uma distribuição conhecida (por exemplo, gaussiana). Outros, como as unidades de navegação por satélite, têm modos de erro mais complicados que dependem de suas circunstâncias correntes. Primeiro, eles podem deixar de produzir um valor em certas circunstâncias. As unidades de navegação por satélite são dependentes do conjunto de satélites correntemente acessíveis sob visualização direta. Elas podem não funcionar dentro de prédios, cujas paredes podem atenuar demais os sinais do satélite para que a unidade funcione. Segundo, o cálculo da localização da unidade depende de fatores dinâmicos, incluindo as posições do satélite, a existência de obstruções próximas e das condições da ionosfera. Mesmo fora de prédios, uma unidade normalmente fornecerá diferentes valores em diferentes ocasiões, para a mesma localização, com apenas uma estimativa de melhor esforço da precisão corrente. Perto de prédios, ou de outros objetos altos, que obstruam ou reflitam sinais de rádio, satélites suficientes podem produzir uma leitura, mas a precisão pode ser baixa e a leitura pode ser totalmente espúria.

Uma maneira útil de expor o comportamento de erro de um sensor é citar a precisão que ele atinge para uma proporção de medidas especificada, por exemplo: "dentro da área determinada, a unidade de navegação por satélite se mostrou precisa dentro de um raio de 10m para 90% das medidas". Outra estratégia é expor um valor de confiança para uma medida em particular – um número (normalmente entre 0 e 1) escolhido de acordo com as incertezas encontradas na dedução da medida.

16.4.2 Arquiteturas de sensoriamento

Salber *et al.* [1999] identificam quatro desafios funcionais a serem superados no projeto de sistemas de reconhecimento de contexto:

Integração de sensores idiossincráticos. Alguns dos sensores necessários para a computação com reconhecimento de contexto são incomuns em sua construção e em suas interfaces de programação. Pode ser necessário um conhecimento especializado para implantá-los corretamente no cenário físico de interesse (por exemplo, onde os acelerômetros devem ser colocados para medir

Hidden page

atributo de contexto, ao mesmo tempo em que ocultam a complexidade dos sensores reais utilizados. Por exemplo, a Figura 16.5 mostra a interface para um elemento de contexto *IdentityPresence*. Ela fornece atributos contextuais para o software que faz *polling* nos elementos de contexto e também ativa *callbacks* quando a informação contextual muda (quando um usuário chega ou vai embora). Conforme indicado anteriormente, as informações de presença poderiam ser extraídas de qualquer uma das várias combinações de sensores em determinada implementação; a abstração permite que o programador do aplicativo ignore esses detalhes.

Os elementos de contexto são construídos a partir de componentes distribuídos. *Geradores* adquirem dados brutos de sensores, como os de pressão no piso, e fornecem esses dados para os elementos de contexto. Os elementos de contexto usam os serviços de *interpretadores*, os quais abstraem atributos contextuais dos dados de baixo nível do gerador, extraindo valores de nível mais alto, como a identidade de uma pessoa que esteja presente, a partir de seus passos característicos. Finalmente, elementos de contexto chamados *servidores* fornecem mais níveis de abstração, reunindo, armazenando e interpretando atributos contextuais de outros elementos de janela. Por exemplo, um elemento de contexto *PersonFinder* de um prédio poderia ser construído a partir dos elementos de contexto *IdentityPresence* para cada sala do prédio (Figura 16.6), os quais, por sua vez, poderiam ser implementados usando a interpretação do passo, a partir das leituras da pressão no piso, ou do reconhecimento da face, a partir de captura de vídeo. O elemento de contexto *PersonFinder* encapsula a complexidade de um prédio para o programador do aplicativo.

Vista em relação aos quatro desafios apresentados anteriormente, expressos pelos projetistas do *Context Toolkit*, essa arquitetura acomoda uma variedade de tipos de sensores; ela se destina à

Atributos (acessíveis por polling)	*Explicação*
Localização	Localização que o elemento de contexto está monitorando
Identidade	Identidade do último usuário detectado
Indicação de tempo (timestamp)	Hora da última saída
Callbacks	
PersonArrives (localização, identidade, indicação de tempo)	Disparado quando um usuário chega
PersonLeaves (localização, identidade, indicação de tempo)	Disparado quando um usuário vai embora

Figura 16.5 A classe de elementos de janela *IdentityPresence* do *Context Toolkit*.

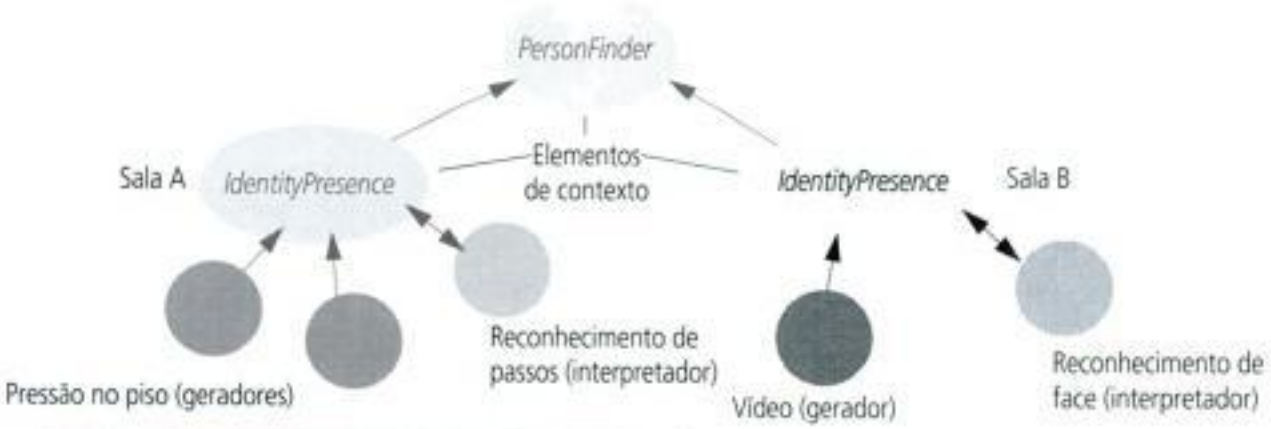


Figura 16.6 Um elemento de contexto *PersonFinder* construído usando elementos de contexto *IdentityPresence*.

produção de atributos contextuais abstratos a partir de dados brutos dos sensores; e, por intermédio de *polling*, ou de *callbacks*, um aplicativo de reconhecimento de contexto pode ficar sabendo das alterações em seu contexto. Entretanto, para soluções práticas, o *toolkit* apresenta limitações. Ele não ajuda os usuários e programadores a integrar sensores idiossincráticos e também não resolve nenhum dos problemas difíceis, inerentes aos processos de interpretação e combinação de um caso específico.

Redes de sensores sem fio ◊ Discutimos arquiteturas para aplicações nas quais o conjunto de sensores é relativamente estável – por exemplo, os sensores são instalados em salas de um prédio, freqüentemente com energia externa e com conexões a redes com fio (cabeadas). Agora, veremos os casos onde o conjunto de sensores forma um sistema volátil. Uma *rede de sensores sem fio* consiste em um número (normalmente grande) de pequenos dispositivos de baixo custo, os *nós*, cada um com recursos para sensoriamento, computação e comunicação sem fio [Culler *et al.* 2004]. Trata-se de um caso especial de *rede ad hoc*: os nós são fisicamente organizados de maneira mais ou menos aleatória, mas podem se comunicar por meio de várias passagens intermediárias (*hops*) entre seus pares. Um objetivo de projeto importante dessas redes é funcionar sem nenhum controle global; cada nó se inicializa sozinho, descobrindo seus vizinhos e comunicando-se apenas por meio deles. A Seção 3.5.2 descreveu as configurações *ad hoc* das redes 802.11, mas as tecnologias de potência mais baixa, como a ZigBee (IEEE 802.15.4), são mais relevantes aqui.

Um motivo pelo qual os nós não se comunicam em um único salto (*hop*) com todos os outros nós, mas, em vez disso, se comunicam diretamente apenas com os nós vizinhos, é que a comunicação sem fio tem um alto consumo de energia que aumenta com o quadrado do alcance do rádio. O outro motivo importante para restringir o alcance de rádios individuais é a redução da disputa pelo acesso a rede.

As redes de sensores sem fio são projetadas para serem adicionadas em um ambiente natural existente, ou construído, e para funcionar independentemente, isto é, sem haver uma infra-estrutura. Dado seu alcance de rádio e percepção limitados, os nós são instalados em uma densidade suficiente para tornar provável que a comunicação por vários *hops* seja possível entre qualquer par de nós e que os fenômenos significativos possam ser capturados (percebidos).

Por exemplo, considere dispositivos colocados por toda uma floresta, cuja tarefa fosse monitorar incêndios e outras condições ambientais, como a presença de animais. Esses nós são exatamente como os dispositivos apresentados na Seção 16.1.1. Cada um deles possui sensores, por exemplo, de temperatura, som e luz; funcionam com baterias; e se comunicam com outros dispositivos de maneira *peer-to-peer*, por intermédio de comunicação via rádio de curto alcance. A volatilidade deriva do fato de que esses dispositivos podem falhar devido ao esgotamento da bateria ou a acidentes, como os incêndios, e sua conectividade pode mudar devido a falhas de nó (os nós retransmitem pacotes entre outros nós), ou a condições ambientais que afetem a propagação do sinal de rádio.

Outro exemplo é onde os nós são colocados em veículos para monitorar o tráfego e as condições da estrada. Um nó que tenha observado uma condição ruim pode retransmitir informações sobre ela por meio de nós nos veículos que estão passando. Com conectividade global suficiente, esse sistema pode relatar o problema para outros motoristas próximos, que estejam indo nessa direção. Aqui, a volatilidade surge principalmente por causa do movimento dos nós, que muda rapidamente o estado da conectividade de cada nó com outros nós. Esse é um exemplo de *rede ad hoc móvel*.

Em geral, as redes de sensores sem fio são dedicadas a um propósito específico da aplicação, isto significa detectar certos *alarmes* – condições de interesse, como incêndios ou condições de estrada ruins. Normalmente, pelo menos um dispositivo mais poderoso, um *nó raiz*, é incluído na rede para prover comunicação de mais longo alcance com um sistema convencional reagindo adequadamente aos alarmes como, por exemplo, chamando os serviços de emergência quando há um incêndio.

Uma estratégia das arquiteturas de software para redes de sensores é tratá-las de modo semelhante às redes convencionais, separando a camada de rede das camadas mais altas. Em particular, é possível adaptar algoritmos de roteamento existentes ao grafo formado pelos nós quando eles descobrem dinamicamente que estão conectados por seus enlaces de rádio diretos, com cada nó capaz de atuar como roteador para as comunicações dos outros nós. O roteamento adaptativo, que tenta acomodar a volatilidade da rede, tem sido assunto de muito estudo e Milanovic *et al.* [2004] fornece uma visão geral de algumas técnicas.

Entretanto, limitar a preocupação com a camada de rede levanta questões. Primeiro, os algoritmos de roteamento adaptativos não são necessariamente ajustados para baixo consumo de energia (e largura de banda). Segundo, a volatilidade abala algumas das suposições das camadas tradicionais *acima* da camada de rede. Uma alternativa é considerar a estratégia inicial para arquiteturas de software para redes de sensores sem fio que é estimulada por dois requisitos principais: economia de energia e operação contínua, a despeito da volatilidade. Esses dois fatores levam a três características principais: processamento na rede, interligação em rede tolerante a rompimento e modelos de programação orientados a dados.

Processamento na rede: não apenas a comunicação sem fio é absolutamente dispendiosa em termos de consumo de energia, como também é relativamente cara em comparação ao processamento. Pottie e Kaiser [2000] calcularam o consumo de energia e descobriram que um processador de propósito geral poderia executar 3 milhões de instruções pela mesma quantidade de energia (3J) usada para transmitir, via rádio, 1 Kbit de dados a 100m. Portanto, em geral, o processamento é preferível em relação à comunicação: é melhor gastar alguns ciclos do processador para determinar se a comunicação é (ainda) necessária do que transmitir cegamente os dados percebidos. Na verdade, é por isso que os nós nas redes de sensores têm capacidade de processamento – caso contrário, eles poderiam consistir simplesmente em módulos de sensoriamento e comunicação, que enviariam os valores capturados para processamento nos nós raízes.

A frase *processamento na rede* se refere ao processamento dentro da rede de sensores; isto é, nos nós da rede. Os nós em uma rede de sensores executam tarefas como: agregar ou tirar a média de valores de nós vizinhos, para examinar valores de uma área, em vez de um único sensor; filtrar dados sem interesse ou repetidos; examinar dados para detectar alarmes; e ativar ou desativar sensores, de acordo com os valores que estejam sendo percebidos. Por exemplo, se sensores de luz de baixa potência indicassem a possível presença de animais (devido à projeção de sombras), então os nós próximos de onde as sombras foram projetadas poderiam ligar seus sensores de potência mais alta, como microfones, para tentar detectar sons do animal. Esse esquema permite que os microfones sejam desligados em outra situação, para economizar energia.

Interligação em rede tolerante a rompimento: o argumento do princípio fim-a-fim (Seção 2.2.1) tem sido um importante princípio para a arquitetura de sistemas distribuídos. Entretanto, nos sistemas voláteis, como as redes de sensores, pode ser que não exista continuamente nenhum caminho do princípio ao fim longo o suficiente para se obter uma operação como a movimentação de um volume de dados em um sistema. Os termos *Disruption Tolerant Networking* (interligação em rede tolerante a rompimento) e *Delay Tolerant Networking* (interligação em rede tolerante a atraso) são usados pelos protocolos para obter transferências de camada mais alta em redes voláteis (e normalmente heterogêneas) [www.dtnrg.org]. As técnicas se destinam não apenas às redes de sensores, mas também a outras redes voláteis, como os sistemas de comunicação interplanetária, necessários para a pesquisa espacial [www.ipnsig.org]. Em vez de contar com conectividade contínua entre dois pontos extremos fixos, a comunicação se torna oportunista: os dados são transferidos como e quando puderem ser, e os nós assumem sucessivas responsabilidades de mover dados em um estilo armazenar-e-encaminhar (*store-and-forward*), até que um objetivo do princípio fim-a-fim, como o transporte de um volume de dados tenha sido atingido. A unidade de transferência entre nós é conhecida como *pacote* [Fall 2003], o qual contém os dados do aplicativo da origem e dados descrevendo como fazer para gerenciá-los e processá-los no ponto extremo e nos nós intermediários. Por exemplo, um pacote pode ser transferido, *hop-by-hop*, usando protocolos de transporte confiáveis e quando um pacote tiver sido entregue, o nó destinatário assume a responsabilidade por sua distribuição subseqüente para o próximo *hop* – e assim sucessivamente. Esse procedimento não conta com nenhuma rota contínua; além disso, os nós com poucos recursos são desobrigados de armazenar os dados assim que os tiverem transferido para o próximo *hop*. Para evitar falha, os dados podem ser encaminhados de forma redundante para vários nós vizinhos.

Modelos de programação orientados a dados: passando para a interação nas camadas de aplicativo, foram desenvolvidas técnicas orientadas a dados, incluindo a *difusão direcionada* e o *processamento de consulta distribuído*, descritos em breve, para aplicações de redes de sensores. Essas

Hidden page

um deles, incluindo o uso de caminhos de forma redundante em caso de falha, ou a aplicação de heurísticas para encontrar um caminho de comprimento mínimo (Figura 16.7C).

O programador de aplicativo também pode fornecer um software chamado *filtro*, que é executado em cada um dos nós para interceptar o fluxo de dados correspondentes que passam pelo nó. Por exemplo, um filtro poderia suprimir alarmes duplicados de detecção de animais que derivassem de diferentes nós percebendo o mesmo animal (possivelmente o nó entre as fontes e o coletor na Figura 16.7C).

Outra estratégia orientada a dados para programação de redes de sensores é o processamento de consulta distribuído [Gehrke e Madden 2004]. Neste caso, entretanto, é usada uma linguagem como a SQL para declarar as consultas que serão feitas coletivamente pelos nós. A execução de uma consulta é normalmente processada no PC do usuário, ou na *estação de base* fora da rede, levando em conta todos os custos conhecidos, associados ao uso de nós sensores em particular. A estação de base distribui a consulta para os nós da rede, por rotas descobertas dinamicamente, levando em conta os padrões de comunicação que o processamento da consulta impõe, como o envio de dados para tirar a média de pontos de coleta. Assim como acontece na difusão direcionada, os dados podem ser agregados na rede para amortizar os custos da comunicação. Os resultados fluem de volta para a estação de base, para mais processamento.

16.4.3 Percepção de localização

De todos os tipos de percepção usados na computação ubíqua, a percepção da localização tem recebido a maior atenção. A localização é um parâmetro óbvio da computação móvel com reconhecimento de contexto. Parece natural fazer os aplicativos e dispositivos se comportarem de uma maneira que dependa de onde o usuário se encontre, como o telefone com reconhecimento de contexto. Mas a percepção da localização tem muitos outros usos, desde ajudar usuários na navegação em áreas urbanas ou rurais, até determinar rotas de rede pela geografia [Imielinski e Navas 1999].

Os sistemas de percepção da localização são projetados para obter dados sobre a posição dos objetos (seres vivos ou não) dentro de algum tipo de região de interesse. Aqui, nos concentraremos nas localizações dos objetos, mas algumas tecnologias também extraem valores de sua orientação e valores de ordem mais alta, como suas velocidades.

Uma distinção importante, especialmente quando se trata da privacidade, é se um objeto, ou um usuário, determina sua própria localização ou se algo a determina. Este último caso é conhecido como *rastreamento*.

A Figura 16.8 (baseada em uma figura semelhante que aparece em [Hightower e Borriello 2001]) mostra alguns tipos de tecnologias de localização e algumas de suas características. Uma das características é o mecanismo usado para inferir uma localização. Às vezes, esse mecanismo impõe limitações sobre onde a tecnologia pode ser implantada, por exemplo, se ela funciona internamente ou ao ar livre, e quais instalações são exigidas na infra-estrutura local. O mecanismo também é associado a uma precisão, dada na Figura 16.8 para uma ordem de grandeza. Em seguida, diferentes tecnologias produzem diferentes tipos de dados sobre a localização de um objeto. Finalmente, as tecnologias diferem nas informações, se houver, fornecidas sobre a entidade que está sendo localizada, o que é relevante para as preocupações dos usuários com a privacidade. Mais tecnologias são examinadas em Hightower e Borriello [2001].

O GPS (*Global Positioning System*) dos EUA é o exemplo mais conhecido de sistema de navegação via satélite – um sistema para determinar a posição aproximada de um *receptor*, ou *unidade*, a partir de sinais de satélite. Outros sistemas de navegação por satélite são o sistema russo GLONASS e o sistema europeu Galileo. O GPS, que só funciona ao ar livre devido à atenuação do sinal dentro de prédios, é usado rotineiramente em veículos e em dispositivos de mão (*handheld*) para navegação, e cada vez é mais empregado em aplicações menos convencionais, como a distribuição de mídia dependente da localização para pessoas em áreas urbanas [Hull *et al.* 2004]. A posição do receptor é calculada com relação a um subconjunto de 24 satélites que orbitam a terra em seis planos, quatro por plano. Cada satélite orbita a terra cerca de duas vezes por dia, divulgando a hora corrente de um relógio atômico a bordo e informações sobre suas localizações em uma gama de tempos (conforme julgados pelas observações de estações em terra). O receptor, cuja localização vai ser determinada,

Tipo	*Mecanismo*	*Limitações*	*Precisão*	*Tipo de dados de localização*	*Privacidade*
GPS	Triangulação de fontes de rádio do satélite	Somente ao ar livre (visibilidade do satélite)	1–10m	Coordenadas geográficas absolutas (latitude, longitude, altitude)	Sim
Balizamento de rádio	Transmissões de estações de base sem fio (GSM, 802.11, Bluetooth)	Áreas com cobertura sem fio	10m–1km	Proximidade de entidade conhecida (normalmente semântica)	Sim
Active Bat	Triangulação de rádio e ultra-som	Sensores montados no teto	10cm	Coordenadas relativas (sala)	Identidade do *morcego* (*bat*) revelada
Ultra Wide Band	Triangulação de recepção de pulsos de rádio	Instalações de receptor	15cm	Coordenadas relativas (sala)	Identidade da etiqueta revelada
Active Badge	Percepção de sinal infravermelho	Luz do sol ou luz fluorescente	Tamanho da sala	Proximidade de entidade conhecida (normalmente semântica)	Identidade do crachá revelada
Etiqueta de identificação automática	RFID, Near Field Communication, etiqueta visual (por exemplo, código de barras)	Instalações do leitor	1cm–10m	Proximidade de entidade conhecida (normalmente semântica)	Identidade da etiqueta revelada
Easy Living	Visão, triangulação	Instalações da câmara	Variável	Coordenadas relativas (sala)	Não

Figura 16.8 Algumas tecnologias de percepção de localização.

calcula sua distância a partir de cada um de vários satélites visíveis, usando a diferença entre a hora de chegada do sinal e a hora em que ele foi transmitido – isto é, a hora codificada no sinal – e uma estimativa da velocidade da propagação de rádio do satélite para a Terra. O leitor calcula então sua posição, usando um cálculo trigonométrico conhecido como *triangulação*. Para se obter uma posição, são exigidos pelo menos três satélites visíveis para o receptor. Se apenas três satélites estiverem visíveis, o leitor só poderá calcular sua latitude e longitude; com mais satélites visíveis, a altitude também poderá ser calculada.

Outro método de posicionamento que potencialmente funciona em uma área ampla, pelo menos em regiões densamente povoadas, é captar identificadores *balizados* (*beacons*), transmitidos periodicamente, a partir de estações de base sem fio, com alcance limitado. Os dispositivos podem comparar as intensidades do sinal como uma medida de qual estação está mais próxima. Cada uma das estações de base GSM para telefones móveis tem uma *identificação de célula*; os pontos de acesso 802.11 têm um identificador BSSID (*Basic Service Set Identifier*). As estações de base balizam seus identificadores, a não ser que sejam configuradas para não fazerem isso, por motivos de segurança. Uma *baliza* Bluetooth é um dispositivo que fornece seu identificador no caso de descoberta por outro dispositivo; na verdade, ele não divulga seu identificador.

O balizamento não determina a posição de uma entidade em si, mas apenas sua proximidade de outra entidade. Se o balizamento da posição da entidade for conhecido, então a posição da entidade de destino será conhecida dentro do alcance do rádio. Portanto, o posicionamento absoluto exige pesquisar o identificador balizado em um banco de dados. As organizações que gerenciam as fontes de rádio, como as operadoras de telecomunicações, normalmente não revelam (ou não revelarão) os detalhes de

suas localizações. Mas existem alguns projetos comunitários por meio dos quais os próprios usuários fornecem os detalhes da localização.

A proximidade pode ser uma propriedade útil em si mesma. Por exemplo, usando a proximidade é possível criar aplicativos com reconhecimento de localização que são disparados pelo retorno a um local visitado anteriormente. Por exemplo, um usuário esperando em uma estação de trem poderia criar um alerta lembrando-o de comprar um novo passe mensal, quando entrar nas proximidades da estação de trem (isto é, quando seu dispositivo receber o mesmo identificador balizado), no primeiro dia do mês. O Bluetooth, uma tecnologia de rádio alternativa, tem a propriedade interessante de que algumas balizas – por exemplo, aqueles integrados com telefones móveis – são eles próprios móveis. Isso pode ser útil. Por exemplo, passageiros habituais de um trem poderiam receber dados das pessoas com quem viajam freqüentemente – "estranhos conhecidos" –, fornecidos pelos seus telefones móveis.

Voltando às formas mais definidas de posicionamento, o GPS extrai as coordenadas *absolutas* (isto é, globais) de um objeto ao ar livre. Em contraste, o sistema *Active Bat* [Harter *et al.* 2002] foi projetado para produzir a localização de um objeto ou ser humano em interiores, em coordenadas *relativas* – isto é, com relação à sala que contém o objeto (Figura 16.9). O sistema *Active Bat* fornece uma precisão de cerca de 10 cm. A localização interior relativamente precisa é útil para aplicações como a detecção de qual tela um usuário móvel está próximo e para "teletransportar" a área de trabalho de seu PC para ela, usando o protocolo VNC (veja "Implementações de clientes "leves"", na Seção 2.2.3). Um *morcego* (*bat*) é um dispositivo ligado a um usuário ou objeto, cuja localização deve ser encontrada, e que recebe sinais de rádio e emite sinais de ultra-som. O sistema conta com uma grade de receptores de ultra-som em locais conhecidos no teto, ligados com fios a uma *estação de base*. Para localizar um morcego, a estação de base emite simultaneamente um sinal de rádio para ele, contendo seu identificador, e um sinal pelo fio para os receptores de ultra-som montados no teto. Quando o morcego com o identificador dado recebe o sinal da estação de base, ele emite um pulso de ultra-som curto. Quando um receptor no teto recebe o sinal da estação de base, ele inicia um temporizador. Como a velocidade da propagação eletromagnética é muito maior do que a velocidade do som, a emissão do pulso de ultra-som e a partida do temporizador são efetivamente simultâneas. Quando um receptor no teto recebe o pulso de ultra-som correspondente (o do morcego), ele lê o tempo decorrido e o encaminha para a estação de base, a qual utiliza uma estimativa da velocidade do som para deduzir a distância do morcego até o receptor. Se a estação de base receber distâncias de pelo menos três receptores de ultra-som não colineares, poderá calcular a posição do *morcego* no espaço tridimensional.

A Ultra Wide Band (UWB) é uma técnica para propagar dados em altas taxas de transmissão (100 Mbps ou mais) em raios curtos de ação (até 10 m). Os bits são propagados em potência muito baixa, mas em um espectro de freqüência muito amplo, usando pulsos estreitos – da ordem de 1ns de largura. Sendo conhecidos o tamanho e o formato do pulso, é possível medir tempos de "viagem" com grande precisão. Organizando-se receptores no ambiente e usando triangulação, como nas tecnologias anteriores, é possível determinar as coordenadas de uma etiqueta UWB com uma precisão de cerca de15 cm. Ao contrário das tecnologias anteriores, os sinais UWB se propagam através de paredes e outros objetos típicos encontrados em um ambiente. Seu baixo consumo de energia é outra vantagem.

Figura 16.9 Localizando um morcego ativo (*active bat*) dentro de uma sala.

O GPS, os *Active Bats* e a UWB, fornecem dados sobre a localização física de um objeto: suas coordenadas em uma região física. Uma vantagem de conhecer a localização física é que, por meio de bancos de dados que incluam *sistemas de informação geográfica* (GIS) e *modelos do mundo* de espaços construídos, uma única localização pode ser relacionada com muitos tipos de informações sobre o objeto, ou seu relacionamento com outros objetos. Entretanto, a desvantagem é o trabalho exigido para produzir e manter esses bancos de dados, os quais podem experimentar altas taxas de mudança.

Em contraste, o sistema *Active Badge* (Seção 16.1) produz a localização semântica de um objeto: o nome ou a descrição da localização. Por exemplo, se um crachá é percebido por um receptor de sinal infravermelho na sala 101, então, a localização desse crachá é determinada como sendo a "Sala 101". (Ao contrário da maioria dos sinais de rádio, as paredes de uma construção atenuam fortemente os sinais infravermelhos; portanto, é improvável que o crachá esteja fora da sala.) Esses dados não nos informam nada, explicitamente, sobre a localização no espaço, mas fornece aos usuários informações relacionadas ao seu conhecimento do mundo em que vivem. Em contraste, a latitude e a longitude do mesmo lugar *51° 27.010 N 002° 37.107 W* são úteis para, digamos, calcular distâncias para outros lugares; mas é difícil para seres humanos trabalharem com elas. Note que as balizas (*beacons*) – que invertem as tecnologias *Active Badge*, colocando o receptor no destino a ser localizado, em vez de colocar na infra-estrutura – podem ser usados para fornecer localizações semânticas ou físicas (muito aproximadas).

Os *Active Badges* são uma forma especializada de etiquetas de identificação automática: identificadores legíveis eletronicamente, normalmente projetados para aplicações industriais de massa. As etiquetas de identificação automática incluem RFID [Want 2004], Near Field Communication (NFC) [www.nfc-forum.org], glifos ou outros símbolos visuais, como os códigos de barras – especialmente aqueles projetados para serem lidos à distância por câmaras [de Ipiña *et al.* 2002]. Essas etiquetas são anexadas no objeto cuja localização deve ser determinada. Quando lida por um leitor com alcance limitado, e em um local conhecido, a localização do objeto de destino se torna conhecida.

Finalmente, o projeto *Easy Living* [Krumm *et al.* 2000] usou algoritmos de visão para localizar um objeto visto por várias câmaras, como um ser humano. Um objeto de destino pode ser localizado se puder ser reconhecido por uma câmara colocada em um local conhecido. Em princípio, com várias câmaras colocadas em locais conhecidos, as diferenças entre as aparências do objeto em suas imagens podem ser usadas para determinar sua localização física.

Conforme demonstrado no estudo de caso do Cooltown (Seção 16.7.2), algumas das tecnologias de localização anteriores – especialmente as etiquetas de identificação automática e as balizas de sinal infravermelho – também podem ser usadas para dar acesso a informações e serviços relativos às entidades nas quais estão anexadas, por meio dos identificadores que tornam disponíveis.

Comparando as tecnologias anteriores com relação à privacidade, a solução do GPS fornece privacidade absoluta: em nenhum ponto na operação do GPS as informações sobre o dispositivo receptor são transmitidas. As balizas podem fornecer privacidade absoluta, mas isso depende de como elas são usadas. Se um dispositivo simplesmente captar as balizas e nunca se comunicar com a infra-estrutura de outra forma, então ele manterá a privacidade. Em contraste, as outras são tecnologias de rastreamento. Os *Active Bats*, a UWB, os *Active Badges* e os métodos de identificação automática produzem um identificador para a infra-estrutura, em um local e momento conhecidos. Mesmo que o usuário associado não revele sua identidade, é possível inferi-la. As técnicas de visão do *Easy Living* contam com o reconhecimento de usuários para localizá-los e, assim, a identidade do usuário é revelada muito mais diretamente.

Arquiteturas para percepção de localização ◊ Duas das principais características exigidas para os sistemas de localização são: (1) generalidade com relação aos tipos de sensores usados para a percepção da localização e (2) mudança de escala com relação ao número de objetos a serem localizados e à taxa de eventos de atualização de posicionamento que ocorrem quando os objetos móveis, como pessoas e veículos, mudam suas localizações. Pesquisadores e desenvolvedores têm produzido arquiteturas modestas para percepção da localização – em espaços inteligentes individuais, como salas, prédios ou ambientes naturais cobertos por redes de sensores; e para sistemas de informação geográficas, destinados a cobrir grandes áreas e incluir as localizações de muitos objetos.

A *pilha de localização* [Hightower *et al.* 2002, Graumann *et al.* 2003] tem como objetivo atender o requisito da generalidade. Ela divide os sistemas de percepção da localização para espaços inteligen-

tes individuais em camadas. A *camada de sensores* contém *drivers* para extrair dados brutos de uma variedade de sensores de localização. Então, a *camada de medidas* transforma esses dados brutos em tipos de medida comuns, incluindo distância, ângulo e velocidade. A *camada de fusão* é a mais baixa disponível para os aplicativos. Ela combina as medidas de diferentes sensores (normalmente, de tipos diferentes), para inferir a localização de um objeto e fornecê-la por meio de uma interface uniforme. Como os sensores produzem dados incertos, as inferências da camada de fusão são probabilísticas. Fox *et al.* [2003] examinam algumas das técnicas bayesianas disponíveis. A *camada de arranjos* deduz os relacionamentos entre os objetos, como o fato de eles estarem no mesmo lugar. Acima dessas estão camadas para combinar dados de localização com dados de outros tipos de sensores, para determinar atributos contextuais mais complexos, como se um grupo de pessoas localizadas em uma casa estão todas dormindo.

A escalabilidade é uma preocupação maior nos sistemas de informação geográfica. Consultas *espaço-temporais*, como "Quem esteve neste prédio nos últimos 60 dias?", "Alguém está me seguindo?" ou "Quais objetos móveis nesta região correm maior perigo de colidir?", ilustram essa necessidade. O número de objetos – em particular, o número de objetos móveis – a serem localizados e o número de consultas concorrentes pode ser grande. Além disso, no último desses exemplos de consulta, é exigida sensibilidade em tempo real. A estratégia óbvia para tornar os sistemas de localização capazes de ser escaláveis é dividir a região de interesse em sub-regiões, recursivamente, usando estruturas de dados como *quadtrees*. Tal indexação de bancos de dados espaciais e temporais é uma área de pesquisa ativa.

16.4.4 Resumo e perspectiva

Esta seção descreveu algumas das infra-estruturas que foram projetadas para computação com reconhecimento de contexto. Nos concentramos principalmente nas maneiras pelas quais os sensores são aproveitados para produzir os atributos contextuais dos quais os aplicativos dependem para definir seu comportamento. Vimos as arquiteturas para conjuntos de sensores relativamente estáticos e arquiteturas para redes de sensores altamente voláteis. Finalmente, descrevemos algumas tecnologias para o caso particularmente importante da percepção da localização.

Por meio do reconhecimento de contexto, integramos o mundo físico cotidiano com sistemas de computador. Um problema importante que permanece é que, comparados ao entendimento sutil que os seres humanos têm de seu mundo físico, os sistemas que descrevemos são bastante toscos. Não apenas os sensores (pelo menos, aqueles baratos o suficiente para serem amplamente distribuídos) são inevitavelmente imprecisos, como também o estágio final da produção semântica de informações ricas e precisas, a partir dos dados brutos do sensor, é extremamente difícil. O mundo da robótica (que envolve acionamento, um assunto que ignoramos, além da percepção) vem tentando resolver essa dificuldade há muitos anos. Em domínios rigorosamente restritos, como um aspirador de pó doméstico ou a produção industrial, os robôs podem funcionar razoavelmente bem. Mas a generalização desses domínios permanece indefinida.

16.5 Segurança e privacidade

Os sistemas voláteis levantam muitas questões tanto sobre a segurança como sobre a privacidade. Primeiro, os usuários e administradores de sistemas voláteis exigem segurança para seus dados e recursos (confidencialidade, integridade e disponibilidade). Entretanto, conforme mencionamos ao descrevermos o modelo de sistemas voláteis, na Seção 16.1, a confiança – a base de toda a segurança – é freqüentemente diminuída nos sistemas voláteis. Ela é diminuída porque os principais, cujos componentes interagem espontaneamente, podem ter pouco (se houver) conhecimento anterior uns dos outros e podem não ter um terceiro participante confiável em comum. Segundo, muitos usuários se preocupam com sua *privacidade* – grosso modo, sua capacidade de controlar a acessibilidade das informações sobre eles mesmos. Mas a privacidade está potencialmente mais ameaçada do que nunca, devido à percepção nos espaços inteligentes pelos quais os usuários passam.

Hidden page

Hidden page

como a associação é espontânea, nenhum dos dispositivos (ou seus usuários) compartilha um segredo com o outro, não possui a chave pública do outro e os dispositivos não têm acesso a um terceiro participante confiável. Mesmo que exista um terceiro participante confiável, ele pode estar off-line. Um invasor pode tentar intrometer-se em *W* e reproduzir e forjar mensagens. Em particular, um invasor pode tentar lançar um ataque do homem no meio (Seção 7.1.1).

Uma solução para esse problema seria permitir que um visitante estabelecesse uma conexão segura com um serviço de projetor ou impressora; os colegas em uma conferência poderiam trocar um documento com segurança entre seus dispositivos portáteis; a enfermeira poderia conectar um monitor de batimentos cardíacos sem fio com segurança em uma unidade de registro de dados na cama do paciente.

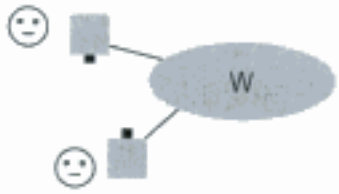
Nenhum volume de comunicação em *W* permitirá a troca de chave segura por si só; portanto é exigida uma comunicação fora da faixa de comunicação. Em particular, o método padrão para estabelecer uma chave em nível de enlace entre dois dispositivos conectados por Bluetooth conta com ações de um ou mais usuários. Um string de dígitos, escolhido em um dispositivo, deve ser digitado por um usuário no outro dispositivo. Mas esse método freqüentemente não é executado com segurança, pois strings de dígitos simples e curtos, como "0000", tendem a ser usados, os quais os invasores podem descobrir por meio de busca exaustiva.

Outra estratégia para resolver o problema da associação segura é usar um canal secundário com certas propriedades físicas. Especificamente, a propagação de sinais por esse canal secundário é restrita em ângulo, alcance ou temporização (ou uma combinação deles). Para um primeiro grau de aproximação, podemos inferir propriedades sobre o remetente ou receptor de mensagens em tais canais, que nos permitam estabelecer associação segura com um dispositivo fisicamente demonstrável, conforme mostraremos em breve. Kindberg *et al.* [2002b] os chamam de *canais fisicamente restritos*, o termo que usaremos aqui; Balfanz *et al.* [2002] se referem aos *canais de localização limitada*; Stajano e Anderson [1999] exploraram pela primeira vez tal canal secundário, na forma de contato físico. Apresentamos alguns exemplos desses canais para os propósitos da associação física de dispositivos, na Seção 16.2.2.

Em um cenário, um dos dispositivos gera uma nova chave de sessão e a envia para o outro por meio de um *canal de recepção restrita*, que fornece certo grau de sigilo – isto é, ele restringe os dispositivos que podem receber a chave. Alguns exemplos de tecnologias de canais de recepção restrita são:

- Contato físico. Cada dispositivo tem terminais para conexão elétrica direta [Stajano e Anderson 1999]. Veja a Figura 16.10.
- Infravermelho. Feixes de raios infravermelhos podem se tornar direcionais dentro de cerca de 60 graus e são bastante atenuados por paredes e janelas. Um usuário pode "emitir" uma chave para o dispositivo receptor exigido, por uma distância de até cerca de um metro [Balfanz *et al.* 2002].
- Áudio. Os dados podem ser transmitidos como modulações de sinal de áudio, como música tocando suavemente em toda uma sala, mas com pouco, ou nenhum, alcance fora dela [Madhavapeddy *et al.* 2003].

1. Nova chave secreta *K* trocada pelo contato físico



2. Os dispositivos se comunicam usando um canal seguro criado sobre *W* usando *K*

Figura 16.10 Associação segura de dispositivos usando contato físico.

- Laser. Um usuário aponta um feixe estreito de raio laser transportando dados, para um receptor no outro dispositivo [Kindberg e Zhang 2003a]. Este método permite mais precisão do que as outras técnicas de longo alcance.
- Código de barras e câmara. Um dispositivo exibe a chave secreta como um código de barras (ou outra imagem que possa ser decodificada) em sua tela, o qual o outro dispositivo – equipado com uma câmara, como um telefone com câmara – lê e decodifica. A precisão deste método é inversamente relacionada à distância entre os dispositivos.

Em geral, os canais fisicamente restritos fornecem apenas um grau de segurança limitado. Um invasor com um receptor suficientemente sensível pode intrometer-se no sinal infravermelho ou de áudio; um invasor com uma câmara poderosa pode ler um código de barras, mesmo em uma tela pequena. O raio laser está sujeito à dispersão na atmosfera, embora as técnicas de modulação de quantum possam tornar o sinal disperso inútil para um invasor [Gibson *et al*. 2004]. Entretanto, quando as tecnologias são implantadas em circunstâncias apropriadas, os ataques exigem um esforço considerável e a segurança obtida pode ser boa o bastante para os propósitos normais.

Uma segunda estratégia para troca segura de uma chave de sessão é usar um canal restrito para autenticar fisicamente a chave pública de um dispositivo, o qual a envia para o outro dispositivo. Então, os dispositivos empregam um protocolo padrão para trocar uma chave de sessão usando a chave pública autenticada. É claro que esse método presume que os dispositivos sejam poderosos o suficiente para realizar criptografia de chave pública.

A maneira mais simples de autenticar a chave pública é fazer com que o dispositivo a transmita por um canal de *envio restrito*, o qual permite um usuário autenticar a chave conforme foi derivada desse dispositivo físico. Existem várias maneiras de implementar canais de envio restrito convenientes. Por exemplo, o contato físico fornece um canal de envio restrito, pois apenas um dispositivo conectado diretamente pode transmitir através dele. O Exercício 16.14 convida o leitor a considerar quais das outras técnicas de canais de recepção restrita descritas anteriormente também fornecem canais de envio restrito. Além disso, é possível implementar um canal de envio restrito usando um canal de recepção restrita ou *vice-versa*. Veja o Exercício 16.15.

Uma terceira estratégia utilizando canais fisicamente restritos é a que faz os dispositivos trocarem uma chave de sessão de forma otimista, mas sem segurança, e então usar um canal fisicamente restrito para *validar* a chave – isto é, usar um canal fisicamente restrito para verificar se a chave pertence unicamente a fonte física exigida.

Primeiramente, consideraremos como trocamos uma chave de sessão espontaneamente, mas possivelmente com o principal errado, e passaremos a examinar algumas tecnologias para validar a troca. Se a validação falhar, então o processo poderá ser repetido.

Na Seção 16.2.2, descrevemos as técnicas físicas e intermediadas por seres humanos para associar dois dispositivos, como o protocolo de dois botões, no qual os dispositivos trocam seus endereços de rede quando seres humanos pressionam seus botões mais ou menos simultaneamente. É simples adaptar esse protocolo de modo que os dispositivos também troquem chaves de sessão, usando o protocolo de Diffie–Hellman [Diffie e Hellman 1976]. Mas, como se vê, esse método não é seguro: ainda é possível que grupos separados de usuários associem dispositivos erroneamente, por acidente, executando o protocolo concorrentemente, e que pessoas maldosas lancem ataques do homem no meio.

As técnicas a seguir nos permitem validar uma chave antes de usá-la. Elas envolvem canais de envio restrito, embora canais de recepção restrita também possam ser usados (veja o Exercício 16.15). Uma propriedade do protocolo de Diffie–Hellman é que um homem do meio não pode (exceto com probabilidade desprezível) trocar a mesma chave com cada dispositivo; portanto, podemos validar a associação comparando os códigos de resumo (*hashing*) seguros das chaves obtidas pelos dois dispositivos, após executar o protocolo de Diffie–Hellman (Figura 16.11).

- Códigos de resumo exibidos. Stajano e Anderson mostraram que cada dispositivo poderia exibir o código de resumo de sua chave pública como caracteres hexadecimais ou de alguma outra forma que os seres humanos poderiam comparar. Entretanto, argumentaram que esse tipo de envolvimento humano é muito propenso a erros. O método anterior do código de barras seria mais confiável. Esse método é outro exemplo do uso de um canal de envio restrito: o

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

ser reescrito e a atividade de transcodificação com uso intenso do computador pode ser realizada em hardware com capacidade de mudança de escala conveniente, como agregados de computadores montados em um *rack*, para manter as latências dentro de limites aceitáveis.

Quando se trata de sistemas voláteis, como os espaços inteligentes, precisamos rever algumas das suposições feitas para a web e para outra adaptação na escala da Internet. Os sistemas voláteis são mais exigentes, no sentido de que podem pedir adaptação entre qualquer par de dispositivos associados dinamicamente e, assim, a adaptação não está restrita aos clientes de um serviço particular da infra-estrutura. Agora existem potencialmente muito mais provedores cujo conteúdo precisa ser adaptado. Além disso, esses provedores também podem ser pobres demais em recursos para realizar sozinhos certos tipos de adaptação.

Uma implicação é que os espaços inteligentes devem fornecer *proxies* em sua infra-estrutura para adaptar conteúdo entre os componentes voláteis que hospedam [Kiciman e Fox 2000; Ponnekanti *et al*. 2001]. A segunda implicação é a necessidade de examinar melhor quais tipos de adaptação de conteúdo podem e devem ser realizados em dispositivos pequenos – em particular, a compactação é um exemplo importante.

Mesmo que exista um poderoso *proxy* de adaptação na infra-estrutura, um dispositivo ainda precisa enviar seus dados para esse *proxy*. Discutimos anteriormente como a comunicação é dispendiosa, comparada com o processamento. Em princípio, pode ser mais eficiente, com relação à energia, compactar os dados antes da transmissão. Entretanto, o padrão de acessos à memória feitos durante a compactação tem forte efeito sobre o consumo de energia. Barr e Asanovic [2003] mostram que, em termos de energia, compactar os dados usando implementações padrão antes de enviá-los, mas que uma otimização cuidadosa dos algoritmos de compactação e descompactação, especialmente com relação aos padrões de acesso à memória, pode levar a economia global de energia, comparada à transmissão de dados não compactados.

16.6.2 Adaptação a sistema de recursos variáveis

Embora os recursos de hardware, como o tamanho da tela, sejam heterogêneos entre os dispositivos, pelo menos eles são estáveis e conhecidos. Entretanto, as aplicações também contam com recursos que estão sujeitos à mudança em tempo de execução e que podem ser difíceis de prever, como a energia e a largura de banda da rede disponíveis. Nesta subseção, discutiremos técnicas para tratar com essas alterações de recursos em tempo de execução. Discutiremos o suporte do sistema operacional para aplicações executadas em sistemas voláteis e o suporte necessário à infra-estrutura do espaço inteligente para melhorar os recursos disponíveis para estas.

Suporte do sistema operacional para adaptação a recursos voláteis ◊ Satyanarayanan [2001] descreve três estratégias de adaptação. Uma estratégia é fazer com que as aplicações solicitem e obtenham reservas de recurso. Embora a reserva de recurso possa ser conveniente para as aplicações (Capítulo 17), às vezes é difícil obter garantias de qualidade de serviço satisfatórias nos sistemas voláteis e, às vezes, elas são até impossíveis, como nos casos de esgotamento da energia. Uma segunda estratégia é notificar o usuário sobre alterações na disponibilidade de recursos para que ele possa agir de acordo com a aplicação. Por exemplo, se a largura de banda se tornar baixa, o usuário de um reprodutor de vídeo pode operar um controle deslizante para trocar a taxa de quadros ou a resolução. A terceira estratégia é o sistema operacional notificar a aplicação sobre mudanças nas condições do recurso e esta se adaptar de acordo com suas necessidades específicas.

A arquitetura Odyssey [Noble e Satyanarayanan 1999] fornece suporte de sistema operacional para aplicações que se adaptam às mudanças nos níveis disponíveis de recursos, como a largura de banda da rede. Por exemplo, se a largura de banda cai, um reprodutor de vídeo pode trocar para um fluxo de vídeo com menos cores ou ajustar a resolução ou, ainda, a taxa de quadros. Na arquitetura Odyssey, as aplicações gerenciam tipos de dados como vídeo e imagens e, quando as condições do recurso mudam, eles ajustam a *fidelidade* – a qualidade específica do tipo – com que esses dados são representados. Um componente de sistema chamado *viceroy* divide os recursos totais do dispositivo entre cada uma das várias aplicações que estão sendo executadas nele. Em dado momento, cada aplicação é executada com uma *janela de tolerância* para as alterações nas condições do recurso. A janela de tolerância é

Hidden page

Hidden page

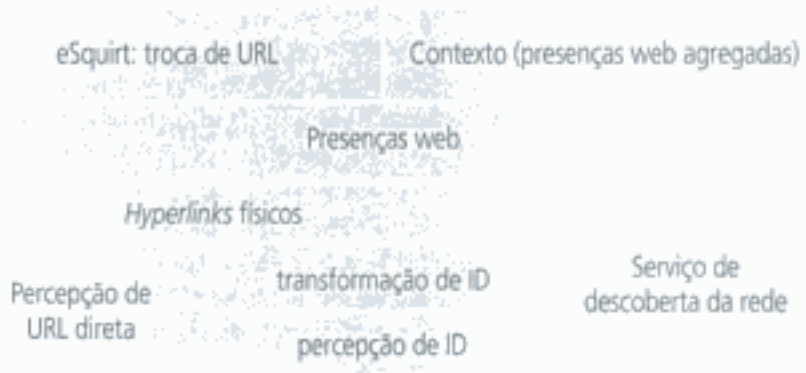


Figura 16.12 Camadas do projeto Cooltown.

dor web. Se a coisa é um dispositivo, então seu URL é o do serviço que ele implementa. Por exemplo, uma "rádio da Internet" é um dispositivo tocando música que hospeda sua própria presença web. Um usuário que tenha descoberto o URL de uma rádio da Internet recupera uma página web com controles que permitem "sintonizar" em uma fonte de transmissão da Internet, ajustar suas configurações, como o volume, ou fazer *upload* do arquivo de som do próprio usuário. Mas até coisas não eletrônicas podem ter uma presença web – um recurso web associado a uma coisa, mas hospedado por um servidor web em outro lugar. Por exemplo, a presença web de um documento impresso poderia ser seu documento eletrônico correspondente: em vez de fazer a fotocópia do documento impresso (com uma conseqüente redução na qualidade), um usuário pode descobrir sua presença web a partir de um artefato físico – conforme explicaremos na Seção 16.7.2 – e solicitar uma nova impressão. A presença web de um CD de música poderia ser algum conteúdo digital associado, como clipes de música e fotografias extras, hospedados na coleção de mídia pessoal de seu proprietário.

Pessoas. As pessoas se tornam presentes na web oferecendo *home pages* globais com serviços para facilitar a comunicação com elas e oferecendo informações sobre seu contexto corrente. Por exemplo, usuários sem telefones móveis poderiam tornar disponível o número do telefone local, por intermédio de sua presença web – um valor que sua presença web atualiza automaticamente, à medida que eles mudam. Mas eles também poderiam escolher para sua presença web um registro explícito de sua localização corrente por meio de um *link* para a presença web do lugar no qual estão presentes fisicamente.

Lugares. Os lugares são espaços inteligentes, para usar a terminologia deste capítulo. Os lugares se tornam presentes na web registrando a presença de pessoas e coisas que estão dentro deles – e até a presença web de lugares aninhados ou relacionados de alguma forma – em um serviço de diretório específico do lugar (Seção 9.3). O diretório de um lugar também contém informações relativamente estáticas, como uma descrição das propriedades físicas e da função do lugar. O serviço de diretório permite que os componentes descubram e, assim, interajam com o conjunto dinâmico de presenças web dentro do lugar. Ele também é usado como uma fonte de informações sobre o lugar e seu conteúdo, para serem apresentadas para os seres humanos, na forma de páginas web.

As entradas de diretório para as presenças web dentro de um lugar podem ser estabelecidas de duas maneiras principais. Primeiro, um serviço de descoberta de rede (Seção 16.2.1) pode ser usado para registrar automaticamente todas as presenças web implementadas pelos dispositivos dentro da sub-rede do lugar – dispositivos conectados sem fio dentro do lugar ou os servidores de infra-estrutura do lugar. Entretanto, embora os serviços de descoberta de rede sejam úteis, eles têm o problema de que nem todas as presenças web são hospedadas pelos dispositivos na sub-rede do lugar. As presenças web de entidades físicas não eletrônicas, como os seres humanos, documentos impressos e CDs de música, que se movem no lugar, ou são levadas para ele, podem estar hospedadas em qualquer lugar. Essas presenças web precisam ser registradas lá manualmente, ou por intermédio de mecanismos de percepção, em um processo chamado *registro físico* [Barton *et al.* 2002] – por exemplo, pela percepção de suas etiquetas RFID.

Um serviço chamado *gerenciador de presença na Web* [Debaty e Caswell 2001] gerencia os lugares presentes na web – por exemplo, todas as salas dentro de um prédio – e também pode gerenciar as presenças de pessoas e coisas. Os lugares são um caso particular da abstração de *contexto* do Cooltown: um conjunto de entidades relacionadas presentes na web, interligadas para propósitos como a navegação. O gerenciador de presença relaciona cada entidade presente na web com as presenças de entidades que estão em seu contexto. Por exemplo, se a entidade é uma coisa, as entidades relacionadas poderiam ser a pessoa que a transporta e o lugar onde ela está localizada. Se a entidade é uma pessoa, suas entidades relacionadas poderiam ser as coisas carregadas por ela, o lugar onde a pessoa está correntemente localizada e, possivelmente, as pessoas que a rodeiam.

16.7.2 Hyperlinks físicos

As presenças web são recursos web como qualquer outro; portanto, as páginas web podem conter *links* de texto ou imagem para presenças web como todos os outros *links*. Mas nesse modelo de *link* padrão, o usuário se depara com a presença web de uma pessoa, lugar ou coisa por intermédio de uma fonte de *informação*: a página web. O projeto Cooltown permite, adicionalmente, que os seres humanos acessem diretamente as presenças web a partir de sua fonte física: a saber, a pessoa, lugar ou coisa física específica que encontraram em suas movimentações diárias pelo mundo físico.

Um *hyperlink físico* é qualquer meio pelo qual um usuário pode recuperar o URL da presença web de uma entidade, a partir da própria entidade física ou de seus vizinhos imediatos. Consideraremos agora as maneiras de implementar os *hyperlinks* físicos. Primeiro, considere a marcação em HTML de um *link* típico em uma página web, digamos:

Tela Chop Suey de Hopper.

Isso vincula o texto "Tela Chop Suey de Hopper" de uma página web a uma página no endereço *http://cdk4.net/Hopper.HTML*, sobre o quadro Chop Suey de Edward Hopper, o qual está sendo referenciado. Agora, considere a questão de como um visitante de um museu que encontrasse o quadro poderia "clicar" na tela para obter informações sobre ela no navegador de seu telefone móvel, PDA ou outro dispositivo portátil. Isso exigira uma maneira de descobrir o URL do quadro a partir da configuração física da própria tela. Uma maneira seria escrever o URL na parede, perto do quadro, para que o usuário pudesse digitá-lo no navegador de seu dispositivo. Mas isso seria deselegante e trabalhoso.

Em vez disso, o Cooltown utiliza o fato de que os usuários têm sensores integrados em seus dispositivos, e os projetistas investigaram duas estratégias principais para descobrir os URLs das entidades por intermédio desses sensores: *percepção direta* e *percepção indireta*.

Percepção direta: neste modelo, o dispositivo do usuário percebe um URL diretamente a partir de uma etiqueta (uma etiqueta de "identificação automática") ou baliza anexada à entidade de interesse ou próxima a ela (veja a Seção 16.4.3). Uma entidade relativamente grande, como uma sala, poderia ter várias etiquetas ou balizas, localizadas em lugares bem visíveis. Uma etiqueta é um dispositivo, ou artefato passivo, que apresenta o URL quando o usuário coloca o sensor do dispositivo perto dela. Por exemplo, um telefone com câmara poderia, em princípio, realizar o reconhecimento ótico de caracteres no URL escrito em uma tabuleta ou poderia ler o URL codificado em um código de barras bidimensional. Uma baliza, por outro lado, emite regularmente o URL da entidade, normalmente por meio de raios infravermelhos (direcionais), em vez de sinais de rádio, que normalmente são onidirecionais e, assim, podem levar a uma ambigüidade quanto a qual URL pertence a qual entidade.

Em particular, o projeto Cooltown desenvolveu balizas na forma de pequenos dispositivos (a alguns centímetros um do outro) que emitem um string a cada poucos segundos por meio de raios infravermelhos, usando um protocolo sem conexão de tentativa única (Figura 16.13A). O string emitido é um documento do tipo XML, consistindo no URL da presença web da entidade e um título curto. Muitos dispositivos portáteis, como os telefones móveis e PDAs, têm tranceptores de sinal infravermelho integrados e, portanto, são capazes de receber esses strings. Quando um programa cliente recebe o string, ele pode, por exemplo, fazer o navegador do dispositivo ir diretamente para o URL recebido ou criar um *hyperlink* para o URL adquirido a partir do título recebido e adicionar esse *hyperlink* em uma lista de *hyperlinks* recebidos, na qual o usuário pode clicar quando quiser.

Figura 16.13 Capturando e imprimindo a presença web de um quadro.

Percepção indireta: a percepção indireta se dá onde o dispositivo do usuário obtém um identificador a partir de uma etiqueta ou baliza, as quais são pesquisadas para se obter um URL. O dispositivo de percepção conhece o URL de um *resolvedor* – um servidor de nomes que mantém um conjunto de vínculos de identificadores para URLs (um contexto de atribuição de nomes, na terminologia do Capítulo 9) e que retorna o URL vinculado ao identificador dado [Kindberg 2002]. De preferência, o espaço de nomes usado para identificadores de entidade deve ser suficientemente grande para permitir que cada entidade física tivesse um identificador exclusivo e, assim, eliminasse a possibilidade de ambigüidade. Entretanto, em princípio, poderiam ser usados identificadores locais, desde que fossem pesquisados apenas usando um resolvedor local – caso contrário, seria possível obter resultados espúrios, pois alguém poderia ter usado o mesmo identificador para uma entidade diferente.

Às vezes, a percepção indireta é usada porque as restrições na tecnologia da etiqueta significam que a percepção direta do URL é impossível. Por exemplo, os códigos de barras lineares não têm capacidade suficiente para armazenar um URL arbitrário; as etiquetas RFID baratas armazenam apenas um identificador binário de comprimento fixo. Em cada caso, o identificador armazenado precisa ser pesquisado para se obter o URL da presença web.

Mas também existe um motivo positivo para se usar percepção indireta: ela permite que determinada entidade física tenha um conjunto de presenças web, em vez de apenas uma. Assim como a mesma frase "Tela Chop Suey de Hopper" pode aparecer em várias páginas web diferentes, determinado quadro físico poderia levar a diferentes presenças web, de acordo com a escolha do resolvedor. Por exemplo, uma presença web do quadro poderia ser um *link* para um serviço que imprimisse uma cópia em uma impressora próxima no museu; outra presença web do mesmo quadro poderia ser uma página fornecendo informações sobre a tela, de um terceiro participante independente, sem nenhuma conexão com o museu.

A implementação da resolução segue a arquitetura da web no sentido de que cada resolvedor é um site da web independente. O software cliente é um navegador melhorado com um *plug-in* simples. Os resolvedores fornecem formulários web contendo um campo em que o cliente preenche como um efeito colateral da percepção, em vez de apresentar o campo para entrada manual do usuário. Quando o usuário, digamos, varre um código de barras, o identificador resultante é automaticamente preenchido no formulário e o cliente envia o formulário para o resolvedor. O resolvedor retorna o URL correspondente, se ele existir.

Como os próprios resolvedores são recursos web, o usuário navega neles como faria em qualquer outra página web [Kindberg 2002] e atualiza o cliente com o que o resolvedor deve usar. Em particular, o usuário pode escolher o URL de um resolvedor local, usando um *hyperlink* físico local. Por exemplo, os administradores do museu poderiam configurar balizas do Cooltown para emitir o URL do resolvedor local, para que os visitantes pudessem usar esse resolvedor para obter presenças web relevantes dos quadros dentro do museu. Igualmente, se os identificadores dos quadros fossem conhecidos e globalmente estabelecidos, então os visitantes poderiam utilizar outros resolvedores de qualquer parte da web – por exemplo, um visitante espanhol poderia utilizar o resolvedor de um site espanhol de comentários sobre arte, enquanto visitasse um museu na América do Norte.

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

17 Sistemas Multimídia Distribuídos

Os aplicativos multimídia geram e consomem fluxos contínuos de dados em tempo real. Eles são compostos por grandes quantidades de áudio, vídeo e outros elementos de dados vinculados no tempo, e o processamento e a distribuição dos elementos de dados individuais (amostras de áudio, quadros de vídeo) respeitando critérios temporais são fundamentais. Os elementos de dados distribuídos com atrasos não têm valor e normalmente são eliminados.

Uma especificação para um fluxo multimídia é expressa em termos de valores aceitáveis para a taxa com que os dados passam de uma origem para um destino (a largura de banda), para o atraso da distribuição de cada elemento (latência) e com a taxa que os elementos são perdidos ou eliminados. A latência é particularmente importante nos aplicativos interativos. Um pequeno grau de perda de dados de fluxos multimídia freqüentemente é aceitável, desde que o aplicativo possa voltar a sincronizar os elementos que vêm após aqueles que foram perdidos.

A alocação e o escalonamento dos recursos para atender as necessidades dos aplicativos multimídia, e outros, é referida como gerenciamento da qualidade do serviço. A alocação da capacidade de processamento, a largura de banda da rede e a memória (para o armazenamento em buffer dos elementos de dados distribuídos antecipadamente), tudo isso é importante. Eles são alocados em resposta às exigências da qualidade do serviço dos aplicativos. Uma requisição de qualidade do serviço bem-sucedida gera uma garantia de qualidade do serviço para o aplicativo e resulta na reserva, e no subseqüente escalonamento, dos recursos solicitados.

Este capítulo recorre substancialmente a um artigo dirigido de Ralf Herrtwich [1995] e agradecemos a ele pela permissão para usar seu material.

17.1 Introdução

Os computadores modernos podem manipular fluxos de dados contínuos (*streams*), baseados no tempo, como áudio e vídeo digital. Essa capacidade levou ao desenvolvimento de aplicações multimídia distribuídas, como as bibliotecas de vídeo interligadas em rede, a telefonia pela Internet e a videoconferência. Tais aplicações são viáveis com as redes e os sistemas de propósito gerais atuais, embora a qualidade do áudio e vídeo resultantes seja freqüentemente insatisfatória. Aplicações mais exigentes, como videoconferência de larga escala, produção de TV digital, TV interativa e sistemas de supervisão com vídeo, estão além da capacidade das tecnologias de interligação em rede e dos sistemas distribuídos atuais.

Os aplicativos multimídia exigem a distribuição de fluxos dos dados multimídia com restrições temporais para os usuários finais. Os fluxos de áudio e vídeo são gerados e consumidos em tempo real e a distribuição oportuna dos elementos individuais (amostras de áudio, quadros de vídeo) é fundamental para a integridade da aplicação. Em resumo, os sistemas multimídia são sistemas em tempo real: eles precisam executar tarefas e apresentar resultados de acordo com um escalonamento determinado externamente. O grau com que isso é conseguido pelo sistema subjacente é conhecido como *qualidade do serviço* (QoS – *Quality of Service*) usufruída por um aplicativo.

Embora os problemas do projeto de sistemas em tempo real tenham sido estudados antes do advento dos sistemas multimídia, e muitos sistemas em tempo real bem-sucedidos foram desenvolvidos (veja, por exemplo, Kopetz e Verissimo [1993]), geralmente, eles não têm sido integrados em sistemas operacionais e redes de propósito gerais. A natureza das tarefas executadas pelos sistemas em tempo real existentes, como na aviação, controle de tráfego aéreo, controle de processos de fabricação e centrais telefônicas, diferem daquelas executadas nos aplicativos multimídia. As primeiras geralmente tratam com volumes de dados relativamente pequenos, e poucas vezes têm *prazos finais rígidos*, mas o não cumprimento de um prazo pode ter conseqüências sérias ou mesmo desastrosas. Em tais casos, a solução adotada têm sido superestimar os recursos de computação e alocá-los com um escalonamento fixo que sempre garanta o atendimento desses requisitos de pior caso.

A alocação e o escalonamento planejado dos recursos para atender as necessidades dos aplicativos multimídia, e outros, é referida como *gerenciamento da qualidade do serviço*. A maioria dos sistemas operacionais e redes atuais não incluem os recursos de gerenciamento da qualidade do serviço necessários para suportar aplicativos multimídia.

As conseqüências do não cumprimento dos prazos finais em aplicativos multimídia podem ser sérias, especialmente nos ambientes comerciais, como nos serviços de vídeo sob demanda, aplicações de videoconferência comercial e medicina à distância, mas os requisitos diferem significativamente daqueles de outros aplicativos em tempo real:

- Freqüentemente, os aplicativos multimídia são altamente distribuídos e operam dentro de ambientes de computação distribuída de propósito geral. Portanto, eles competem com outros aplicativos distribuídos pela largura de banda da rede e pelos recursos de computação nas estações de trabalho dos usuários e nos servidores.
- Os requisitos de recursos dos aplicativos multimídia são dinâmicos. Uma videoconferência exigirá mais ou menos largura de banda de rede, à medida que o número de participantes aumenta ou diminui. O uso de recursos de computação na estação de trabalho de cada usuário também variará, pois, por exemplo, o número de fluxos de vídeo que precisa ser exibido varia. Os aplicativos multimídia podem envolver outras cargas variáveis ou intermitentes. Por exemplo, uma aula expositiva multimídia poderia incluir uma atividade de simulação com uso intenso do processador.
- Freqüentemente, os usuários desejam contrabalançar os custos de recursos de um aplicativo multimídia com outras atividades. Assim, talvez eles queiram reduzir suas demandas por largura de banda de vídeo em um aplicativo de videoconferência para permitir que uma conversa de voz separada prossiga, ou que o desenvolvimento de um programa, ou de uma atividade de processamento de texto, continue enquanto estão participando de uma conferência.

Os sistemas de gerenciamento de qualidade do serviço se destinam a satisfazer todas essas necessidades, administrando os recursos disponíveis dinamicamente e variando as alocações em resposta

às demandas variáveis e às prioridades do usuário. Um sistema de gerenciamento da qualidade do serviço deve supervisionar todos os recursos de computação e comunicação necessários para adquirir, processar e transmitir fluxos de dados multimídia, especialmente onde os recursos são compartilhados entre os aplicativos.

A Figura 17.1 ilustra um sistema multimídia distribuído típico, capaz de suportar uma variedade de aplicações, como videoconferência em *desktop*, o fornecimento de acesso a seqüências de vídeo armazenadas, transmissão de TV e rádio digital. Os recursos para os quais o gerenciamento da qualidade do serviço é exigido incluem largura de banda de rede, ciclos do processador e capacidade da memória. A largura de banda de disco no servidor de vídeo também pode ser incluída. Adotaremos o termo genérico *largura de banda de recurso* para nos referirmos à capacidade de qualquer recurso de hardware (rede, processador central, subsistema de disco) transmitir ou processar dados multimídia.

Em um sistema distribuído aberto, os aplicativos multimídia podem ser iniciados e usados sem organização anterior. Vários aplicativos podem coexistir na mesma rede e até na mesma estação de trabalho. Portanto, surge a necessidade de gerenciamento da qualidade do serviço, independentemente da *quantidade total* de largura de banda de recurso ou da capacidade de memória presente no sistema. O gerenciamento da qualidade do serviço é necessário para *garantir* que os aplicativos possam obter a quantidade de recursos necessária nos momentos exigidos, mesmo quando outros aplicativos estão competindo pelos recursos.

Alguns aplicativos multimídia têm sido implantados, mesmo nos atuais ambientes de computação e de redes baseados em melhor esforço e com pouco suporte à qualidade de serviço. Isso inclui:

Multimídia baseada na web: são aplicativos que fornecem o melhor acesso possível aos fluxos de dados de áudio e vídeo publicados por meio da web. Eles têm sido bem-sucedidos quando há pouca ou nenhuma necessidade de sincronização dos fluxos de dados em diferentes locais. Seu desempenho é restrito pela largura de banda limitada, pelas latências variáveis encontradas nas redes atuais, e pela incapacidade dos sistemas operacionais atuais de suportar escalonamento de recursos em tempo real. Para seqüências de áudio e vídeo de baixa qualidade, o uso extensivo de buffers no destino para atenuar as variações na largura de banda e na latência resulta na exibição contínua e suave de seqüências de vídeo, mas com um atraso da origem para o destino que pode chegar a vários segundos.

Telefone de rede e conferência por áudio: esta aplicação tem requisitos de largura de banda relativamente baixos, especialmente quando são usadas técnicas eficientes de compactação. Mas sua natureza interativa exige baixos atrasos de tempo de ida e volta (RTT - *round-trip time*) e eles nem sempre podem ser obtidos.

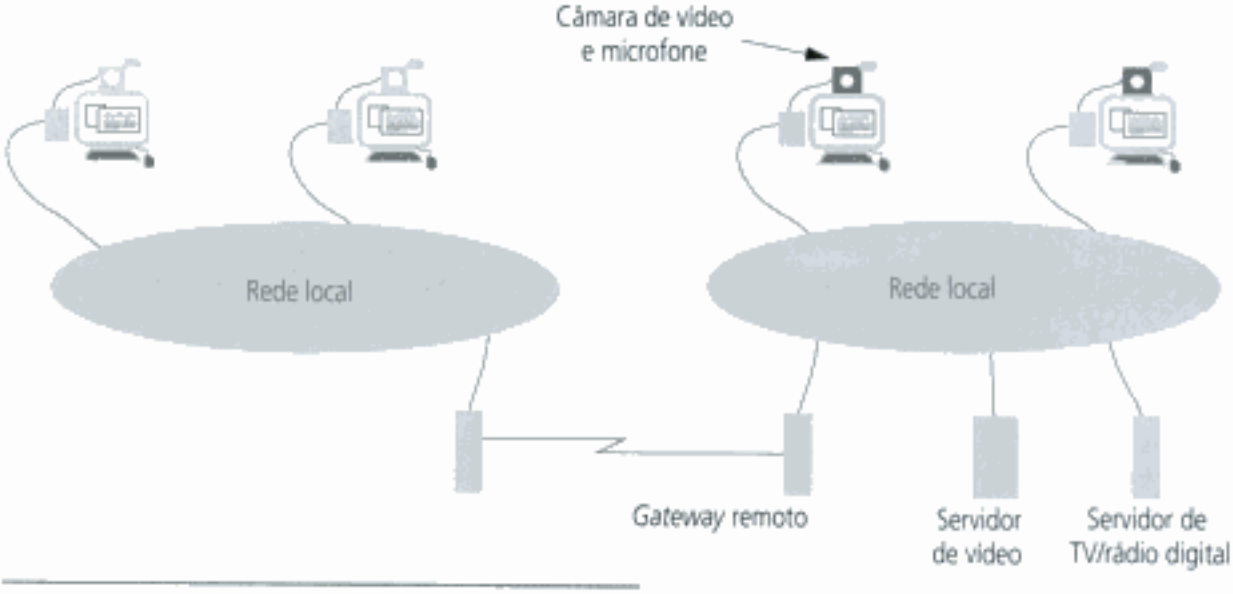


Figura 17.1 Um sistema multimídia distribuído.

Hidden page

Hidden page

Hidden page

Hidden page

métodos de melhor esforço para compartilhamento de ciclos de processador e largura de banda de rede, não podem satisfazer as necessidades das aplicações multimídia. Conforme vimos, o processamento e a transmissão de fluxos multimídia em momentos oportunos são fundamentais para eles. Uma distribuição atrasada não tem nenhum valor. Para conseguir a distribuição com restrições temporais, as aplicações precisam garantir que os recursos necessários sejam alocados e escalonados nos momentos exigidos.

O gerenciamento e a alocação de recursos para fornecer tais garantias são referidos como *gerenciamento da qualidade do serviço*. A Figura 17.4 mostra os componentes de infra-estrutura para uma aplicação simples de conferência multimídia sendo executada em dois computadores pessoais, usando compactação de dados de software e conversão de formato. As caixas brancas representam os componentes de software cujos requisitos de recursos podem afetar a qualidade do serviço da aplicação.

A figura mostra a arquitetura abstrata mais comumente usada para software multimídia, na qual *fluxos* de mídia de elementos de dados gerados continuamente (quadros de vídeo, amostras de áudio) são processados por um conjunto de processos e transferidos por meio de conexões entre eles. Os processos produzem, transformam e consomem fluxos contínuos de dados multimídia. As conexões ligam os processos em uma seqüência de uma *origem* para um *destino*, no qual os elementos de mídia são representados ou consumidos. As conexões entre os processos podem ser implementadas por conexões de rede, ou por meio de transferências na memória quando os processos residem na mesma máquina. Para os elementos de dados multimídia chegarem a seu destino a tempo, cada processo deve receber tempo de CPU, capacidade de memória e largura de banda de rede adequadas para a execução da tarefa designada, e devem ser programados para usar os recursos de forma suficientemente freqüente para enviar a tempo os elementos de dados de seu fluxo para o próximo processo.

Na Figura 17.5, especificamos os requisitos de recurso dos principais componentes de software e conexões de rede da Figura 17.4 (observe as letras correspondentes aos componentes nessas duas figuras). Claramente, os recursos exigidos só poderão ser garantidos se houver um componente de sistema responsável pela alocação e pelo escalonamento desses recursos. Vamos nos referir a esse componente como *gerenciador de qualidade do serviço*.

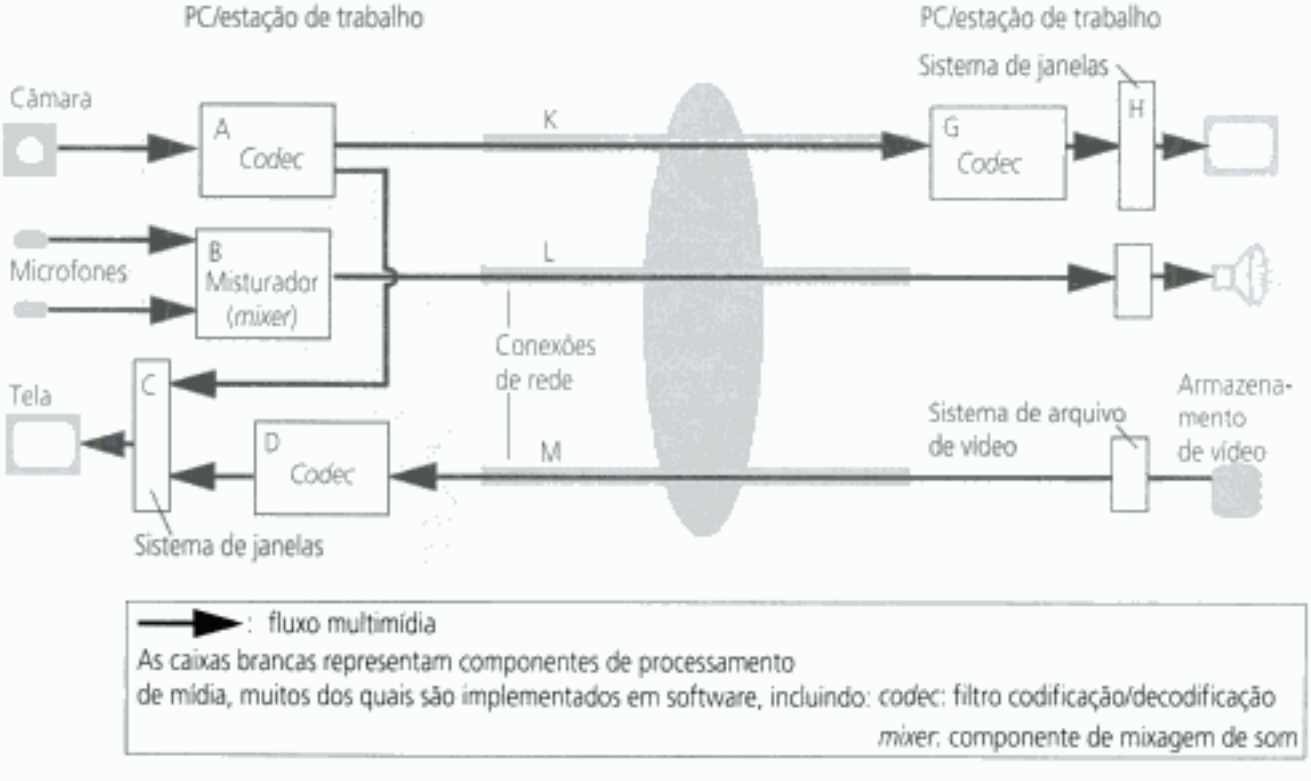


Figura 17.4 Componentes típicos de infra-estrutura para aplicações multimídia.

	Componente		*Largura de banda*	*Latência*	*Taxa de perda*	*Recursos exigidos*
	Câmara	Saída:	10 quadros/s, vídeo bruto	–	Zero	–
A	Codec	Entrada: Saída:	10 quadros/s, vídeo bruto fluxo MPEG-1	Interativa	Baixa	1 ms CPU a cada 100 ms; 1 Mbytes RAM
B	Misturador	Entrada: Saída:	2 × 44 kbps áudio 1 × 44 kbps áudio	Interativa	Muito baixa	1 ms CPU a cada 100 ms; 1 Mbytes RAM
H	Sistema de janelas	Entrada: Saída:	várias	Interativa	Baixa	5 ms CPU a cada 100 ms; 5 Mbytes RAM
K	Conexão de rede	Entrada/ Saída:	Fluxo MPEG-1, aprox. 1.5 Mbps	Interativa	Baixa	1,5 Mbps, protocolo de fluxo de baixa perda
L	Conexão de rede	Entrada/ Saída:	Áudio 44 kbps	Interativa	Muito baixa	44 kbps, protocolo de fluxo de perda muito baixa

Figura 17.5 Especificações de qualidade do serviço para componentes de aplicações multimída mostrada na Figura 17.4.

A Figura 17.6 mostra as responsabilidades do gerenciador de qualidade do serviço em forma de fluxograma. Nas duas próximas subseções, descreveremos as duas principais subtarefas do gerenciador de qualidade do serviço:

Negociação da qualidade do serviço: a aplicação indica seus requisitos de recurso para o gerenciador de qualidade do serviço. O gerenciador de qualidade do serviço avalia a possibilidade de atender os requisitos com um banco de dados dos recursos disponíveis e em relação aos comprometimentos de recurso correntes, e dá uma resposta positiva ou negativa. Se a resposta for negativa, a aplicação poderá ser reconfigurada para usar recursos reduzidos e o procedimento é repetido.

Controle de admissão: se o resultado da avaliação de recurso for positivo, os recursos solicitados serão reservados e a aplicação receberá um *contrato de recurso* indicando os recursos que foram reservados. O contrato inclui um limite de tempo. Então, a aplicação fica livre para ser executada. Se ela mudar seus requisitos de recurso, deverá notificar o gerenciador de qualidade do serviço. Se os requisitos diminuírem, os recursos liberados serão retornados ao banco de dados como recursos disponíveis. Se eles aumentarem, uma nova rodada de negociação e controle de admissão será iniciada.

No restante desta seção, descreveremos com mais detalhes as técnicas para executar essas subtarefas. É claro que, enquanto uma aplicação está em execução, há necessidade de escalonamento refinado de recursos como tempo de processador e largura de banda de rede, para garantir que os processos em tempo real recebam a tempo seus recursos alocados. As técnicas para isso serão discutidas na Seção 17.4.

17.3.1 Negociação da qualidade do serviço

Para negociar a qualidade do serviço entre uma aplicação e seu sistema subjacente, a aplicação deve especificar seus requisitos de qualidade do serviço para o gerenciador de qualidade do serviço. Isso é feito por meio da transmissão de um conjunto de parâmetros. Três parâmetros têm maior interesse quando se trata do processamento e transporte de fluxos multimídia: *largura de banda*, *latência* e *taxa de perda*.

Largura de banda: a largura de banda de um fluxo, ou componente multimídia, é a taxa na qual os dados fluem por ela.

Hidden page

Hidden page

tomar um cuidado extra para evitar o *jitter*. O *jitter* é resolvido basicamente com o uso de buffers, mas sua capacidade de remoção é limitada, pois o atraso de ponta a ponta total é restrito pela consideração mencionada anteriormente; portanto, a reprodução de seqüências de mídia também exige que os elementos da mídia cheguem antes de prazos finais fixos.

Taxa de perda: a taxa de perda é o parâmetro da qualidade do serviço mais difícil de especificar. Os valores de taxa de perda normais resultam de cálculos de probabilidade sobre o estouro de buffers e mensagens atrasadas. Esses cálculos são baseados em suposições de pior caso ou em distribuições padrão. Nenhum deles é necessariamente uma boa solução para situações práticas. Entretanto, as especificações de taxa de perda são necessárias para quantificar os parâmetros de largura de banda e latência: duas aplicações podem ter as mesmas características de largura de banda e latência; elas parecerão substancialmente diferentes quando uma aplicação perder cada quinto elemento da mídia e a outra perder apenas um em um milhão.

Assim como nas especificações de largura de banda, onde é importante não apenas o volume dos dados enviados em um intervalo de tempo, mas também sua distribuição nesse intervalo de tempo, uma especificação de taxa de perda precisa determinar o intervalo de tempo durante o qual deve esperar certa perda. Em particular, taxas de perda dadas para períodos de tempo infinitos não são úteis, pois qualquer perda em um tempo curto pode ultrapassar a taxa de longo prazo significativamente.

Moldagem de tráfego ◊ Moldagem do tráfego é o termo usado para descrever o uso de buffers de saída para suavizar o fluxo de elementos de dados. O parâmetro da largura de banda de um fluxo multimídia normalmente fornece uma aproximação ideal do padrão de tráfego real que ocorrerá quando o fluxo for transmitido. Quanto mais próximo for o padrão de tráfego real da sua descrição, melhor o sistema poderá manipular o tráfego; em particular, quando ele utilizar métodos de escalonamento projetados para pedidos periódicos.

O modelo LBAP de variações de largura de banda exige regular a taxa de rajada dos fluxos multimídia. Todo fluxo pode ser regulado pela inserção de um buffer na origem e pela definição de um método com que os elementos de dados sejam consumidos do buffer. Uma boa ilustração desse método é a imagem de um balde furado (Figura 17.7(a)): o balde pode ser preenchido arbitrariamente com água, até ficar cheio e; por meio de um furo embaixo do balde, a água fluirá continuamente. O algoritmo do balde furado garante que um fluxo nunca será maior do que uma taxa R. O tamanho do buffer B define a rajada máxima que um fluxo pode estar sujeito sem perder elementos. B também limita o tempo durante o qual um elemento permanecerá no balde.

O algoritmo do balde furado elimina completamente as rajadas. Tal eliminação nem sempre é necessária, já que a largura de banda é limitada em qualquer intervalo de tempo. O algoritmo do balde com marcas (*tokens*) consegue isso, permitindo que rajadas maiores ocorram, quando um fluxo estiver ocioso por algum tempo (Figura 17.7(b)). Trata-se de uma variação do algoritmo do balde furado, na qual são gerados *tokens* para enviar dados a uma velocidade fixa R. Eles são coletados em um balde de tamanho B. Dados de tamanho S só podem ser enviados se pelo menos S *tokens* estiverem no balde.

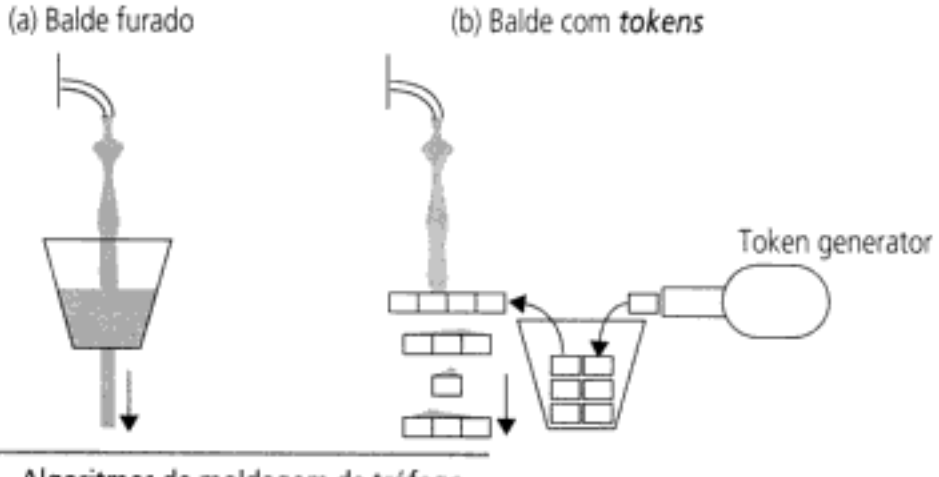


Figura 17.7 Algoritmos de moldagem de tráfego.

Hidden page

de alta fidelidade, vídeo ao vivo e reprodução, etc. Os requisitos de todas as classes devem ser conhecidos implicitamente por todos os componentes do sistema; o sistema pode até ser configurado para certa mistura de tráfego.

Procedimentos de negociação ◊ Para aplicações multimídia distribuídas, os componentes de um fluxo provavelmente estarão localizados em vários nós. Haverá um gerenciador de qualidade de serviço em cada nó. Uma estratégia simples de negociação de qualidade de serviço é seguir o fluxo de dados ao longo de cada fluxo, da origem até o destino. Um componente de origem inicia a negociação enviando uma especificação de fluxo para seu gerenciador de qualidade do serviço local. O gerenciador pode verificar em seu banco de dados de recursos disponíveis, se a qualidade de serviço solicitada pode ser fornecida. Se outros sistemas estiverem envolvidos na aplicação, a especificação de fluxo será encaminhada para o próximo nó onde os recursos são exigidos. A especificação de fluxo percorre todos os nós, até que o destino final seja encontrado. Então, a informação sobre se a qualidade de serviço desejada pode ser fornecida pelo sistema é passada de volta para a origem. Essa estratégia simples de negociação é satisfatória para muitos propósitos, mas ela não considera as possibilidades de conflito entre negociações de qualidade de serviço concorrentes partindo de nós diferentes. Um procedimento de negociação de qualidade de serviço transacional distribuído exigiria uma solução completa para esse problema.

Raramente as aplicações têm requisitos de qualidade do serviço fixos. Em vez de retornar um valor booleano sobre se certa qualidade de serviço pode ser fornecida ou não, é mais apropriado o sistema determinar qual tipo de qualidade de serviço pode fornecer e deixar a aplicação decidir se ela é aceitável. Para evitar qualidade do serviço super-otimizada, ou para cancelar a negociação quando se tornar claro que a qualidade desejada não pode ser obtida, é comum especificar um valor desejado e o pior valor para cada parâmetro de qualidade do serviço. Uma aplicação pode especificar que deseja uma largura de banda de 1,5 Mbps, mas que também poderia manipular 1 Mbps, ou que o atraso deve ser de 200 ms, mas que 300 ms seria o pior caso ainda aceitável. Como apenas um parâmetro pode ser otimizado por vez, sistemas como o HeiRAT [Vogt *et al.* 1993] esperam que o usuário defina valores de apenas dois parâmetros e deixam que o sistema otimize o terceiro.

Se um fluxo tem vários destinos, o caminho de negociação bifurca de acordo com o fluxo de dados. Como uma ampliação simples do esquema anterior, nós intermediários podem agregar mensagens de retorno de qualidade do serviço dos destinos, para produzir valores de pior caso para os parâmetros da qualidade de serviço. Então, a largura de banda disponível se torna a menor largura de banda disponível de todos os destinos, o atraso se torna o mais longo de todos os destinos e a taxa de perda se torna a maior de todos os destinos. Esse é o procedimento praticado pelos protocolos de negociação iniciada pelo remetente, como SRP, ST-II e RCAP [Banerjea e Mah 1991].

Nas situações com destinos heterogêneos, normalmente é inadequado atribuir uma qualidade de serviço de pior caso comum para todos os destinos. Em vez disso, cada destino deve receber a melhor qualidade de serviço possível. Isso exige um processo de negociação iniciada pelo receptor, em vez de orientada pelo remetente. O RSVP [Zhang *et al.* 1993] é um protocolo de negociação de qualidade de serviço alternativo no qual os destinos se conectam a fluxos. As origens comunicam a existência de fluxos e suas características inerentes para todos os destinos. Então, os destinos podem se conectar no nó mais próximo através do qual o fluxo passa e extrair dados de lá. Para que obtenham dados com a qualidade de serviço apropriada, talvez eles precisem usar técnicas como a filtragem (discutida na Seção 17.5).

17.3.2 Controle de admissão

O controle de admissão regula o acesso aos recursos para evitar sobrecarga de recurso e para proteger os recursos de pedidos que eles não possam atender. Isso envolve rejeitar pedidos de serviço, caso os requisitos de recurso de um novo fluxo multimídia violem as garantias de qualidade do serviço existentes.

Um esquema de controle de admissão é baseado em algum conhecimento da capacidade global do sistema e da carga gerada por cada aplicação. A especificação do requisito de largura de banda de uma aplicação pode refletir a quantidade máxima de largura de banda que a aplicação exigirá, a

Hidden page

17.4 Gerenciamento de recursos

Para fornecer certo nível de qualidade de serviço para uma aplicação, um sistema não apenas precisa ter recursos suficientes (desempenho), como também precisa tornar esses recursos disponíveis para a aplicação, quando eles forem necessários (escalonamento).

17.4.1 Escalonamento de recursos

Os processos precisam ter recursos atribuídos de acordo com suas prioridades. Um escalonador de recursos determina a prioridade dos processos com base em certos critérios. Os escalonadores tradicionalmente encontrados em sistemas operacionais freqüentemente baseiam suas decisões de alocar a CPU a processos em função do tempo de resposta e na imparcialidade: as tarefas com uso intenso de E/S recebem alta prioridade para garantir resposta rápida aos pedidos de usuário, as tarefas ligadas à CPU recebem prioridades mais baixas e, de modo geral, os processos na mesma classe são tratados igualmente.

Os dois critérios permanecem válidos para sistemas multimídia, mas a existência de prazos finais para a distribuição de elementos de dados multimídia individuais muda a natureza do problema do escalonamento. Algoritmos de escalonamento em tempo real podem ser aplicados a esse problema, conforme discutido a seguir. Como os sistemas multimídia têm de manipular mídia discreta e contínua, torna-se um desafio fornecer um serviço suficientemente ágil para tratar fluxos dependentes do tempo, sem causar inanição de acesso à mídia discreta e de outras aplicações interativas.

Métodos de escalonamento precisam ser aplicados (e coordenados) em todos os recursos que afetam o desempenho de uma aplicação multimídia. Em um cenário típico, um fluxo multimídia seria recuperado do disco e depois enviado, por meio de uma rede, para uma estação de destino, onde seria sincronizado com um fluxo proveniente de outra origem e, finalmente, exibido. Os recursos exigidos nesse exemplo incluem o disco, a rede e as CPUs, assim como memória e largura de banda, em todos os sistemas envolvidos.

Escalonamento imparcial (*fairness*) ◊ Se vários fluxos concorrem pelo mesmo recurso, torna-se necessário ser imparcial e impedir que fluxos mal comportados ocupem largura de banda em demasia. Uma estratégia simples para garantir a imparcialidade é aplicar escalonamento em rodízio em todos os fluxos da mesma classe. Embora em Nagle [1987] tal método tenha sido introduzido pacote por pacote, em Demers *et al.* [1989] ele é usado bit por bit, o que proporciona uma maior imparcialidade com relação aos tamanhos de pacote variados e aos tempos de chegada de pacote. Esses métodos são conhecidos como enfileiramento imparcial (*fair queuing*).

Na prática, os pacotes não podem ser enviados bit a bit, mas dada uma taxa de quadros é possível calcular, para cada pacote, quando ele deve ser enviado completamente. Se as transmissões de pacote forem ordenadas com base nesse cálculo, se obterá quase o mesmo comportamento do rodízio de bit a bit real, exceto que, quando um pacote grande for enviado, ele poderá bloquear a transmissão de um pacote menor, o qual teria sido preferido no esquema bit a bit. Entretanto, nenhum pacote é atrasado por mais do que o tempo de transmissão de pacote máximo.

Todos os esquemas de rodízio básicos atribuem a mesma largura de banda para cada fluxo. Para levar em conta a largura de banda individual dos fluxos, o esquema bit a bit pode ser ampliado de modo que, para certos fluxos, um número maior de bits possa ser transmitido por ciclo. Esse método é chamado de enfileiramento imparcial ponderado (*weighted fair queuing*).

Escalonamento em tempo real ◊ Vários algoritmos de escalonamento em tempo real foram desenvolvidos para atender as necessidades de escalonamento da CPU de aplicações como o controle de processo industrial. Supondo que os recursos de CPU não tenham sido alocados em demasia (o que é a tarefa do gerenciador de qualidade de serviço), esses algoritmos atribuem repartições de tempo de CPU para um conjunto de processos de uma maneira que garanta a execução de suas tarefas a tempo.

Os métodos tradicionais de escalonamento em tempo real se adaptam muito bem ao modelo de fluxos multimídia contínuos regulares. O escalonamento EDF (*Earliest-Deadline-First*) quase tem se tornado um sinônimo para esses métodos. Um escalonador EDF utiliza um prazo final, que é associa-

Hidden page

pendentes da mídia, embora a estratégia de mudança de escala global seja a mesma: fazer uma nova amostragem de determinado sinal. Para informações de áudio, essa nova amostragem pode ser obtida reduzindo-se a taxa de amostragem de áudio. Ela também pode ser obtida eliminando-se um canal em uma transmissão em estéreo. Conforme esse exemplo mostra, diferentes métodos de mudança de escala podem trabalhar com diferentes granularidades.

Para vídeo, os seguintes métodos de mudança de escala são apropriados:

Mudança de escala temporal: reduz a resolução do fluxo de vídeo no domínio do tempo, diminuindo o número de quadros de vídeo transmitidos dentro de um intervalo. A mudança de escala temporal é mais conveniente para fluxos de vídeo em que os quadros individuais são autocontidos e podem ser acessados independentemente. As técnicas de compactação delta são mais difíceis de manipular, pois nem todos os quadros podem ser eliminados facilmente. Portanto, a mudança de escala temporal é mais conveniente para Motion JPEG do que para fluxos MPEG.

Mudança de escala espacial: reduz o número de *pixels* de cada imagem em um fluxo de vídeo. Para a mudança de escala espacial, a organização hierárquica é ideal, pois o vídeo compactado está disponível imediatamente, em várias resoluções. Portanto, o vídeo pode ser transferido pela rede usando diferentes resoluções, sem gravação de cada imagem antes de finalmente transmitila. JPEG e MPEG-2 suportam diferentes resoluções espaciais de imagens e são muito convenientes para esse tipo de mudança de escala.

Mudança de escala de freqüência: modifica o algoritmo de compactação aplicado a uma imagem. Isso resulta em alguma perda de qualidade, mas em um cenário típico, a compactação pode ser aumentada significativamente, antes que uma redução da qualidade da imagem se torne visível.

Mudança de escala de amplitude: reduz as intensidades das cores de cada *pixel* da imagem. Este método de mudança de escala é usado nas codificações H.261 para chegar a uma vazão (*throughput*) constante quando o conteúdo da imagem varia.

Mudança de escala do espaço de cores: reduz o número de entradas no espaço de cores. Uma maneira de perceber a mudança de escala do espaço de cores é trocar de apresentação em cores para escala de tons.

Se necessário, podem ser usadas combinações desses métodos de mudança de escala.

Um sistema para realizar mudança de escala consiste em um processo monitor no destino e um processo de mudança de escala na origem. O monitor controla os tempos de chegada de mensagens em um fluxo. Mensagens atrasadas são uma indicação de um gargalo no sistema. Então, o monitor envia uma mensagem de *diminuir escala* para a origem e esta reduz a largura de banda do fluxo. Após algum período de tempo, a origem aumenta a escala do fluxo novamente. Se o gargalo ainda existir, o monitor detectará novamente um atraso e reduzirá a escala do fluxo [Delgrossi *et al.* 1993]. Um problema para o sistema de mudança de escala é evitar operações de *aumento de escala* desnecessárias e evitar a oscilação do sistema.

17.5.2 Filtragem

Como a mudança de escala modifica um fluxo na origem, ela nem sempre é conveniente para aplicações que envolvam vários receptores: quando ocorre um gargalo na rota para um destino, esse destino envia uma mensagem de *diminuir escala* para a origem e todos os destinos recebem a qualidade degradada, embora alguns não tivessem problemas para manipular o fluxo original.

A filtragem é um método que fornece a melhor qualidade de serviço possível para cada destino, aplicando mudança de escala em cada nó relevante no caminho da origem para o destino (Figura 17.9). O RSVP (*Resource reservation protocol*) [Zhang *et al.* 1993] é um exemplo de protocolo de negociação de qualidade de serviço que suporta filtragem. A filtragem exige que um fluxo seja dividido em um conjunto de subfluxos hierárquicos, cada um adicionando um nível mais alto de qualidade. A capacidade dos nós em um caminho determina o número de subfluxos recebidos por um destino. Todos os outros subfluxos são filtrados o mais próximo da origem possível (talvez até na origem), para evitar a transferência de dados que posteriormente serão jogados fora. Um subfluxo não é filtrado em um nó intermediário, caso exista um caminho em algum lugar mais adiante que possa transportar o subfluxo inteiro.

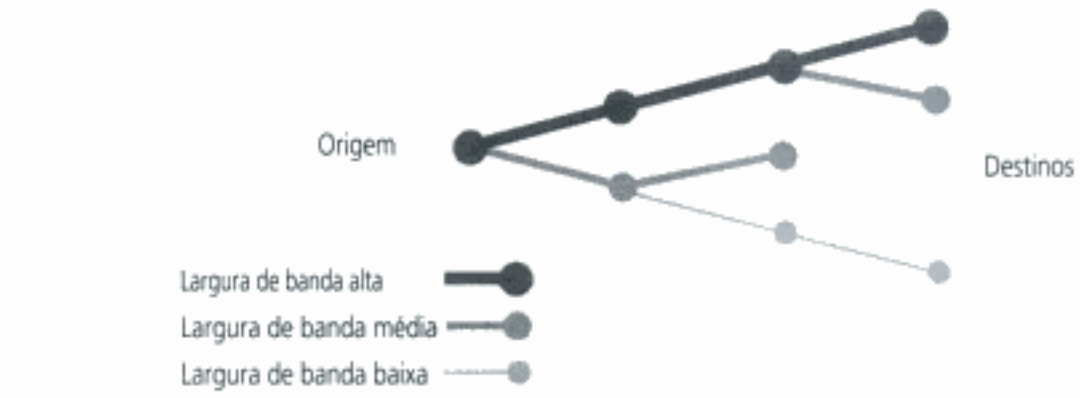


Figura 17.9 **Filtragem.**

17.6 Estudo de caso: o servidor de arquivos de vídeo Tiger

Um sistema de armazenamento de vídeos que fornece múltiplos fluxos de vídeo em tempo real simultaneamente é considerado um importante componente de sistema para suportar aplicações multimídia orientadas para o consumidor. Vários protótipos de sistemas desse tipo foram desenvolvidos e alguns até se transformaram em produtos (consulte [Cheng 1998]). Um dos mais avançados é o servidor de arquivos de vídeo Tiger, desenvolvido no Microsoft Research Labs [Bolosky *et al.* 1996].

Objetivos de projeto ◊ Os principais objetivos de projeto do sistema foram os seguintes:

Vídeo sob demanda para um grande número de usuários: a aplicação típica é um serviço que fornece filmes para clientes pagantes. Os filmes são selecionados em uma grande biblioteca de filmes digitais armazenados. Os clientes devem receber os primeiros quadros de seus filmes escolhidos dentro de poucos segundos após emitirem um pedido e devem ser capazes de executar operações de pausa, retrocesso e avanço rápido à vontade. Embora a biblioteca de filmes disponíveis seja grande, poucos filmes podem ser muito populares e eles serão o motivo de vários pedidos não sincronizados, resultando em várias reproduções concorrentes, porém deslocadas no tempo.

Qualidade do serviço: os fluxos de vídeo devem ser fornecidos a uma velocidade constante, com uma flutuação (*jitter*) máxima determinada pela quantidade (presumida como sendo pequena) de uso de buffers disponíveis nos clientes e com uma taxa de perda muito baixa.

Sistema com mudança de escala e distribuído: o objetivo era projetar um sistema com uma arquitetura extensível (pela adição de computadores) para suportar até 10.000 clientes simultaneamente.

Hardware de baixo custo: o sistema era para ser construído usando hardware de baixo custo (PCs comerciais com unidades de disco padrão).

Tolerante a falhas: o sistema deveria continuar a funcionar sem degradação notável após a falha de qualquer computador servidor ou unidade de disco.

Tomados em conjunto, esses requisitos exigem uma estratégia radical para o armazenamento e a recuperação de dados de vídeo e um algoritmo de escalonamento eficiente, que harmonize a carga de trabalho entre um grande número de servidores semelhantes. A principal tarefa é a transferência de fluxos de dados de vídeo de largura de banda alta do armazenamento em disco para uma rede e é essa carga que precisa ser compartilhada entre os servidores.

Arquitetura ◊ A arquitetura de hardware do Tiger aparece na Figura 17.10. Todos os componentes são produtos de prateleira. Os computadores *filhotes* (*cub*) mostrados na figura são PCs idênticos, com o mesmo número de unidades de disco rígido padrão (normalmente, entre 2 e 4) ligadas em cada um. Eles também estão equipados com placas de rede Ethernet e ATM. O *controlador* é outro PC. Ele não está envolvido na manipulação de dados multimídia e é responsável apenas por manipular os pedidos de clientes e pelo gerenciamento das programações de execução dos *cubs*.

Hidden page

- posição do visualizador no arquivo (o próximo bloco a ser distribuído no fluxo);
- pelo número de seqüência de exibição do visualizador (a partir do qual pode ser calculado um tempo de distribuição para o próximo bloco);
- por algumas informações de contabilidade.

O escalonamento está ilustrado na Figura 17.11. O *tempo de reprodução do bloco T* é o tempo que será exigido para um visualizador exibir um bloco no computador cliente; normalmente, cerca de 1 segundo, e é suposto ser o mesmo para todos os filmes armazenados. Portanto, o Tiger deve manter um intervalo de tempo *T* entre os tempos de distribuição dos blocos em cada fluxo, com uma pequena flutuação (*jitter*) permitida, que é determinada pelo uso de buffers disponíveis nos computadores clientes.

Cada *cub* mantém um ponteiro para o escalonamento de cada disco que controla. Durante cada tempo de reprodução de bloco, ele deve processar todas as repartições com números de bloco que caiam nos discos que controla, e tempos de distribuição que caiam dentro do tempo de reprodução de bloco corrente. O *cub* percorre o escalonamento em repartições de processamento em tempo real, como segue:

1. Lê o próximo bloco no armazenamento em buffer no *cub*.
2. Empacota o bloco e o envia para o controlador de rede ATM do *cub*, com o endereço do computador cliente.
3. Atualiza o estado do visualizador no escalonamento para mostrar o novo próximo bloco e exibir o número de seqüência, e passa a repartição atualizada para o próximo *cub*.

Supõe-se que essas ações ocupam um tempo máximo *t*, que é conhecido como tempo de serviço do bloco. Conforme pode ser visto na Figura 17.11, *t* é substancialmente menor do que o tempo de reprodução do bloco. O valor de *t* é determinado pela largura de banda de disco ou pela largura de banda de rede, qual for menor. (Os recursos de processamento em um *cub* são adequados para executar o trabalho programado de todos os discos anexos.) Quando um *cub* tiver concluído as tarefas programadas para o tempo de reprodução corrente do bloco, ele estará disponível para tarefas não programadas, até o início do próximo tempo de reprodução. Na prática, os discos não fornecem blocos com um atraso fixo e, para acomodar sua distribuição desigual, a leitura do disco é iniciada pelo menos um tempo de serviço do bloco antes que o bloco seja necessário para empacotamento e distribuição.

Um disco pode fazer o trabalho de atender *T/t* fluxos e os valores de *T* e *t* normalmente resultam em um valor > 4 para essa relação. Isso e o número de discos no sistema inteiro determinam o número de visualizadores que um sistema Tiger pode atender. Por exemplo, um sistema Tiger com cinco *cubs* e três discos ligados a cada um pode distribuir aproximadamente 70 fluxos de vídeo simultaneamente.

Tolerância a falhas ◊ Devido à segmentação de todos os arquivos de filme entre todos os discos em um sistema Tiger, a falha de qualquer componente (uma unidade de disco ou um *cub*) resultaria em uma interrupção do serviço para todos os clientes. O projeto do Tiger resolve isso por meio da recuperação de dados das cópias secundárias espelhadas, quando um bloco primário está indisponível devido à falha de um *cub* ou de uma unidade de disco. Lembre-se de que os blocos secundários são menores

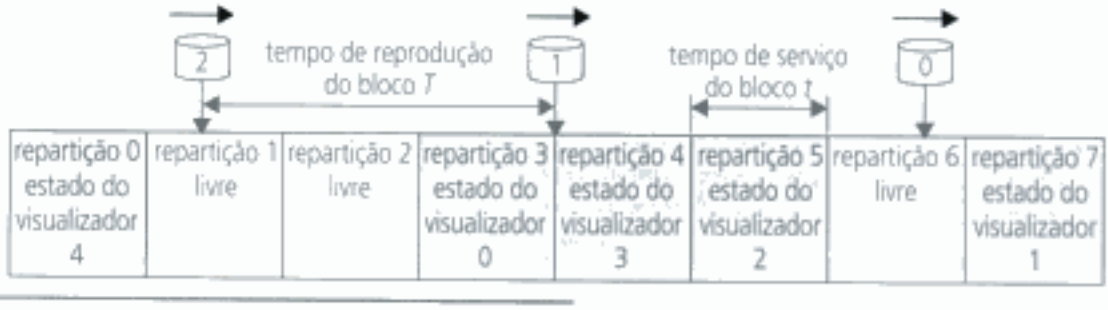


Figura 17.11 Escalonamento Tiger.

Hidden page

perda de menos de um bloco em 180.000, quando todos os *cubs* estavam funcionando. Com um *cub* defeituoso, menos de 1 em 40.000 blocos eram perdidos. Esses resultados são impressionantes e parecem corroborar a afirmação de que um sistema Tiger poderia ser configurado com até 1.000 *cubs*, servindo até 30.000–40.000 visualizadores simultâneos.

17.7 Resumo

As aplicações multimídia exigem novos mecanismos de sistema para permitir que eles manipulem grandes volumes de dados dependentes do tempo. Os mais importantes desses mecanismos se preocupam com o gerenciamento da qualidade do serviço. Eles devem alocar largura de banda, e outros recursos, de uma maneira que garanta que os requisitos de recursos da aplicação sejam atendidos e devem programar o uso dos recursos para que os prazos finais muito refinados das aplicações multimídia sejam atendidos.

O gerenciamento de qualidade de serviço trata dos pedidos de qualidade de serviço das aplicações, especificando a largura de banda, a latência e taxas de perda aceitáveis para os fluxos multimídia, e realiza o controle de admissão, determinando se recursos não reservados suficientes estão disponíveis para atender cada novo pedido e negociar com a aplicação, se necessário. Uma vez que um pedido de qualidade de serviço seja aceito, os recursos são reservados e uma garantia é emitida para a aplicação.

A capacidade do processador e a largura de banda de rede alocada para uma aplicação devem, então, ser programadas para atender suas necessidades. Um algoritmo de escalonamento de processador em tempo real, como o *earliest-deadline-first* (EDF) ou o *rate-monotonic* (RM), é exigido para garantir que cada elemento do fluxo seja processado a tempo.

Moldagem de tráfego é o nome dado aos algoritmos que colocam dados no buffer em tempo real para suavizar as irregularidades da temporização que surgem inevitavelmente. Os fluxos podem ser adaptados para utilizar menos recursos, reduzindo a largura de banda da origem (mudança de escala) ou em pontos ao longo do caminho (filtragem).

O servidor de arquivo de vídeo Tiger é um exemplo excelente de sistema com capacidade de mudança de escala que fornece distribuição de fluxo em uma escala potencialmente muito grande, com fortes garantias de qualidade do serviço. Seu escalonamento de recurso é altamente especializado e ele oferece um exemplo excelente da estratégia de projeto alterada que freqüentemente é exigida para tais sistemas.

Exercícios

17.1 Descreva em linhas gerais um sistema para suportar um recurso de ensaio musical distribuído. Sugira os requisitos de qualidade de serviço convenientes e uma configuração de hardware e software que possa ser usada. *páginas 622–623, 627–628*

17.2 Atualmente, a Internet não oferece nenhuma facilidade de reserva de recurso ou de gerenciamento de qualidade de serviço. Como as aplicações de fluxo de áudio e vídeo baseados na Internet existentes obtêm qualidade aceitável? Quais limitações as soluções que eles adotam são impostas as aplicações multimídia? *páginas 622–623, 631–632, 636–637*

17.3 Explique as diferenças entre as três formas de sincronização (estado distribuído síncrono, sincronização de mídia e sincronização externa) que podem ser exigidas nas aplicações multimídia distribuídas. Sugira mecanismos por meio dos quais cada uma delas poderia ser obtida, por exemplo, em uma aplicação de videoconferência. *página 623–624*

17.4 Descreva em linhas gerais o projeto de um gerenciador da qualidade do serviço para permitir que computadores *desktop* conectados por uma rede ATM suportem várias aplicações multimídia concorrentes. Defina uma API para seu gerenciador de qualidade de serviço, fornecendo as principais operações, com seus parâmetros e resultados. *páginas 627–630*

17.5 Para especificar os componentes de software dos requisitos de recurso que processam dados multimídia, precisamos de estimativas de suas cargas de processamento. Como essa informação pode ser obtida sem trabalho desnecessário? *páginas 627–630*

17.6 Como o sistema Tiger suporta um grande número de clientes, todos solicitando o mesmo filme em momentos aleatórios? *páginas 637–642*

17.7 A escalonamento do Tiger é potencialmente uma grande estrutura de dados que muda freqüentemente, mas cada *cub* precisa de uma representação atualizada das partes que está manipulando no momento. Sugira um mecanismo para a distribuição do escalonamento dos *cubs*. *páginas 637–642*

17.8 Quando o Tiger está operando com um disco, ou com um *cub* defeituoso, blocos de dados secundários são usados no lugar dos primários ausentes. Os blocos secundários são n vezes menores do que os primários (onde n é o fator de desagrupamento). Como o sistema acomoda essa variabilidade no tamanho do bloco? *página 640–641*

18 Memória Compartilhada Distribuída

Este capítulo descreve a memória compartilhada distribuída (DSM – *Distributed Shared Memory*), uma abstração usada para compartilhar dados entre processos em computadores que não compartilham memória física. A motivação para a DSM é que ela permite o emprego de um modelo de programação de memória compartilhada, o qual tem algumas vantagens em relação aos modelos baseados em mensagem. Por exemplo, os programadores não precisam empacotar dados na DSM.

Um problema fundamental na implementação da DSM é como obter e manter bom desempenho quando os sistemas passam para grandes números de computadores. Os acessos à DSM envolvem comunicação na rede subjacente. Os processos que concorrem pelos mesmos itens de dados, ou por itens vizinhos, podem ocasionar grandes volumes de comunicação. O volume de comunicação está fortemente relacionado ao modelo de consistência de uma DSM – o modelo que determina qual dos muitos valores possíveis escritos em memória será retornado quando um processo ler dados de um local da DSM.

Este capítulo discute os problemas de projeto da DSM, como o modelo de consistência e os problemas de implementação, tais como invalidar ou atualizar cópias de um item de dado quando ele for escrito na memória compartilhada. Finalmente, o capítulo descreve a consistência relaxada – um modelo de consistência relativamente fraco, que é adequado para muitos propósitos e relativamente barato (computacionalmente) de implementar.

18.1 Introdução

A memória compartilhada distribuída (DSM – *Distributed Shared Memory*) é uma abstração usada para o compartilhamento de dados entre computadores que não compartilham memória física. Os processos acessam a DSM por meio de leituras e atualizações no que parece ser uma memória normal, dentro de seu espaço de endereçamento. Entretanto, um *runtime* subjacente garante, de forma transparente, que os processos que estão em execução em diferentes computadores observem as atualizações feitas pelos outros. É como se os processos acessassem uma memória compartilhada única, mas na verdade a memória física é distribuída (veja a Figura 18.1).

A principal característica da DSM é que ela poupa o programador das preocupações com a passagem de mensagens ao escrever aplicações que, de outra forma, talvez tivessem de usá-la. A DSM é principalmente uma ferramenta para aplicações paralelas, para quaisquer aplicações distribuídas ou grupo de aplicações nas quais itens de dados compartilhados individuais podem ser acessados diretamente. Em geral, a DSM é menos apropriada nos sistemas cliente-servidor, onde os clientes normalmente vêem os recursos mantidos no servidor como dados abstratos e os acessam por meio de requisições (por razões de modularidade e proteção). Entretanto, os servidores podem fornecer uma DSM que seja compartilhada entre os clientes. Por exemplo, os arquivos mapeados na memória que são compartilhados, e para os quais é mantido certo grau de consistência, são uma forma de DSM. (Os arquivos mapeados foram introduzidos com o sistema operacional MULTICS [Organick 1972].)

A passagem de mensagens não pode ser totalmente evitada em um sistema distribuído: na ausência de memória fisicamente compartilhada, o suporte de *runtime* da DSM precisa enviar atualizações em mensagens entre os computadores. Os sistemas de DSM gerenciam dados replicados: cada computador tem uma cópia local dos itens de dados armazenados na DSM e acessados recentemente, para obter velocidade de acesso. Os problemas de implementação da DSM são relacionados àqueles discutidos no Capítulo 15, assim como àqueles do armazenamento de arquivos compartilhados em cache, discutidos no Capítulo 8.

Um dos primeiros exemplos notáveis de implementação de uma DSM foi o sistema de arquivos Apollo Domain [Leach *et al.* 1983], no qual os processos contidos em diferentes estações de trabalho compartilhavam arquivos mapeando-os simultaneamente em seus espaços de endereçamento. Esse exemplo mostra que a memória compartilhada distribuída pode ser persistente. Isto é, ela pode durar mais do que a execução de qualquer processo, ou grupo de processos, que a acesse e, com o passar do tempo, ser compartilhada por diferentes processos.

A importância da DSM aumentou primeiramente com o desenvolvimento de multiprocessadores de memória compartilhada (veja a Seção 6.3). Houve muitas pesquisas investigando algoritmos convenientes para computação paralela nesses multiprocessadores. Em relação à arquitetura de hardware, os desenvolvimentos incluem estratégias de uso da cache e interconexões rápidas entre memória e processador para maximizar o número de processadores que podem ser atendidos, enquanto se obtém

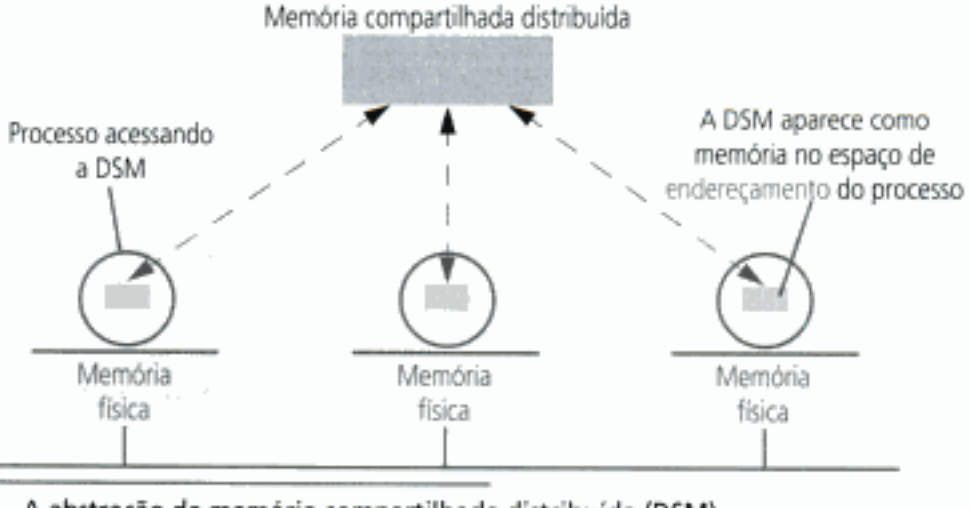


Figura 18.1 A abstração da memória compartilhada distribuída (DSM).

Hidden page

Eficiência: as experiências mostram que certos programas paralelos desenvolvidos para DSM podem ser executados quase tão bem quanto programas funcionalmente equivalentes escritos para plataformas de passagem de mensagens, no mesmo hardware [Carter *et al.* 1991] – pelo menos no caso de um número relativamente pequeno de computadores (10, mais ou menos). Entretanto, esse resultado não pode ser generalizado. O desempenho de um programa baseado em DSM depende de muitos fatores, conforme discutiremos a seguir – particularmente do padrão de compartilhamento de dados (como o fato de um item ser atualizado por vários processos).

Existe uma diferença na visibilidade dos custos associados aos dois tipos de programação. Na passagem de mensagens, todos os acessos a dados remotos são explícitos e, portanto, o programador sempre sabe se uma operação em particular se dá no processo ou envolve um custo de comunicação. Usando DSM, entretanto, qualquer leitura, ou atualização, em particular, pode ou não envolver comunicação por parte do suporte de *runtime* subjacente. O envolvimento ou não depende de fatores como o fato dos dados terem sido acessados antes e do padrão de compartilhamento entre processos nos diferentes computadores.

Não existe uma resposta definitiva para o fato de a DSM ser preferível à passagem de mensagens para qualquer aplicação em particular. A DSM é uma ferramenta promissora, cujo status final depende da eficiência com que pode ser implementada.

18.1.2 Estratégias de implementação de DSM

A memória compartilhada distribuída é implementada usando-se hardware especializado, ou uma combinação de hardware especializado, memória virtual paginada convencional ou *middleware*:

Hardware: as arquiteturas de multiprocessadores com memória compartilhada baseadas em uma arquitetura NUMA (por exemplo, Dash [Lenoski *et al.* 1992] e PLUS [Bisiani e Ravishankar 1990]) contam com hardware especializado para fornecer aos processadores uma visão consistente da memória compartilhada. Elas tratam das instruções de memória LOAD e STORE comunicando-se com a memória remota e com módulos de cache, conforme for necessário, para armazenar e recuperar dados. Essa comunicação se dá por meio de uma interconexão de alta velocidade, que é análoga a uma rede. O protótipo do multiprocessador Dash tem 64 nós conectados em uma arquitetura NUMA.

Memória virtual paginada: muitos sistemas, incluindo Ivy [Li e Hudak 1989], Munin [Carter *et al.* 1991], Mirage [Fleisch e Popek 1989], Clouds [Dasgupta *et al.* 1991] (veja www.cdk4.net/oss), Choices [Sane *et al.* 1990], COOL [Lea *et al.* 1993] e Mether [Minnich e Farber 1989], implementam a DSM como uma região de memória virtual ocupando o mesmo intervalo de endereços no espaço de endereçamento de cada processo participante. Esse tipo de implementação normalmente é conveniente apenas para um conjunto de computadores homogêneos, com dados e formatos de páginas comuns.

Middleware: algumas linguagens, como Orca [Bal *et al.* 1990], e *middleware*, como Linda [Carriero e Gelernter 1989] e seus derivados JavaSpaces [Bishop e Warren 2003] e TSpaces [Wyckoff *et al.* 1998], suportam formas de DSM sem nenhum suporte para hardware ou paginação, de uma maneira neutra quanto a plataforma. Nesse tipo de implementação, o compartilhamento é implementado por meio da comunicação entre instâncias da camada de suporte em nível de usuário nos clientes e servidores. Os processos fazem chamadas a essa camada quando acessam itens de dados na DSM. As instâncias dessa camada nos diferentes computadores acessam itens de dados locais e se comunicam, conforme for necessário, para manter a consistência.

Este capítulo se concentra no uso de software para implementar DSM em computadores padrão. Mesmo com suporte de hardware, técnicas de software de alto nível podem ser usadas para minimizar o volume de comunicação entre os componentes de uma implementação de DSM.

A estratégia baseada em paginação tem a vantagem de não impor nenhuma estrutura em particular para a DSM, que aparece como uma seqüência de bytes. Em princípio, ela permite que os programas projetados para um multiprocessador de memória compartilhada sejam executados em

computadores sem memória compartilhada, com pouca ou nenhuma adaptação. Micronúcleos como o Mach e o Chorus fornecem suporte nativo para DSM (e outras abstrações de memória – os recursos da memória virtual Mach estão descritos em www.cdk4.net/mach). Normalmente, a DSM baseada em paginação é implementada principalmente em nível de usuário, para tirar proveito da flexibilidade que isso oferece. A implementação utiliza suporte do núcleo para rotinas de tratamento de erro de páginas em nível de usuário. O UNIX e algumas variantes do Windows fornecem esse recurso. Os microprocessadores com espaços de endereço de 64 bits aumentam o escopo de uma DSM baseada em paginação, diminuindo as restrições sobre o gerenciamento do espaço de endereçamento [Bartoli *et al.* 1993].

O exemplo da Figura 18.2 mostra dois programas em C, *Reader* e *Writer*, que comunicam por intermédio da DSM baseada em paginação fornecida pelo sistema Mether [Minnich e Farber 1989]. O programa *Writer* atualiza dois campos em uma estrutura sobreposta no início do segmento da DSM do Mether (começando no endereço *METHERBASE*) e o programa *Reader* imprime periodicamente os valores que lê nesses campos.

Os dois programas não contêm nenhuma operação especial; eles são compilados em instruções de máquina que acessam um intervalo comum de endereços de memória virtual (começando em *METHERBASE*). O sistema Mether foi executado em uma estação de trabalho Sun e em hardware de rede convencional.

A estratégia de *middleware* é bastante diferente quanto ao uso de hardware especializado e paginação, pois não se destina a utilizar código de memória compartilhada existente. Sua importância é que ela nos permite desenvolver abstrações de nível mais alto dos objetos compartilhados, em vez de posições de memória compartilhada.

```
#include "world.h"
struct shared { int a, b; };
```

Programa *Writer*:

```
main()
{
    struct shared *p;
    methersetup();                          /* Inicializa o runtime Mether */
    p = (struct shared *)METHERBASE;
                        /* estrutura sobreposta no segmento METHER */
    p->a = p->b = 0;                        /* inicializa os campos com zero */
    while(TRUE){                            /* atualiza os campos da estrutura continuamente */
        p ->a = p ->a + 1;
        p ->b = p ->b - 1;
    }
}
```

Programa *Reader*:

```
main()
{
    struct shared *p;
    methersetup();
    p = (struct shared *)METHERBASE;
    while(TRUE){                            /* lê os campos uma vez a cada segundo */
        printf("a = %d, b = %d\n", p ->a, p ->b);
        sleep(1);
    }
}
```

Figura 18.2 Programa no sistema Mether.

18.2 Problemas de projeto e de implementação

Esta seção discute as opções de projeto e implementação relativas aos principais recursos que caracterizam um sistema de DSM. São eles a estrutura de dados mantida na DSM, o modelo de sincronização usado para acessar a DSM de forma consistente no nível da aplicação, o modelo de consistência da DSM que governa a consistência dos valores de dados acessados de diferentes computadores, as opções de atualização para comunicar os valores escritos entre computadores, a granularidade do compartilhamento em uma implementação de DSM e o problema da ultrapaginação (*thrashing*).

18.2.1 Estrutura

No Capítulo 15, consideramos os sistemas que replicam um conjunto de objetos, como agendas e arquivos. Esses sistemas permitem que os programas clientes executem operações sobre os objetos como se houvesse apenas uma cópia de cada objeto, mas na realidade eles podem estar acessando diferentes réplicas físicas. Os sistemas dão garantias sobre até que ponto as réplicas dos objetos podem divergir.

Um sistema de DSM é exatamente como um sistema de replicação. Cada processo enxerga alguma abstração de um conjunto de objetos, mas nesse caso o conjunto é mais ou menos parecido com a memória. Isto é, os objetos podem ser endereçados de uma maneira ou de outra. Diferentes estratégias para a DSM variam em relação ao que consideram como objeto e no modo como os objetos são endereçados. Consideraremos três estratégias, as quais vêem a DSM como sendo composta, respectivamente, de bytes adjacentes, objetos em nível de linguagem ou itens de dados imutáveis.

Orientada a bytes ◊ Este tipo de DSM é acessada como a memória virtual normal – um array de bytes adjacentes. Essa é a visão ilustrada anteriormente pelo sistema Mether. Essa também é a visão de muitos outros sistemas de DSM, incluindo o Ivy, que discutiremos na Seção 18.3. Ela permite que as aplicações (e implementações de linguagem) imponham as estruturas de dados que quiserem na memória compartilhada. Os objetos compartilhados são posições de memória endereçáveis diretamente (na prática, as posições compartilhadas podem ser palavras de vários bytes, em vez de bytes individuais). As únicas operações sobre esses objetos são *read* (ou LOAD) e *write* (ou STORE). Se x e y são duas posições de memória, então denotamos as instâncias dessas operações como segue:

$R(x)a$ – uma operação *read* que lê o valor a de uma posição x.
$W(x)b$ – uma operação *write* que armazena o valor b na posição x.

Um exemplo de execução é $W(x)1$, $R(x)2$. Esse processo grava o valor 1 na posição x e depois lê o valor 2 nela. Entretanto, algum outro processo deve ter gravado o valor 2 nessa posição.

Orientada a objetos ◊ A memória compartilhada é estruturada como um conjunto de objetos em nível de linguagem, com semântica de nível mais alto do que as variáveis *read/write* simples, como pilhas e dicionários. O conteúdo da memória compartilhada só é alterado por invocações a esses objetos e nunca pelo acesso direto às suas variáveis membro. Uma vantagem de considerar a memória dessa maneira é que a semântica do objeto pode ser utilizada ao impor a consistência. A linguagem Orca vê a DSM como um conjunto de objetos compartilhados e, automaticamente, dispõe em série as operações sobre qualquer objeto dado.

Dados imutáveis ◊ Aqui, a DSM é vista como um conjunto de itens de dados imutáveis que os processos podem ler, adicionar e remover. Exemplos incluem Agora [Bisiani e Forin 1988] e, mais significativamente, Linda e seus derivados, TSpaces e JavaSpaces.

Os sistemas do tipo Linda fornecem ao programador coleções de tuplas chamadas de *espaço de tuplas* (veja a Seção 16.3.1). As tuplas consistem em uma seqüência de um ou mais campos de dados tipados, como <"fred", 1958>, <"sid", 1964> e <4, 9.8, "Yes">. Pode existir qualquer combinação de tipos de tuplas no mesmo espaço de tuplas. Os processos compartilham dados acessando o mesmo espaço de tuplas: eles colocam as tuplas no espaço de tuplas usando a operação *read* e as lêem, ou extraem, do espaço de tuplas usando a operação *read* ou *take*. A operação *write* adiciona uma tupla sem afetar as tuplas existentes no espaço. A operação *read* retorna o valor de uma tupla sem afetar o

conteúdo do espaço de tuplas. A operação *take* também retorna uma tupla, mas nesse caso ela também remove a tupla do espaço de tuplas.

Ao ler (*read*), ou pegar (*take*), uma tupla do espaço de tupla, um processo fornece uma especificação de tupla e o espaço de tuplas retorna qualquer tupla que combine com essa especificação – esse é um tipo de endereçamento associativo. Para permitir que os processos sincronizem suas atividades, as operações *read* e *take* são bloqueadas até que exista uma tupla correspondente no espaço de tuplas. Uma especificação de tupla inclui o número de campos e os valores ou tipos exigidos dos campos. Por exemplo, *take*(<String, integer>) poderia extrair <"fred", 1958> ou <"sid", 1964>; *take*(<String, 1958>) extrairia desses dois apenas <"fred", 1958>.

Na linguagem Linda não é permitido nenhum acesso direto às tuplas no espaço de tuplas e os processos precisam substituir as tuplas nesse espaço, em vez de modificá-las. Suponha, por exemplo, que um conjunto de processos mantenha um contador compartilhado no espaço de tuplas. A contagem corrente (digamos, 64) está na tupla <"counter", 64>. Um processo deve executar código da seguinte forma para incrementar o contador em um espaço de tuplas *myTS*:

```
<s, count>:= myTS.take(<"counter", integer>);
myTS.write(<"counter", count+1>);
```

O leitor deve verificar que não surgem condições de corrida, pois a operação *take* extrai a tupla *counter* do espaço de tuplas.

18.2.2 Modelo de sincronização

Muitas aplicações utilizam restrições relativas aos valores armazenados na memória compartilhada. Isso vale tanto para aplicações baseadas em DSM como para aplicações escritas para multiprocessadores de memória compartilhada (ou, na verdade, para qualquer programa concorrente que compartilhe dados, como núcleos de sistema operacional e servidores *multithreadeds*). Por exemplo, se a e b são duas variáveis armazenadas na DSM, então uma restrição poderia ser que $a = b$, sempre. Se dois ou mais processos executam o código a seguir:

```
a:= a + 1;
b:= b + 1;
```

então pode surgir uma inconsistência. Suponha que a e b sejam inicialmente zero e que um processo 1 atribua 1 para a. Antes que possa incrementar b, um processo 2 incrementa a para 2 e atribui 1 para b. A restrição foi violada. A solução é converter esse trecho de código em uma seção crítica: sincronizar os processos para garantir que apenas um possa ser executado por vez.

Então, para usar DSM, precisa ser providenciado um serviço de sincronização distribuído, incluindo construções familiares, como travas e semáforos. Mesmo quando a DSM é estruturada como um conjunto de objetos, os desenvolvedores precisam se preocupar com a sincronização. As construções de sincronização são implementadas usando-se passagem de mensagens (veja no Capítulo 13 uma descrição de um servidor de travas). Instruções de máquina especiais, como *testAndSet*, que são usadas para sincronização em multiprocessadores de memória compartilhada, são aplicáveis à DSM baseada em paginação, mas sua operação no caso distribuído pode ser muito ineficiente. As implementações de DSM tiram proveito da sincronização em nível de aplicação para reduzir o volume de transmissão de atualização. Então, a DSM inclui sincronização como um componente integrado.

18.2.3 Modelo de consistência

Conforme descrevemos no Capítulo 15, o problema da consistência surge para um sistema como a DSM, que replica o conteúdo da memória compartilhada armazenando-o em cache em computadores separados. Na terminologia do Capítulo 15, cada processo tem um gerenciador de réplica local, o qual contém as réplicas dos objetos armazenadas na cache. Na maioria das implementações, por questão de eficiência, os dados são lidos a partir das réplicas locais, mas as atualizações precisam ser propagadas para os outros gerenciadores de réplica.

Hidden page

A reação imediata do leitor ao exemplo que acabamos de dar provavelmente é a de que a implementação de DSM, que inverte a ordem das duas atualizações, está incorreta. Se o processo 1 e o processo 2 fossem executados em conjunto em um computador com um único processador, iríamos supor que o subsistema de memória estaria com problemas. Entretanto, essa pode ser uma implementação correta, no caso distribuído, de um modelo de consistência mais fraca do que muitos de nós intuitivamente esperaria, mas que, contudo, pode ser útil e é relativamente eficiente.

Mosberger [1993] descreve uma variedade de modelos destinados a multiprocessadores de memória compartilhada e sistemas de DSM de software. Os principais modelos de consistência que podem ser praticamente realizados em implementações de DSM são a consistência seqüencial e os modelos baseados em consistência mais fraca.

A principal pergunta a ser feita para caracterizar um modelo de consistência de memória em particular é a seguinte: quando é feito um acesso de leitura a uma posição de memória, quais acessos de escrita à posição, cujos valores poderiam ser fornecidos para a leitura, são candidatos? No extremo mais fraco, a resposta é: qualquer escrita que tenha sido executada antes da leitura. Este modelo seria obtido se os gerenciadores de réplica pudessem atrasar indefinidamente a propagação das atualizações para seus pares. Ele é fraco demais para ser útil.

No extremo mais forte, todos os valores escritos estão instantaneamente disponíveis para todos os processos: uma leitura retorna a escrita mais recente no momento em que a leitura ocorre. Essa definição é problemática por dois motivos. Primeiro, nem as escritas nem as leituras ocorrem em um único ponto no tempo; portanto, o significado de 'mais recente' nem sempre é claro. Cada tipo de acesso tem um ponto de execução bem definido, mas eles terminam em algum momento posterior (por exemplo, após ocorrer a passagem das mensagens). Segundo, o Capítulo 11 mostrou que existem limites sobre a precisão com que os relógios podem ser sincronizados em um sistema distribuído. Portanto, nem sempre é possível determinar precisamente se um evento ocorreu antes de outro.

Contudo, este modelo foi especificado e estudado. O leitor já pode tê-lo reconhecido: ele é o que chamamos de capacidade de linearização no Capítulo 15. A capacidade de linearização é mais comumente chamada de *consistência atômica* na literatura sobre DSM. Vamos agora reformular a definição de capacidade de linearização do Capítulo 15.

Diz-se que um serviço de objeto compartilhado e replicado pode ser linearizado se, *para qualquer execução*, houver alguma interposição das séries de operações executadas por todos os clientes que satisfaça aos dois critérios a seguir:

L1: A seqüência interposta de operações satisfaz a especificação de uma (única) cópia correta dos objetos.

L2: A ordem das operações na interposição é consistente com os tempos reais nos quais as operações ocorreram na execução real.

Essa definição é geral e se aplica a qualquer sistema contendo objetos compartilhados e replicados. Podemos ser mais específicos agora, pois sabemos que estamos tratando com uma memória compartilhada. Considere o caso simples onde a memória compartilhada é estruturada como um conjunto de variáveis que podem ser lidas ou escritas. Todas as operações são leituras e escritas, para as quais apresentamos uma notação na Seção 18.2.1: uma leitura do valor de uma variável x é denotada como $R(x)a$; uma escrita do valor b na variável x é denotada como $W(x)b$. Podemos agora expressar o primeiro critério L1, em termos de variáveis (os objetos compartilhados), como segue:

L1': A seqüência interposta de operações é tal que, se $R(x)a$ ocorre na seqüência, então ou a última operação de escrita que ocorre antes dela na seqüência interposta é $W(x)a$ ou não ocorre nenhuma operação de escrita antes dela e a é o valor inicial de x.

Esse critério expressa nossa intuição de que uma variável só pode ser alterada por uma operação de escrita. O segundo critério da capacidade de linearização, L2, permanece o mesmo.

Consistência seqüencial ◊ A capacidade de linearização é restrita demais para a maioria dos propósitos práticos. O modelo de memória mais forte usado na prática para a DSM é a *consistência seqüencial* [Lamport 1979], que apresentamos no Capítulo 15. Adaptamos a definição do Capítulo 15 para o caso particular das variáveis compartilhadas, como segue.

Diz-se que um sistema de DSM tem consistência seqüencial se, *para qualquer execução*, existe alguma interposição da série de operações executadas por todos os processos que satisfaça os dois critérios a seguir:

CS1: A seqüência interposta de operações é tal que, se $R(x)a$ ocorre na seqüência, então ou a última operação de escrita que ocorre antes dela na seqüência interposta é $W(x)a$ ou não ocorre nenhuma operação de escrita antes dela e a é o valor inicial de x.

CS2: A ordem das operações na interposição é consistente com a ordem do programa em que cada cliente individual as executou.

O critério CS1 é igual ao L1'. O critério CS2 se refere à ordem do programa, em vez da ordem temporal, que é o que torna possível implementar a consistência seqüencial.

A condição pode ser reformulada, como segue: existe uma interposição virtual das operações *read* e *write* de todos os processos sobre uma única imagem virtual da memória; a ordem do programa de cada processo individual é preservada nessa interposição, e cada processo sempre lê o valor mais recente escrito dentro da interposição.

Em uma execução real, as operações de memória podem ser sobrepostas e algumas atualizações podem ter ordem diferente em diversos processos, desde que com isso as restrições da definição não sejam violadas. Note que as operações de memória sobre a DSM precisam ser levadas em conta para satisfazer as condições de consistência seqüencial – e não apenas as operações sobre cada posição individual.

A combinação $ar = 0$, $br = 1$ no exemplo anterior não poderia ocorrer sob consistência seqüencial, pois o processo 1 estaria lendo valores que entrariam em conflito com a ordem do programa do processo 2. Um exemplo de interposição dos acessos à memória dos processos em uma execução com consistência seqüencial aparece na Figura 18.4. Mais uma vez, embora isso mostre uma interposição real das operações de leitura e escrita, a definição estipula apenas que a execução deve acontecer *como se* tal interposição restrita ocorresse.

A DSM com consistência seqüencial poder ser implementada usando-se um único servidor para conter todos os dados compartilhados e fazendo-se todos os processos realizarem leituras ou escritas enviando requisições para o servidor, o qual as ordenaria globalmente. Essa arquitetura é ineficiente demais para uma implementação de DSM, e o significado prático da obtenção da consistência seqüencial está descrito a seguir. Contudo, ele continua sendo um modelo dispendioso para implementar.

Coerência ◊ Uma reação ao custo da consistência seqüencial é estabelecer um modelo mais fraco, com propriedades bem definidas. A *coerência* é um exemplo de uma forma mais fraca de consistência. Sob a coerência, cada processo concorda com a ordem das operações *write* na mesma posição, mas eles não concordam necessariamente com a ordem das operações *write* em posições diferentes. Podemos considerar a coerência como a consistência seqüencial posição por posição. A DSM coerente pode ser implementada pegando-se um protocolo para implementar consistência seqüencial e aplicando-o separadamente em cada unidade de dados replicados – por exemplo, em cada página. A

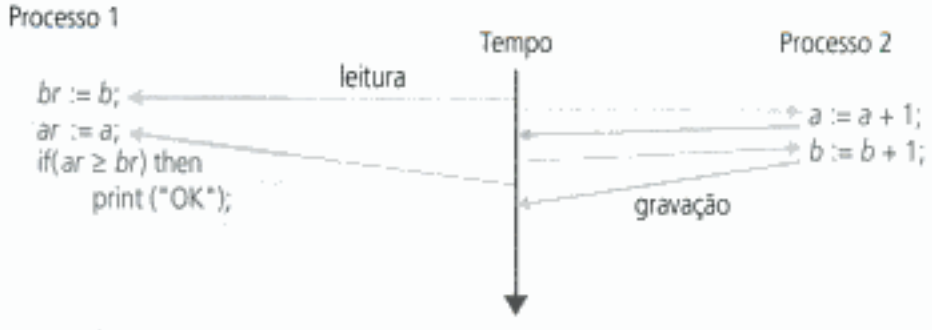


Figura 18.4 Interposição sob a consistência seqüencial.

economia vem do fato de que os acessos a duas páginas diferentes são independentes e não precisam atrasar um ao outro, pois o protocolo é aplicado a eles separadamente.

Consistência fraca ◊ Dubois *et al.* [1988] desenvolveram o modelo de consistência fraca em uma tentativa de evitar os custos da consistência seqüencial em multiprocessadores, enquanto mantinham o *efeito* da consistência seqüencial. Esse modelo explora o conhecimento das operações de sincronização para diminuir a consistência da memória, embora pareça para o programador que é implementado uma consistência seqüencial (pelo menos, sob certas condições que estão fora dos objetivos deste livro). Por exemplo, se o programador usa uma trava para implementar uma seção crítica, então um sistema de DSM pode presumir que nenhum outro processo pode acessar os itens de dados acessados dentro dele sob exclusão mútua. Portanto, é redundante o sistema de DSM propagar atualizações nesses itens até que o processo saia da seção crítica. Embora os itens sejam deixados com valores inconsistentes em parte do tempo, eles não são acessados nesses momentos; a execução parece ter consistência seqüencial. Adve e Hill [1990] descrevem uma generalização dessa noção, chamada de ordenação fraca: (Um sistema de DSM) tem ordenação fraca com relação a um modelo de sincronização se e somente se tiver consistência seqüencial para todo software que obedecer ao modelo de sincronização. A consistência relaxada, que é uma evolução da consistência fraca, será descrita na Seção 18.4.

18.2.4 Opções de atualização

Duas escolhas de implementação principais foram arquitetadas para propagar as atualizações feitas por um processo para os outros: escrita-atualização e escrita-invalidação. Elas são aplicáveis a uma variedade de modelos de consistência de DSM, incluindo a consistência seqüencial. Em linhas gerais, as opções são as seguintes:

Escrita-atualização: as atualizações feitas por um processo são realizadas de modo local e enviadas por *multicast* para todos os outros gerenciadores de réplica que possuam uma cópia do item de dados, os quais modificam imediatamente os dados lidos pelos processos locais (Figura 18.5). Os processos lêem as cópias locais dos itens de dados, sem necessidade de comunicação. Além de permitir vários leitores, diversos processos podem escrever o mesmo item de dados ao mesmo tempo; isso é conhecido como *compartilhamento de vários leitores/vários escritores*.

O modelo de consistência de memória implementado com escrita-atualização depende de vários fatores, principalmente da propriedade de ordenação do *multicast*. A consistência seqüencial pode ser obtida usando-se *multicast* totalmente ordenado (veja no Capítulo 12 uma definição de *multicast* totalmente ordenado), as quais não retornam até que a mensagem de atualização tenha sido distribuída de forma local. Então, todos os processos concordam com a ordem das atualizações. O conjunto de leituras que ocorrem entre quaisquer duas atualizações consecutivas é bem definido e sua ordenação é irrelevante para a consistência seqüencial.

As leituras são computacionalmente baratas na opção de escrita-atualização. Entretanto, o Capítulo 12 mostrou que os protocolos de ordenação de *multicast* são relativamente dispendiosos para se implementar em software. A linguagem Orca usa escrita-atualização e emprega o protocolo de *multicast* do Amoeba [Kaashoek e Tanenbaum 1991] (veja www.cdk4.net/coordination), que usa suporte de hardware para *multicast*. O Munin suporta escrita-atualização como uma opção. Um protocolo de escrita-atualização é usado com suporte de hardware especializado na arquitetura de multiprocessador PLUS.

Escrita-invalidação: esta opção é implementada normalmente na forma de compartilhamento de vários leitores/um escritor. A qualquer momento, um item de dados pode ser acessado no modo somente de leitura por um ou mais processos ou pode ser lido e escrito por um único processo. Um item que é correntemente acessado no modo somente de leitura pode ser copiado indefinidamente em outros processos. Quando um processo tenta escrever nele, primeiramente uma mensagem *multicast* é enviada para todas as outras cópias, para invalidá-las, e isso é reconhecido antes que a escrita possa ocorrer; assim, os outros processos são impedidos de ler dados antigos (isto é, dados que não estão atualizados). Todos os processos que tentarem acessar o item de dados serão

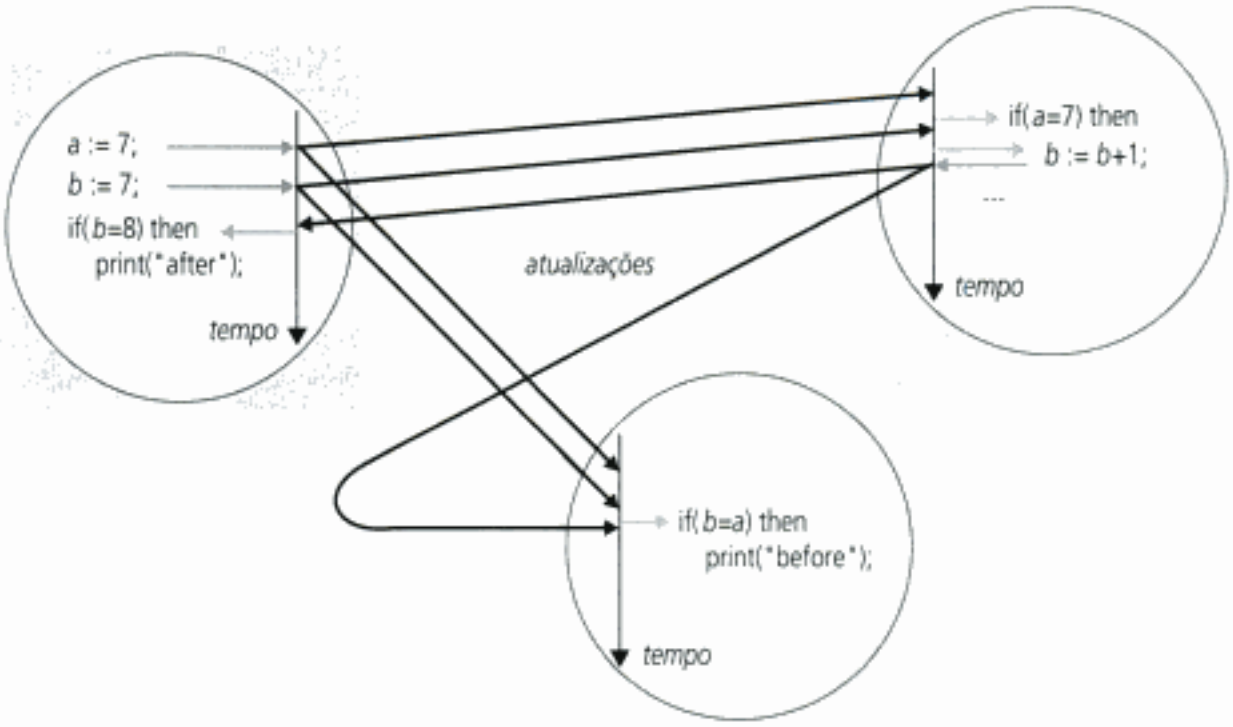


Figura 18.5 **DSM usando escrita-atualização.**

obstruídos, caso exista um gravador. Finalmente, o controle é transferido do processo que está escrevendo e outros acessos podem ocorrer quando a atualização tiver sido enviada. O efeito é processar todos os acessos ao item, na base do primeiro a chegar é o primeiro a ser servido. Pela prova dada por Lamport [1979], esse esquema obtém consistência seqüencial. Na Seção 18.4, veremos que as invalidações podem ser atrasadas sob a consistência relaxada.

Sob o esquema de invalidação, as atualizações só são propagadas quando dados são lidos e várias atualizações podem ocorrer antes que a comunicação seja necessária. Contra isso deve ser colocado o custo da invalidação de cópias somente de leitura antes que uma escrita possa ocorrer. No esquema de vários leitores/um escritor, isso é potencialmente dispendioso. Mas se a relação leitura/escrita for suficientemente alta, então o paralelismo obtido pelo fato de permitir a existência de vários leitores simultâneos compensa esse custo. Onde a relação leitura/escrita é relativamente pequena, um esquema de um leitor/um escritor pode ser mais apropriado: isto é, um esquema no qual no máximo um processo por vez pode ter acesso somente de leitura garantido.

18.2.5 Granularidade

Um problema relacionado à estrutura da DSM é a granularidade do compartilhamento. Conceitualmente, todos os processos compartilham o conteúdo inteiro de uma DSM. Entretanto, à medida que os programas que compartilham a DSM são executados, somente certas partes dos dados são realmente compartilhadas e apenas por certos momentos, durante a execução. Seria claramente muito desperdício o fato da implementação de DSM sempre transmitir o conteúdo inteiro da DSM à medida que os processos a acessam e atualizam. Qual deve ser a unidade de compartilhamento em uma implementação de DSM? Isto é, quando um processo tiver escrito na DSM, quais dados o *runtime* da DSM envia para fornecer valores consistentes para outras partes?

Focalizaremos aqui as implementações baseadas em paginação, embora a questão da granularidade surja em outras implementações (veja o Exercício 18.11). Em uma DSM baseada em paginação, o hardware suporta eficientemente alterações em um espaço de endereçamento, em unidades de páginas

Hidden page

18.3 Consistência seqüencial e estudo de caso (Ivy)

Esta seção descreve métodos de implementação da DSM baseada em paginação com consistência seqüencial. Ela usa o Ivy [Li e Hudak 1989] como estudo de caso.

18.3.1 O modelo de sistema

O modelo básico a ser considerado é aquele no qual um conjunto de processos compartilha um segmento de DSM (Figura 18.7). O segmento é mapeado no mesmo intervalo de endereçamento em cada processo, de modo que valores válidos de ponteiros podem ser armazenados no segmento. Os processos são executados em computadores equipados com uma unidade de gerenciamento de memória paginada. Vamos supor que exista apenas um processo por computador que acessa o segmento da DSM. Na realidade, pode haver vários processos desses em um computador. Entretanto, eles poderiam compartilhar páginas da DSM diretamente (o mesmo quadro de página – *frame* – pode ser usado nas tabelas de página utilizadas pelos diferentes processos). A única complicação seria coordenar a busca e a propagação das atualizações em uma página, quando dois ou mais processos locais a acessassem. Esta descrição ignora tais detalhes.

A paginação é transparente para os componentes da aplicação dentro dos processos; eles podem ler e escrever quaisquer dados logicamente na DSM. Entretanto, o *runtime* da DSM restringe as permissões de acesso à página para manter a consistência seqüencial ao processar leituras e escritas. As unidades de gerenciamento de memória paginada possibilitam que as permissões de acesso a uma página de dados sejam configuradas como *nenhuma*, *somente de leitura* ou *leitura-escrita*. Se um processo exceder as permissões correntes de acesso, recebe um erro de leitura, de escrita ou um erro de página, de acordo com o tipo de acesso. O núcleo redireciona o erro para uma rotina de tratamento especificada pela camada de *runtime* da DSM em cada processo. A rotina de tratamento de erro – que é executada de forma transparente para a aplicação – processa o erro de uma maneira especial, a ser descrita a seguir, antes de retornar o controle para a aplicação. Nos sistemas de DSM originais, como o Ivy, o próprio núcleo realiza grande parte do processamento que descrevemos aqui. Falaremos dos próprios processos realizando o tratamento de erros de página e de comunicação. Na realidade, uma combinação da camada de *runtime* da DSM no processo e o núcleo executam essas funções de tratamento. Normalmente, o *runtime* da DSM contém a funcionalidade mais significativa, para que isso possa ser reimplementado e otimizado sem os problemas associados à alteração de um núcleo.

Esta descrição ignorará o processamento de erro de página que ocorre como parte da implementação de memória virtual normal. Fora o fato de que os segmentos da DSM competem com outros segmentos pelos quadros de página, as implementações são independentes.

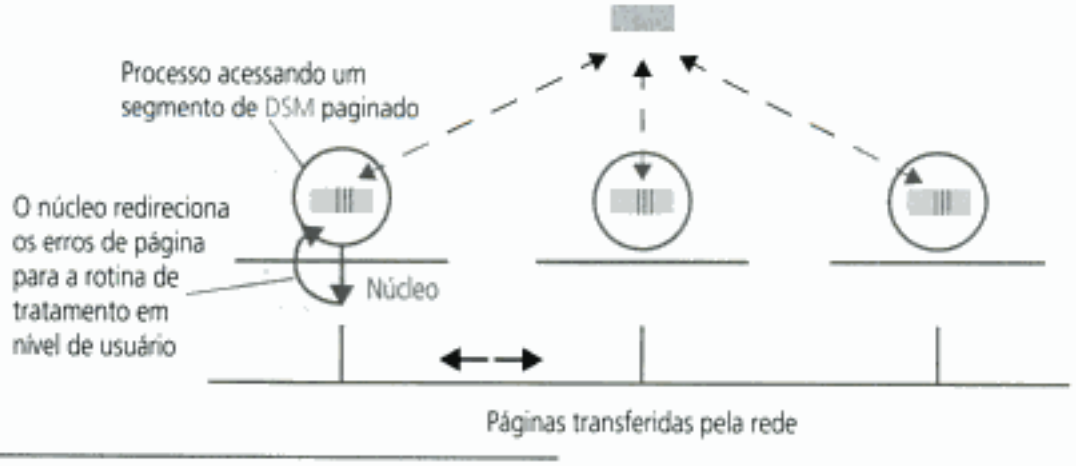
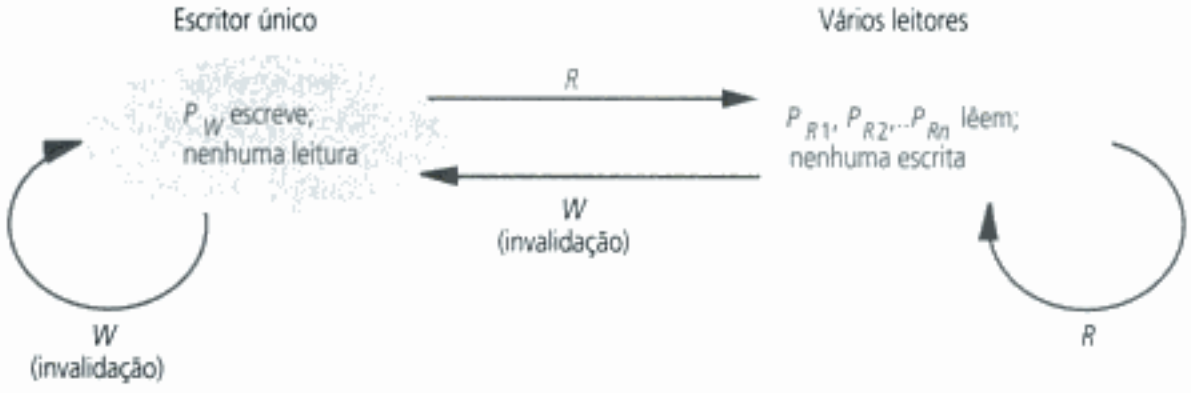


Figura 18.7 Modelo de sistema para DSM baseada em paginação.

Hidden page



Nota: R = ocorre erro de leitura; W = ocorre erro de escrita.

Figura 18.8 **Transições de estado sob escrita-invalidação.**

Note que dois ou mais processos com cópias somente de leitura podem receber erros de escrita mais ou menos ao mesmo tempo. Uma cópia somente de leitura de uma página pode estar desatualizada quando a posse for finalmente garantida. Para detectar se uma cópia corrente somente de leitura de uma página está desatualizada, cada página pode ser associada a um número em seqüência, o qual é incrementado quando a posse é transferida. Um processo que exija acesso de escrita inclui o número da seqüência de sua cópia somente de leitura, caso possua um. O proprietário corrente pode então identificar se a página foi modificada e, portanto, precisa ser enviada. Esse esquema é descrito por Kessler e Livny [1989] como algoritmo astuto.

Quando um processo P_R tenta ler uma página p para a qual não possui permissões de acesso, ocorre um erro de leitura de página. O procedimento de tratamento de erro de página é o seguinte:

- A página é copiada de *owner*(p) para P_R.
- Se o proprietário corrente for um escritor único, então ele permanecerá como proprietário de p e sua permissão de acesso para p será configurada como acesso somente de leitura. Manter o acesso de leitura é desejável para o caso do processo tentar ler a página subseqüentemente – ele terá mantido uma versão atualizada da página. Entretanto, como proprietário, ele terá de processar as requisições subseqüentes da página, mesmo que não acesse a página novamente. Portanto, poderia ser mais apropriado reduzir a permissão para nenhum acesso e transferir a posse para P_R.
- $copyset(p) := copyset(p) \cup \{P_R\}$.
- A camada de *runtime* da DSM em P_R coloca a página com permissões somente de leitura no local apropriado em seu espaço de endereçamento e reinicia a instrução que causou a falha.

É possível que um segundo erro de página ocorra durante os algoritmos de transição que acabamos de descrever. Para que as transições ocorram de forma consistente, qualquer nova requisição da página só será processada depois que a transição corrente tiver terminado.

A descrição que acabamos de dar explicou apenas o que deve ser feito. O problema de *como* implementar eficientemente o tratamento do erro de página será tratado agora.

18.3.3 Protocolos de invalidação

Dois problemas importantes ainda precisam ser resolvidos em um protocolo para implementar o esquema de invalidação:

1. Como localizar *owner*(p) para determinada página p.
2. Onde armazenar *copyset*(p).

Hidden page

dos outros. Agora, descreveremos o gerenciamento baseado em *multicast* e o gerenciamento distribuído dinâmico.

Usando *multicast* para localizar o proprietário ◊ O *multicast* pode ser usado para eliminar o gerenciador completamente. Quando um processo provoca um erro, ele envia por *multicast* sua requisição de página para todos os outros processos. Somente o processo que possui a página responde. Deve-se tomar o cuidado de garantir o comportamento correto, caso dois clientes solicitem a mesma página mais ou menos ao mesmo tempo: finalmente, cada cliente deve obter a página, mesmo que sua requisição seja feita durante a transferência da posse.

Considere dois clientes C_1 e C_2, que usam *multicast* para localizar uma página pertencente a O. Suponha que O receba primeiro a requisição de C_1 e transfira a posse para ele. Antes que a página chegue, a requisição de C_2 chega em O e em C_1. O descartará a requisição de C_2, pois não possui mais a página. Li e Hudak mostraram que C_1 deve adiar o processamento da requisição de C_2 para depois que tenha obtido a página – caso contrário, ele descartaria a requisição por não ser o proprietário e a requisição de C_2 seria completamente perdida. Entretanto, ainda resta um problema. Nesse meio-tempo, a requisição de C_1 foi enfileirada em C_2. Após C_1 ter finalmente concedido a página para C_2, C_2 receberá e processará a requisição de C_1 – que agora é obsoleta!

Uma solução é usar *multicast* totalmente ordenado, para que os clientes possam descartar com segurança as requisições que chegam antes da sua própria (as requisições são enviadas para eles mesmos, assim como para outros processos). Outra solução, que usa um *multicast* não ordenado computacionalmente mais barato, mas que consome mais largura de banda, é associar cada página a uma indicação de tempo vetorial, com uma entrada por processo (veja no Capítulo 11 uma descrição das indicações de tempo vetoriais). Quando a posse da página é transferida, a indicação de tempo também é. Quando um processo obtém a posse, ele incrementa sua entrada na indicação de tempo. Quando um processo solicita a posse, ele inclui a última indicação de tempo que mantinha para a página. Em nosso exemplo, C_2 poderia descartar a requisição de C_1, pois a entrada de C_1 na indicação de tempo da requisição é menor do que aquela que chegou com a página.

Seja usado *multicast* ordenado ou não ordenado, esse esquema tem a desvantagem usual dos esquemas de *multicast*: os processos que não são os proprietários de uma página são interrompidos por mensagens irrelevantes, desperdiçando tempo de processamento.

18.3.4 Um algoritmo de gerenciador distribuído dinâmico

Li e Hudak sugeriram o algoritmo do gerenciador distribuído dinâmico, o qual permite que a posse da página seja transferida entre processos, mas que usa uma alternativa ao *multicast* como método de localização do proprietário de uma página. A idéia é dividir as sobrecargas da localização de páginas entre os computadores que as acessam. Para cada página p, todo processo mantém uma 'dica' do proprietário corrente da página – o provável proprietário de p, ou *probOwner*(p). Inicialmente, todo processo recebe localizações de página precisas. Em geral, entretanto, esses valores são *dicas*, pois as páginas podem ser transferidas para qualquer lugar, a qualquer momento. Assim como nos algoritmos anteriores, a posse é transferida apenas quando ocorre um erro de escrita.

O proprietário de uma página é localizado seguindo-se os encadeamentos de dicas que são estabelecidos quando a posse da página é transferida de um computador para outro. O comprimento do encadeamento – isto é, o número de mensagens de encaminhamento necessárias para localizar o proprietário – ameaça a aumentar indefinidamente. O algoritmo supera isso atualizando as dicas à medida que valores mais atualizados se tornam disponíveis. As dicas são atualizadas e as requisições são encaminhadas, como segue:

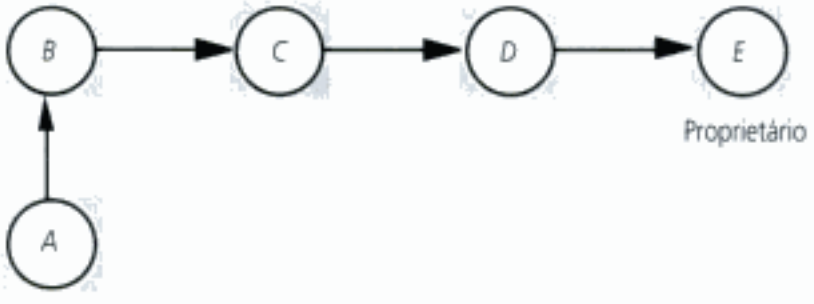
- Quando um processo transfere a posse da página p para outro processo, ele atualiza *probOwner*(p) para ser o destinatário.
- Quando um processo trata de uma requisição de invalidação de uma página p, ele atualiza *probOwner*(p) para ser o solicitante.
- Quando um processo que solicitou acesso de leitura para uma página p o recebe, ele atualiza *probOwner*(p) para ser o provedor.

- Quando um processo recebe uma requisição de uma página *p* que não possui, ele encaminha a requisição para *probOwner*(*p*) e reconfigura *probOwner*(*p*) para ser o solicitante.

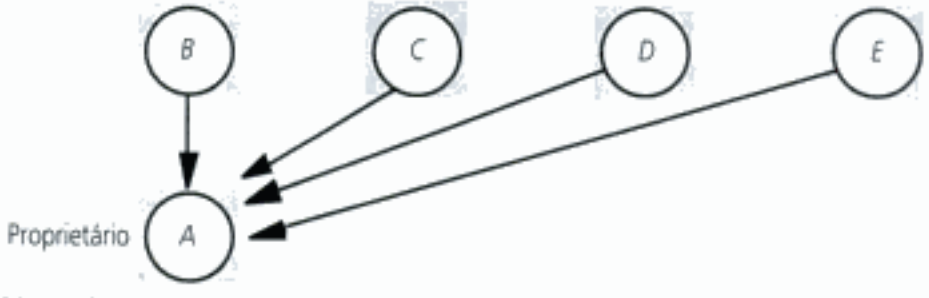
As três primeiras atualizações resultam simplesmente do protocolo para transferir a posse da página e fornecer cópias somente de leitura. O fundamento lógico da atualização no encaminhamento de requisições é que, para requisições de escrita, em breve o solicitante será o proprietário, mesmo que não seja correntemente. Na verdade, no algoritmo de Li e Hudak presumido aqui, a atualização *probOwner* é feita, seja a requisição de acesso para leitura ou para escrita. Voltaremos a esse ponto em breve.

A Figura 18.10 ((a) e (b)) ilustra ponteiros *probOwner* antes e depois do processo *A* receber um erro de escrita de página. O ponteiro *probOwner* de *A* da página aponta inicialmente para *B*. Os processos *B*, *C* e *D* encaminham a requisição para *E*, seguindo seus próprios ponteiros *probOwner*; daí em diante, todos são configurados para apontar para *A*, como resultado das regras de atualização que acabamos de descrever. O arranjo após o tratamento da falha é claramente melhor do que aquele que o precedeu: o encadeamento de ponteiros diminuiu.

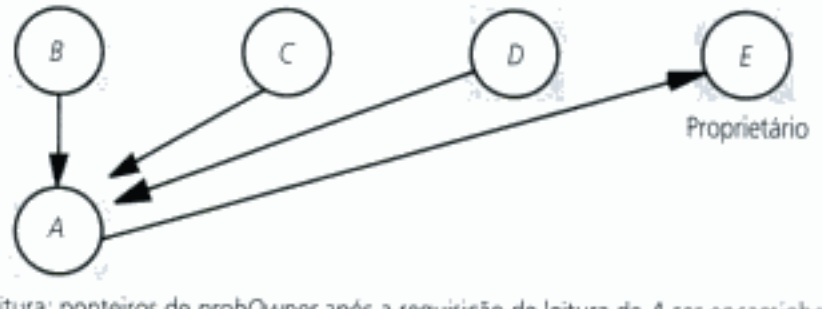
Entretanto, se *A* receber um erro de leitura, então o processo *B* estará em melhor situação (duas etapas, em vez de três para *E*), a situação de *C* será a mesma de antes (duas etapas), mas *D* estará em pior situação, com duas etapas, em vez de uma (Figura 18.10(c)). São exigidas simulações para investigar o efeito global dessa tática para obter desempenho.



(a) Ponteiros de *probOwner* imediatamente antes que o processo *A* receba um erro de página de uma página pertencente a *E*



(b) Erro de escrita: ponteiros de *probOwner* após a requisição de escrita de *A* ser encaminhada



(c) Erro de leitura: ponteiros de *probOwner* após a requisição de leitura de *A* ser encaminhada

Figura 18.10 Atualizando ponteiros de *probOwner*.

O comprimento médio dos encadeamentos de ponteiro pode ser melhor controlado transmitindo-se periodicamente a localização do proprietário corrente para todos os processos. Isso tem o efeito de reduzir todos os encadeamentos para o comprimento 1.

Li e Hudak descrevem os resultados das simulações que realizaram para investigar a eficácia de suas atualizações de ponteiro. Com processos que causam erro escolhidos aleatoriamente, para 1024 processadores eles verificaram que o número médio de mensagens exigidas para se chegar ao proprietário de uma página foi de 2,34, caso as transmissões anunciando a localização do proprietário sejam feitas a cada 256 erros, e de 3,64, se as transmissões forem feitas a cada 1024 erros. Esses valores são dados apenas como ilustrações: um conjunto completo dos resultados é dado por Li e Hudak [1989]. Note que um sistema de DSM que usa um gerenciador central exige duas mensagens para chegar ao proprietário de uma página.

Finalmente, Li e Hudak descrevem uma otimização que potencialmente torna a invalidação mais eficiente e reduz o número de mensagens exigidas para tratar de um erro de leitura de página. Em vez de obter uma cópia da página do proprietário de uma página, um cliente pode obter uma cópia de qualquer processo que possua uma cópia válida. Existe a chance de que um cliente que esteja tentando localizar o proprietário encontre tal processo antes do proprietário no encadeamento de ponteiros.

Isso é feito com a condição de que os processos mantenham um registro dos clientes que obtiveram uma cópia de uma página deles. O conjunto de processos que possuem cópias somente de leitura de uma página forma, assim, uma árvore cuja raiz é o proprietário, com cada nó apontando para os descendentes mais abaixo, os quais obtiveram cópias dele. A invalidação de uma página começa no proprietário e desce na árvore. Ao receber uma mensagem de invalidação, um nó a encaminha para seus descendentes, além de invalidar sua própria cópia. O efeito global é que algumas invalidações ocorrem em paralelo. Isso pode reduzir o tempo global gasto para invalidar uma página – especialmente em um ambiente sem suporte de hardware para *multicast*.

18.3.5 Ultrapaginação

Pode ser argumentado que é responsabilidade do programador evitar a ultrapaginação. O programador poderia anotar itens de dados para ajudar o *runtime* da DSM na minimização das cópias de página e das transferências de posse. Esta última estratégia será discutida na próxima seção, no contexto do sistema de DSM Munin.

O Mirage [Fleisch e Popek 1989] adota uma estratégia de ultrapaginação que se destina a ser transparente para os programadores. O Mirage associa cada página a um pequeno intervalo de tempo. Quando um processo tiver acesso a uma página, ele poderá manter o acesso pelo intervalo determinado, o qual serve como um tipo de fatia de tempo. Nesse meio-tempo, outras requisições para a página são refreadas. Uma desvantagem óbvia desse esquema é que é muito difícil escolher o tamanho da fatia de tempo. Se o sistema usar um período de tempo escolhido estaticamente, ele estará sujeito a ser inadequado em muitos casos. Um processo poderia, por exemplo, escrever uma página apenas uma vez e, daí em diante, não acessá-la mais; contudo, outros processos seriam impedidos de acessá-la. Igualmente, o sistema poderia garantir o acesso à página a outro processo, antes de ter terminado de usá-la.

Um sistema de DSM poderia escolher o tamanho da fatia de tempo dinamicamente. Uma possível base para isso é a observação dos acessos à página (usando os bits de referência da unidade de gerenciamento de memória). Outro fator que poderia ser levado em conta é o comprimento da fila de processos que estão esperando pela página.

18.4 Consistência relaxada e estudo de caso (Munin)

Os algoritmos da seção anterior foram projetados para obter DSM com consistência seqüencial. A vantagem da consistência seqüencial é que a DSM se comporta da maneira que os programadores esperam que a memória compartilhada se comporte. Sua desvantagem é que ela é dispendiosa para implementar. Freqüentemente, os sistemas de DSM exigem o uso de *multicast* em suas implementa-

ções, seja usando escrita-atualização ou escrita-invalidação – embora o *multicast* não ordenado seja suficiente para a invalidação. A localização do proprietário de uma página tende a ser dispendiosa: um gerenciador central que conheça a localização do proprietário de cada página age como um gargalo; em média, seguir ponteiros envolve mais mensagens. Além disso, os algoritmos baseados em invalidação podem ocasionar a ultrapaginação.

A consistência relaxada foi introduzida com o multiprocessador Dash, o qual implementa DSM em hardware usando um protocolo de escrita-invalidação [Lenoski *et al.* 1992]. O Munin e o Treadmarks [Keleher *et al.* 1992] adotaram uma implementação de software. A consistência relaxada é mais fraca do que a consistência seqüencial e computacionalmente mais barata de implementar, e tem uma semântica razoável, fácil para os programadores tratarem.

A idéia da consistência relaxada é reduzir as sobrecargas da DSM explorando o fato de que os programadores usam objetos de sincronização, como semáforos, travas e barreiras. Uma implementação de DSM pode usar o conhecimento dos acessos a esses objetos para permitir que a memória se torne inconsistente em certos pontos, enquanto, contudo, o uso de objetos de sincronização preserva a consistência em nível de aplicação.

18.4.1 Acessos à memória

Para entendermos a consistência relaxada – ou qualquer outro modelo de memória que leve em conta a sincronização –, começaremos classificando os acessos à memória de acordo com sua função, se houver, na sincronização. Além disso, vamos discutir como os acessos à memória podem ser realizados de forma assíncrona para obter ganhos de desempenho e fornecer um modelo operacional simples de como os acessos à memória entram em vigor.

Conforme dissemos anteriormente, por motivos de eficiência, as implementações de DSM em sistemas distribuídos de propósito geral podem usar passagem de mensagens, em vez de variáveis compartilhadas, para implementar a sincronização. Mas ter em mente a sincronização baseada em variáveis compartilhadas pode ajudar na discussão a seguir. O pseudo-código seguinte implementa travas usando a operação *testAndSet* nas variáveis. A função *testAndSet* põe a trava em 1 e retorna 0 se o encontrar como zero; caso contrário, ela retornará 1. Ela faz isso de forma atômica.

```
acquireLock(var int lock):          // a trava é passada por referência
        while (testAndSet(lock) = 1)
            skip;
releaseLock(var int lock):          // a trava é passada por referência
        lock:= 0;
```

Tipos de acesso à memória ◊ A principal distinção é entre *acessos concorrentes* e *acessos não concorrentes* (*normais*). Dois acessos são concorrentes, se:

- eles podem ocorrer concorrentemente (não existe nenhuma ordem imposta entre eles) e
- pelo menos um é uma *escrita*.

Portanto, duas operações *de leitura* nunca podem ser concorrentes; uma *leitura* e uma *escrita* feitas no mesmo local por dois processos que se sincronizam entre as operações (e, portanto, as ordenam) são não concorrentes.

Dividimos ainda os acessos concorrentes em acessos *sincronizados* e *não-sincronizados:*

- os acessos sincronizados são operações *de leitura* ou *de escrita* que contribuem para a sincronização;
- os acessos não-sincronizados são operações *de leitura* ou *de escrita* concorrentes, mas que não contribuem para a sincronização.

A operação de *escrita* indicada por '*lock*:= 0' em *releaseLock* (acima) é um acesso sincronizado. O mesmo se dá com a operação de *leitura* implícita em *testAndSet*.

Os acessos sincronizados são concorrentes, pois potencialmente os processos com sincronização devem ser capazes de acessar variáveis de sincronização concorrentemente e devem atualizá-las: ope-

Hidden page

A memória com consistência relaxada é projetada para satisfazer esses requisitos. Gharachorloo *et al.* [1990] definem a consistência relaxada como segue:

CR1: antes que uma operação *de leitura* ou *de escrita* possa ser executada com relação a qualquer outro processo, todos os acessos de *aquisição* anteriores devem ser realizados.

CR2: antes que uma operação *de liberação* possa ser executada com relação a qualquer outro processo, todas as operações *de leitura* e *de escrita* anteriores devem ser executadas.

CR3: as operações *de aquisição* e *de liberação* têm consistência seqüencial uma com relação à outra.

CR1 e CR2 garantem que, quando ocorre uma liberação, nenhum outro processo que esteja adquirindo uma trava pode ler versões antigas de dados modificados pelo processo que realiza a liberação. Isso é consistente com a expectativa do programador de que a liberação de uma trava, por exemplo, significa que um processo terminou de modificar dados dentro de uma seção crítica.

O *runtime* da DSM só pode impor a garantia da consistência relaxada se souber dos acessos sincronizados. No Munin, por exemplo, o programador é obrigado a usar as primitivas *acquireLock*, *releaseLock* e *waitAtBarrier* do próprio Munin. (Uma barreira é um objeto de sincronização que bloqueia cada processo de um conjunto de processos até que todos tenham esperado por ele; então, todos os processos continuam.) Um programa deve usar sincronização para garantir que as atualizações se tornem visíveis para os outros processos. Dois processos que compartilham a DSM, mas nunca usam objetos de sincronização, poderão nunca ver as atualizações um do outro se a implementação aplicar rigorosamente a garantia única dada anteriormente.

Note que o modelo de consistência relaxada permite que uma implementação empregue algumas operações assíncronas. Por exemplo, um processo não precisa ser bloqueado ao fazer atualizações dentro de uma seção crítica. Nem suas atualizações precisam ser propagadas, até que ele deixe a seção crítica, liberando uma trava. Além disso, as atualizações podem então ser coletadas e enviadas em uma única mensagem. Somente a atualização final em cada item de dados precisa ser enviada.

Considere os processos da Figura 18.12, que adquirem e liberam uma trava para acessar duas variáveis, a e b (a e b são inicializadas com zero). O processo 1 atualiza a e b sob condições de exclusão mútua, de modo que o processo 2 não pode ler a e b ao mesmo tempo e de modo que encontre $a = b = 0$ ou $a = b = 1$. As seções críticas impõem a consistência – a igualdade de a e b – *no nível da aplicação*. É redundante propagar atualizações nas variáveis afetadas durante a seção crítica. Se o processo 2 tentar acessar a, digamos, fora de uma seção crítica, então ele poderá encontrar um valor antigo. Essa é uma questão para o desenvolvedor da aplicação.

Vamos supor que o processo 1 adquira a trava primeiro. O processo 2 será bloqueado e não causará nenhuma atividade relacionada à DSM até ter adquirido a trava e tentar acessar a e b. Se os dois processos fossem executados em uma memória com consistência seqüencial, então o processo 1 seria bloqueado quando atualizasse a e b. Sob um protocolo de escrita-atualização, ele seria bloqueado enquanto todas as versões dos dados fossem atualizadas; sob um protocolo de escrita-invalidação, ele seria bloqueado enquanto todas as cópias fossem invalidadas.

```
Processo 1:
        acquireLock();          // entra na seção crítica
        a:= a+ 1;
        b:= b+ 1;
        releaseLock();          // sai da seção crítica

Processo 2:
        acquireLock();          // entra na seção crítica
        print ("Os valores de a e b são: ", a, b);
        releaseLock();          // sai da seção crítica
```

Figura 18.12 Processos executando em uma DSM com consistência relaxada.

Hidden page

Somente de leitura: nenhuma atualização pode ser feita após a inicialização e o item pode ser copiado livremente.

Migratório: normalmente, os processos fazem vários acessos a um item e pelo menos um dos quais é uma atualização. Por exemplo, o item poderia ser acessado dentro de uma seção crítica. O Munin sempre fornece acesso de leitura e escrita em conjunto para tal objeto, mesmo quando um processo recebe um erro de leitura. Isso economiza processamento de erro de escrita subseqüente.

Escrita compartilhada: vários processos atualizam o mesmo item de dados (por exemplo, um array) concorrentemente, mas essa anotação é uma declaração do programador dizendo que os processos não atualizam as mesmas partes dele. Isso significa que o Munin pode evitar o falso compartilhamento, mas deve propagar apenas as palavras do item de dados que são realmente atualizadas em cada processo. Para isso, o Munin faz uma cópia de uma página (dentro de uma rotina de tratamento de erro de escrita), imediatamente antes de ela ser atualizada de forma local. Apenas as diferenças entre as duas versões são enviadas em uma atualização.

Produtor-consumidor: o objeto de dados é compartilhado por um conjunto fixo de processos, mas apenas um o atualiza. Conforme explicamos ao discutirmos a ultrapaginação, anteriormente, um protocolo de escrita-atualização é o mais conveniente aqui. Além disso, as atualizações podem ser retardadas sob o modelo de consistência relaxada, supondo que os processos usem travas para sincronizar seus acessos.

Redução: o item de dados é sempre modificado, sendo bloqueado, lido, atualizado e desbloqueado. Um exemplo disso é um mínimo global em uma computação paralela, o qual deve ser acessado e modificado de forma atômica, se for maior do que o mínimo local. Esses itens são armazenados em um proprietário fixo. As atualizações são enviadas para o proprietário, o qual as propaga.

Resultado: vários processos atualizam diferentes palavras dentro do item de dados; um único processo lê o item inteiro. Por exemplo, diferentes processos operários poderiam preencher diferentes elementos de um array, o qual é então processado por um processo mestre. O ponto aqui é que as atualizações só precisam ser propagadas para o mestre e não para os operários (como ocorreria sob a anotação com a escrita compartilhada que acabamos de descrever).

Convencional: o item de dados é gerenciado sob um protocolo de invalidação semelhante àquele descrito na seção anterior. Portanto, nenhum processo pode ler uma versão antiga do item de dados.

Carter *et al.* [1991] detalham as opções de parâmetro usadas para cada uma das anotações que fornecemos. Esse conjunto de anotações não é fixo. Outras podem ser criadas, à medida que padrões de compartilhamento exigindo diferentes opções de parâmetro são encontrados.

18.5 Outros modelos de consistência

Os modelos de consistência de memória podem ser divididos em *modelos uniformes*, que não distinguem entre os tipos de acesso à memória, e *modelos mistos*, que distinguem entre acessos normais e sincronizados (assim como outros tipos de acesso).

Existem vários modelos uniformes que são mais fracos do que a consistência seqüencial. Apresentamos a coerência, na Seção 18.2.3, na qual a memória tem consistência seqüencial de acordo com o local. Os processadores concordam com a ordem de todas as escritas em determinado local, mas eles podem diferir na ordem das escritas de diferentes processadores em diferentes locais [Goodman 1989, Gharachorloo *et al.* 1990].

Outros modelos de consistência uniformes incluem:

Consistência causal: as leituras e escritas podem ter o relacionamento antes do acontecido (veja o Capítulo 11). Isso é definido como válido entre operações de memória quando (a) elas são executadas pelo mesmo processo, (b) um processo lê um valor escrito por outro processo ou (c) exis-

te uma seqüência dessas operações vinculando as duas operações. A restrição do modelo é que o valor retornado por uma leitura deve ser consistente com o relacionamento antes do acontecido. Isso é descrito por Hutto e Ahamad [1990].

Consistência de processador: a memória é coerente e obedece o modelo de RAM em pipeline (veja a seguir). A maneira mais simples de pensar na consistência de processador é que a memória é coerente e que todos os processos concordam com a ordem de quaisquer dois acessos de escrita feitos pelo mesmo processo – isto é, eles concordam com sua ordem de programa. Isso foi definido informalmente pela primeira vez por Goodman [1989] e, posteriormente, foi definido formalmente por Gharachorloo *et al.* [1990] e Ahamad *et al.* [1992].

RAM em pipeline: todos os processadores concordam com a ordem das escritas realizadas por qualquer processador dado [Lipton e Sandberg 1988].

Além da consistência relaxada, os modelos mistos incluem:

Consistência de entrada: a consistência de entrada foi proposta para o sistema de DSM Midway [Bershad *et al.* 1993]. Nesse modelo, cada variável compartilhada é vinculada a um objeto de sincronização, como uma trava, o qual governa o acesso a essa variável. É garantido que qualquer processo que adquira a trava, primeiro leia o valor mais recente da variável. Um processo que queira gravar a variável deve primeiro obter a trava correspondente no modo exclusivo – tornando-o o único processo capaz de acessar a variável. Vários processos podem ler a variável concorrentemente, mantendo a trava no modo não exclusivo. O Midway evita a tendência do falso compartilhamento na consistência relaxada, mas às custas de uma maior complexidade na programação.

Consistência de escopo: este modelo de memória [Iftode *et al.* 1996] tenta simplificar o modelo de programação de consistência de entrada. Na consistência de escopo, as variáveis são associadas a objetos de sincronização de forma automática, em vez de contar com o programador para associar travas às variáveis explicitamente. Por exemplo, o sistema pode monitorar quais variáveis são atualizadas em uma seção crítica.

Consistência fraca: a consistência fraca [Dubois *et al.* 1988] não distingue entre acessos com sincronização de *aquisição* e *liberação*. Uma de suas garantias é que todos os acessos normais anteriores terminam antes que *um dos dois* tipos de acesso com sincronização termine.

Discussão ◊ A consistência relaxada, e alguns dos outros modelos de consistência mais fracos do que a consistência seqüencial, parecem ser os mais promissores para a DSM. Não parece ser uma desvantagem significativa do modelo de consistência relaxada o fato das operações com sincronização precisarem ser conhecidas pelo *runtime* da DSM – desde que aquelas fornecidas pelo sistema sejam suficientemente poderosas para atender as necessidades dos programadores.

É importante perceber que, sob os modelos mistos, a maioria dos programadores não é obrigada a considerar a semântica de consistência de memória em particular usada, desde que sincronize seus acessos aos dados adequadamente. Mas existe um perigo geral nos projetos de DSM, de pedir ao programador para que faça muitas anotações em seu programa para tornar sua execução eficiente. Isso inclui tanto anotações identificando itens de dados com objetos de sincronização como anotações de compartilhamento, como aquelas do Munin. Supõe-se que uma das vantagens da programação com memória compartilhada em relação à passagem de mensagens é sua conveniência relativa.

18.6 Resumo

Este capítulo descreveu e motivou o conceito de memória compartilhada distribuída como uma abstração de memória compartilhada, que é uma alternativa à comunicação baseada em mensagens em um sistema distribuído. A DSM se destina principalmente ao processamento paralelo e ao compartilhamento de dados. Foi mostrado que ela funciona tão bem quanto a passagem de mensagens para certas aplicações paralelas, mas é difícil de implementar eficientemente e seu desempenho varia com as aplicações.

O capítulo se concentrou nas implementações de software da DSM – particularmente aquelas que usam o subsistema de memória virtual – mas ela tem sido implementada com suporte de hardware.

Os principais problemas de projeto e implementação são a estrutura da DSM, os meios pelos quais as aplicações são sincronizadas, o modelo de consistência de memória, o uso de protocolos de escrita-atualização ou escrita-invalidação, a granularidade do compartilhamento e a ultrapaginação.

A DSM é estruturada como uma série de bytes, como um conjunto de objetos compartilhados ou como um conjunto de itens de dados imutáveis, como as tuplas.

As aplicações que usam DSM exigem sincronização para satisfazer as restrições de consistência específicas da aplicação. Para isso, elas usam objetos, como travas, implementados usando passagem de mensagens por motivos de eficiência.

O tipo restrito mais comum de consistência de memória implementado nos sistemas de DSM é a consistência seqüencial. Devido ao seu custo, foram desenvolvidos modelos de consistência mais fracos, como a coerência e a consistência relaxada. A consistência relaxada permite que a implementação explore o uso de objetos de sincronização para obter maior eficiência, sem violar as restrições de consistência em nível da aplicação. Vários outros modelos de consistência foram mencionados, incluindo consistência de entrada, de escopo e fraca, todos os quais exploram a sincronização.

Os protocolos de escrita-atualização são aqueles nos quais as atualizações são propagadas em todas as cópias quando os itens de dados são atualizados. Normalmente, eles são implementados em hardware, embora existam implementações de software usando *multicast* totalmente ordenado. Os protocolos de escrita-invalidação impedem que dados antigos sejam lidos, invalidando as cópias quando os itens de dados são atualizados. Eles são mais convenientes para a DSM baseada em paginação, para a qual a escrita-atualização pode ser uma opção dispendiosa.

A granularidade da DSM afeta a probabilidade de disputa entre processos que compartilham itens de dados falsamente por estarem contidos na mesma unidade de compartilhamento (por exemplo, uma página). Ela também afeta o custo por byte da transferência de atualizações entre computadores.

A ultrapaginação pode ocorrer quando é usada escrita-invalidação. Trata-se da transferência repetida de dados entre processos concorrentes, às custas do progresso da aplicação. Isso pode ser reduzido pela sincronização em nível da aplicação, permitindo-se que os computadores mantenham uma página por um tempo mínimo, ou rotulando-se os itens de dados de modo que o acesso de leitura e de escrita sejam sempre garantidos em conjunto.

Este capítulo descreveu os três principais protocolos de escrita-invalidação do Ivy para DSM baseada em paginação, os quais tratam dos problemas do gerenciamento do conjunto de cópia e da localização do proprietário de uma página. Foram eles: o protocolo do gerenciador central, no qual um único processo armazena o endereço do proprietário corrente de cada página, o protocolo que usa *multicast* para localizar o proprietário corrente de uma página e o protocolo do gerenciador distribuído dinâmico, que usa encaminhamento de ponteiros para localizar o proprietário corrente de uma página.

O Munin é um exemplo de implementação da consistência relaxada. Ele implementa consistência relaxada *ávida*, no sentido de que propaga as mensagens de atualização ou invalidação assim que uma trava é liberada. Existem implementações *preguiçosas* alternativas, as quais propagam essas mensagens somente quando elas são exigidas. O Munin permite que os programadores anotem seus itens de dados para selecionar as opções de protocolo mais convenientes, de acordo com a maneira como eles são compartilhados.

Exercícios

18.1 Explique sob quais aspectos a DSM é conveniente ou inconveniente para sistemas cliente-servidor.
página 645

18.2 Discuta se a passagem de mensagens ou a DSM é preferível para aplicações tolerantes a falhas.
página 645–646

18.3 Como você lidaria com o problema das representações de dados diferentes para uma implementação de DSM baseada em *middleware*, em computadores heterogêneos? Como você abordaria o problema em uma implementação baseada em paginação? Sua solução serve para ponteiros?
página 647–648

18.4 Por que devemos implementar DSM baseada em paginação em nível de usuário e o que é exigido para conseguir isso? *página 648–649*

18.5 Como você implementaria um semáforo usando um espaço de tuplas? *página 649–650*

18.6 A memória subjacente na execução dos dois processos a seguir tem consistência seqüencial (supondo que, inicialmente, todas as variáveis são configuradas como zero)?

P_1: $R(x)1$; $R(x)2$; $W(y)1$

P_2: $W(x)1$; $R(y)1$; $W(x)2$ *página 652–653*

18.7 Usando a notação $R()$, $W()$, dê um exemplo de execução em uma memória que seja coerente mas não apresente consistência seqüencial. Uma memória pode ter consistência seqüencial, mas não ser coerente? *página 652–653*

18.8 Na escrita-atualização, mostre que a consistência seqüencial poderia ser violada se cada atualização fosse feita de forma local, antes de enviá-la de forma assíncrona para outros gerenciadores de réplica, mesmo que o envio seja feito por *multicast* totalmente ordenado. Discuta se um *multicast* assíncrono pode ser usado para obter consistência seqüencial. (Dica: considere se vai haver bloqueio das operações subseqüentes.) *página 653–654*

18.9 Uma memória com consistência seqüencial pode ser implementada usando-se um protocolo de escrita-atualização, empregando-se um *multicast* síncrono totalmente ordenado. Discuta quais requisitos de ordenação de *multicast* seriam necessários para implementar memória coerente. *página 653–654*

18.10 Explique por que, sob um protocolo de escrita-atualização, é necessário cuidado para propagar apenas as palavras que foram atualizadas de forma local dentro de um item de dados.

Projete um algoritmo para representar as diferenças entre uma página e uma versão atualizada dela. Discuta o desempenho desse algoritmo. *página 653–654*

18.11 Explique por que a granularidade é uma questão importante nos sistemas de DSM. Compare o problema da granularidade entre sistemas de DSM orientados a objetos e orientados a bytes, tendo em mente suas implementações.

Por que a granularidade é relevante para espaços de tuplas que contêm dados imutáveis?

O que é falso compartilhamento? Ele pode levar a execuções incorretas? *página 655–656*

18.12 Quais são as implicações da DSM para as políticas de substituição de página (isto é, na escolha de qual página vai ser eliminada da memória principal para trazer uma nova página)?*página 656–657*

18.13 Prove que o protocolo de escrita-invalidação do Ivy garante a consistência seqüencial. *página 658*

18.14 No algoritmo do gerenciador distribuído dinâmico do Ivy, quais etapas são necessárias para minimizar o número de pesquisas exigidas para encontrar uma página? *página 660–661*

18.15 Por que a ultrapaginação é uma questão importante nos sistemas de DSM e quais métodos estão disponíveis para tratar dela? *página 663–664*

18.16 Discuta como a condição CR2 da consistência relaxada poderia ser atenuada. Daí, faça distinção entre consistência relaxada ávida e preguiçosa. *página 665–666*

18.17 Um processo sensor grava a temperatura corrente em uma variável t armazenada em uma DSM com consistência relaxada. Periodicamente, um processo monitor lê t. Explique a necessidade de sincronização para propagar as atualizações feitas em t, mesmo que nenhuma seja necessária no nível da aplicação. Quais desses processos precisa efetuar operações de sincronização? *página 665*

18.18 Mostre que o histórico a seguir não tem consistência causal:

P_1: $W(a)0$; $W(a)1$

P_2: $R(a)1$; $W(b)2$

P_3: $R(b)2$; $R(a)0$ *página 668–669*

18.19 Que vantagem uma implementação de DSM pode obter pelo fato de conhecer a associação entre itens de dados e objetos de sincronização? Qual é a desvantagem de tornar a associação explícita? *página 669*

19 Serviços Web

Um serviço web (*web service*) fornece uma interface de serviço que permite aos clientes interagirem com servidores de uma maneira mais geral do que acontece com os navegadores web. Os clientes acessam as operações na interface de um serviço web por meio de requisições e respostas formatadas em XML e, normalmente, transmitidas por HTTP. Os serviços web podem ser acessados de uma maneira mais *ad hoc* do que os serviços baseados em CORBA, permitindo que eles sejam mais facilmente usados em aplicações de Internet.

Assim como no CORBA e em Java, as interfaces dos serviços web podem ser descritas em uma IDL. Mas, para os serviços web, informações adicionais precisam ser descritas, incluindo a codificação e os protocolos de comunicação em uso e o local do serviço.

Os usuários exigem uma maneira segura de criar, armazenar e modificar documentos e trocá-los pela Internet. Os canais seguros TLS não fornecem todos os requisitos necessários. A segurança da XML se destina a suprir essa falta.

Grade (*grid*) é o nome usado para referenciar uma plataforma de *middleware* baseada em serviços web e projetada para uso por grandes grupos dispersos de usuários, com recursos maciços de dados que exigem um processamento substancial. O World-Wide Telescope é uma aplicação típica de grade para colaboração científica na área da astronomia. As características das aplicações científicas com uso intenso de dados são derivadas de um estudo do World-Wide Telescope. Essas características levaram a um conjunto de requisitos para uma arquitetura de grade.

19.1 Introdução

O crescimento da web nos últimos cinco anos (veja a Figura 1.6) prova a eficácia do uso de protocolos simples na Internet, como base para um grande número de serviços e aplicações remotos. Em particular, o protocolo de requisição e resposta HTTP (Seção 4.4) permite que clientes de propósito geral, chamados de navegadores, vejam páginas web e outros recursos com referência aos seus URLs. Veja uma nota, no quadro a seguir, sobre URIs, URLs e URNs.

Entretanto, o uso de um navegador de propósito geral como cliente, mesmo com as melhorias fornecidas por *applets* específicos de uma aplicação, carregados por *download*, restringe a abrangência em potencial dessas aplicações. No modelo cliente-servidor original, tanto o cliente como o servidor eram funcionalmente especializados. Os serviços web (*web services*) retornam a esse modelo, no qual um cliente específico da aplicação interage pela Internet com um serviço que possui uma interface funcionalmente especializada.

Assim, os serviços web fornecem uma infra-estrutura para manter uma forma mais rica e mais estruturada de interoperabilidade entre clientes e servidores. Eles fornecem uma base por meio da qual um programa cliente em uma organização pode interagir com um servidor em outra organização, sem supervisão humana. Em particular, os serviços web permitem o desenvolvimento de aplicações complexas, fornecendo serviços que integram vários outros serviços. Devido à generalidade de suas interações, os serviços web não podem ser acessados diretamente pelos navegadores.

O fornecimento de serviços web como um acréscimo aos servidores web é baseado na capacidade de usar uma requisição HTTP para provocar a execução de um programa. Lembre-se de que, quando um URL em uma requisição HTTP se refere a um programa executável, por exemplo, uma busca, o resultado é produzido por esse programa e retornado. De maneira semelhante, os serviços web são uma extensão da web e podem ser fornecidos por servidores web. Entretanto, seus servidores não precisam ser servidores web. Os termos *servidor web* e *serviços web* não devem ser confundidos: um servidor web fornece um serviço HTTP básico, enquanto um serviço web fornece um serviço baseado nas operações definidas em sua interface.

A representação de dados externa e o empacotamento das mensagens trocadas entre clientes e serviços web são feitos em XML, que foi descrita na Seção 4.3.3. Para recapitular, a XML é uma representação textual que, embora mais volumosa do que as representações alternativas, foi adotada por sua legibilidade e pela conseqüente facilidade de depuração.

O protocolo SOAP (Seção 19.2.1) especifica as regras de uso da XML para empacotar mensagens, por exemplo, para suportar um protocolo de requisição e resposta.

A Figura 19.1 resume os principais pontos a respeito da arquitetura de comunicação onde os serviços web operam: um serviço web é identificado por um URI e pode ser acessado pelos clientes usando mensagens formatadas em XML. O protocolo SOAP é usado para encapsular essas mensagens e transmiti-las por HTTP ou outro protocolo, por exemplo, TCP ou SMTP. Um serviço web distribui descrições de serviço para especificar a interface e outros aspectos do serviço em proveito de clientes em potencial.

A camada superior da figura ilustra o seguinte:

- Os serviços e aplicações web podem ser construídos em cima de outros serviços web.
- Alguns serviços web em particular fornecem a funcionalidade geral exigida para a operação de um grande número de outros serviços web. Eles incluem os serviços de diretório, segurança e coreografia, todos os quais serão discutidos posteriormente neste capítulo.

URI, URL e URN ◊ O URI (*Uniform Resource Identifier*) é um identificador de recurso geral, cujo valor pode ser um URL ou um URN. O URL, que inclui informações de localização do recurso, como o nome de domínio do servidor de um recurso que está sendo nomeado, é bem conhecido de todos os usuários da web. Os URNs (*Uniform Resource Names*) são independentes da localização – eles contam com um serviço de pesquisa para fazer o mapeamento para os URLs dos recursos. Os URNs foram discutidos com mais detalhes na Seção 9.1.

Figura 19.1 Infra-estrutura e componentes dos serviços web.

Geralmente, um serviço web fornece uma *descrição do serviço*, a qual inclui uma definição de interface e outras informações, como o URL do servidor. Isso é usado como base para um entendimento comum entre cliente e servidor quanto ao serviço oferecido. A Seção 19.3 apresentará a WSDL (*Web Services Description Language*).

Outra necessidade comum no *middleware* é um serviço de atribuição de nomes ou de diretório para permitir que os clientes descubram serviços. Os clientes de serviços web têm necessidades semelhantes, mas freqüentemente saem-se bem sem os serviços de diretório. Por exemplo, freqüentemente, eles descobrem serviços a partir das informações presentes em uma página web, por exemplo, como resultado de uma busca no Google. Entretanto, algum trabalho precisa ser feito para fornecer um serviço de diretório que seja conveniente para uso dentro das organizações. Isso será discutido na Seção 19.4.

A segurança da XML será apresentada na Seção 19.5. Nessa estratégia de segurança, documentos ou partes de documentos podem ser assinados ou cifrados. Um documento que possui elementos assinados ou cifrados pode então ser transmitido ou armazenado; posteriormente, podem ser feitas adições e estas também poderão ser assinadas ou cifradas.

Os serviços web dão acesso a recursos de clientes remotos, mas não fornecem uma maneira de coordenar suas operações mútuas. A Seção 19.7 discutirá a coreografia dos serviços web, a qual se destina a permitir que um serviço web utilize padrões de acesso predefinidos no uso de um conjunto de outros serviços web.

A última seção deste capítulo contém um estudo de caso de Grade – uma aplicação baseada em serviços web. A Grade (*grid*) é uma infra-estrutura destinada a dar acesso ao compartilhamento de recursos em larga escala, incluindo programas, arquivos, dados, computadores, sensores e redes. A web foi criada originalmente para atender as necessidades de equipes de físicos em diferentes locais que queriam compartilhar a documentação de suas experiências. Mais recentemente, o conceito de grade foi desenvolvido entre os cientistas, com a necessidade de um ambiente de computação distribuída mais geral na Internet.

19.2 Serviços web

Geralmente, uma interface de serviço web consiste em um conjunto de operações que podem ser usadas por um cliente na Internet. As operações de um serviço web podem ser fornecidas por uma variedade de recursos diferentes, por exemplo, programas, objetos ou bancos de dados. Um serviço web pode ser gerenciado por um servidor web, junto com páginas web, ou pode ser um serviço totalmente separado.

A principal característica da maioria dos serviços web é que eles podem processar mensagens SOAP formatadas em XML (veja a Seção 19.2.1). Uma alternativa é a estratégia REST, que está descrita em linhas gerais a seguir. Cada serviço web usa sua própria descrição para tratar das carac-

terísticas específicas das mensagens que recebe. Para ler uma boa narrativa de muitos aspectos mais detalhados dos serviços web, consulte Newcomer [2002].

Muitos servidores web comerciais conhecidos, incluindo Amazon, Yahoo, Google e eBay, oferecem interfaces de serviço que permitem aos clientes manipular seus recursos web. Como exemplo, o serviço web oferecido pela Amazon.com fornece operações que permitem aos clientes obter informações sobre produtos, adicionar um item em um carrinho de compras ou verificar o status de uma transação. Os serviços web da Amazon [associates.amazon.com] podem ser acessado por SOAP (Seção 19.2.1) ou por REST. Isso permite que aplicações de outros fornecedores construam serviços com valor agregado sobre aqueles fornecidos pela Amazon.com. Por exemplo, uma aplicação de controle de inventário e aquisição poderia pedir o fornecimento de mercadorias da Amazon.com, à medida que elas fossem necessárias, e controlar automaticamente a mudança de status de cada requisição. Mais de 50.000 desenvolvedores se registraram para uso desses serviços web nos dois anos após eles serem introduzidos [Greenfield e Dornan 2004].

Outro exemplo interessante de aplicação que exige a presença de um serviço web é a que implementa *sniping* em leilões da eBay. *Sniping* significa fazer um lance durante os últimos segundos antes que um leilão termine. Embora os seres humanos possam realizar as mesmas ações, por meio da interação direta com a página web, eles não podem fazer isso com tanta rapidez.

Combinação de serviços web ◊ O fornecimento de uma interface de serviço permite que suas operações sejam combinadas com as de outros serviços para fornecer nova funcionalidade. Na verdade, a aplicação de aquisição anterior também poderia estar usando outros fornecedores. Como outro exemplo das vantagens de combinar vários serviços, considere o fato de que atualmente as pessoas utilizam seus navegadores para fazer reservas de passagens aéreas, hotéis e alugar carros, com uma seleção de sites web diferentes. Entretanto, se cada um desses sites web fornecesse uma interface de serviço padrão, então um Serviço de Agente de Viagens poderia usar suas operações para fornecer ao viajante uma combinação desses serviços. Esse ponto está ilustrado na Figura 19.2.

Padrões de comunicação ◊ O serviço de agente de viagens ilustra o possível uso dos dois padrões de comunicação alternativos disponíveis nos serviços web:

- o processamento de uma reserva demora um longo tempo para terminar, e bem poderia ser suportado por uma troca assíncrona de documentos, começando com os detalhes das datas e destinos, seguida do retorno das informações de status de tempos em tempos e, finalmente, dos detalhes da conclusão. O desempenho não é problema neste caso.
- enquanto a verificação dos detalhes do cartão de crédito e as interações com o cliente devem ser fornecidas por um protocolo de requisição e resposta.

Em geral, os serviços web usam um padrão de comunicação de requisição e resposta síncrona com seus clientes, ou se comunicam por meio de mensagens assíncronas. Este último estilo de comunicação pode ser usado mesmo quando as requisições exigem respostas, no caso em que o cliente envia

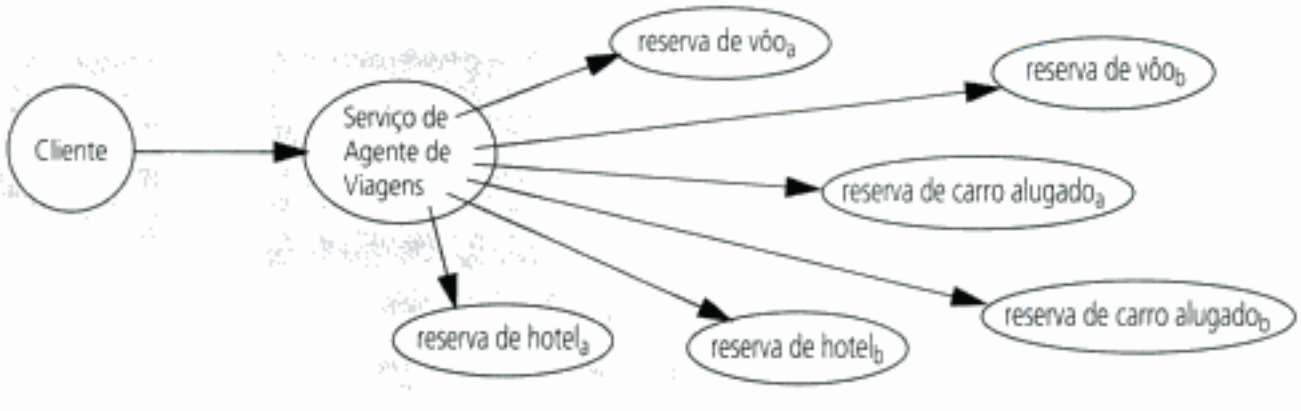


Figura 19.2 O serviço de agente de viagens combina vários serviços web.

Hidden page

Hidden page

Hidden page

- Para comunicação cliente-servidor, o elemento *corpo* contém uma requisição (*Request*) ou uma resposta (*Reply*). Esses dois casos estão ilustrados nas Figuras 19.4 e 19.5.

A Figura 19.4 mostra um exemplo de mensagem de requisição simples, sem cabeçalho. O *corpo* engloba um elemento com o nome do procedimento a ser chamado e o URI do espaço de nomes (o arquivo que contém o esquema XML) para a descrição do serviço relevante, que é denotada por *m*. Os elementos internos de uma mensagem de requisição contêm os argumentos do procedimento. Essa mensagem de requisição fornece dois strings a serem retornados na ordem oposta pelo procedimento que está no servidor. O espaço de nomes XML, denotado por *env*, contém as definições SOAP de um *envelope*. A Figura 19.5 mostra a mensagem de resposta bem-sucedida correspondente, a qual contém os dois argumentos de saída. Note que o nome do procedimento tem *Response* anexado a ele. Se um procedimento tem um valor de retorno, então ele pode ser denotado como um elemento chamado *rpc: result*. Note que a mensagem de resposta usa os mesmos dois esquemas XML da mensagem de requisição, o primeiro definindo o envelope SOAP e o segundo definindo os nomes do procedimento e do argumento específicos da aplicação.

Erros SOAP: se uma requisição falha de alguma maneira, as descrições do erro são transmitidas no corpo de uma mensagem de resposta em um elemento *falha*. Esse elemento contém informações sobre o erro, incluindo um código e um string associado, junto com detalhes específicos da aplicação.

Cabeçalhos SOAP ◊ Os cabeçalhos das mensagens se destinam a ser usados pelos intermediários para adicionar no serviço que trata da mensagem transportada no corpo correspondente. Entretanto, dois aspectos dessa utilização não ficam claros na especificação do SOAP:

Nesta figura e na próxima, cada elemento XML é representado por uma caixa sombreada com seu nome em itálico, seguido dos atributos e seu conteúdo.

Figura 19.4 **Exemplo de uma requisição simples sem cabeçalhos.**

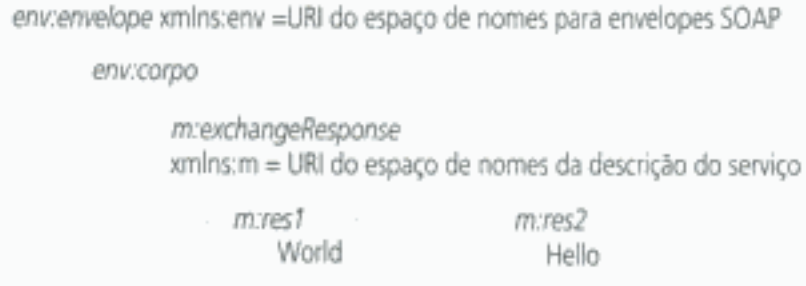


Figura 19.5 **Exemplo de resposta correspondente à requisição da Figura 19.4.**

Hidden page

sições de compra e perguntas, os quais são tratados por diferentes módulos de software. O cabeçalho *Action* permite que o módulo correto seja escolhido sem inspecionar a mensagem SOAP. Esse cabeçalho pode ser usado, caso o tipo de conteúdo HTTP seja especificado como *application/soap+xml*.

A separação da definição do envelope SOAP das informações sobre como e para onde elas devem ser enviadas torna possível usar uma variedade de protocolos subjacentes diferentes. A especificação SOAP informa como o protocolo SMTP pode ser usado como uma maneira alternativa de transmitir documentos codificados como mensagens SOAP.

Mas essa vantagem também é uma desvantagem. Isso implica que o desenvolvedor deve estar envolvido nos detalhes do protocolo de transporte específico escolhido. Além disso, ela torna difícil usar diferentes protocolos para diferentes partes da rota seguida por uma mensagem em particular.

Avanços no endereçamento e no direcionamento do protocolo SOAP ◊ Dois problemas foram mencionados anteriormente:

- como tornar o protocolo SOAP independente do transporte subjacente usado;
- e como especificar uma rota a ser seguida por uma mensagem SOAP por meio de um conjunto de intermediários.

Um trabalho anterior nessa área, chamado WS-Routing, de Nielsen e Thatte [2001], sugere que o endereço do ponto final e a informação de envio devem ser especificados nos cabeçalhos SOAP. Isso separa efetivamente o destino da mensagem do protocolo subjacente. Eles sugeriram especificar o caminho a ser seguido fornecendo o endereço do ponto final e o próximo *hop*. Cada um dos intermediários atualizaria a informação do próximo *hop*.

Um trabalho mais recente, relatado por Box e Curbera *et al.* [2004] e descrito como WS-Addressing, sugere que fazer os intermediários alterarem os cabeçalhos poderia levar a brechas de segurança. Eles propõem uma alternativa onde os cabeçalhos especificam o endereço do ponto final e uma infraestrutura SOAP subjacente fornece a informação do próximo *hop*. Além disso, eles propõem o uso de cabeçalhos SOAP para o endereço de retorno e um identificador de mensagem.

Comunicação confiável ◊ Os serviços web não possuem um protocolo que envie mensagens de modo confiável na presença de falhas. O protocolo normal do SOAP, HTTP, é executado sobre TCP, cujo modelo de falha foi discutido na Seção 4.2.4. Para resumir: o protocolo TCP não garante o envio de mensagens em face de todas as dificuldades – e quando atinge um tempo limite na espera por confirmações, ele declara que a conexão está desfeita, no ponto em que os processos que estão se comunicando ficam sem saber se as mensagens que enviaram recentemente foram recebidas ou não.

Foram feitos alguns trabalhos visando o fornecimento de comunicação confiável de mensagens SOAP com distribuição garantida, sem duplicação e com a ordem das mensagens assegurada. Duas especificações concorrentes foram fornecidas por Ferris e Langworthy [2004] e Evans *et al.* [2003].

Será interessante ver se as medidas de tolerância a falhas estabelecidas, como confirmações, retransmissão de requisições, filtragem de duplicatas e números de seqüência, serão eficazes para o ambiente heterogêneo e de grande escala como a Internet. Em particular, a escolha de períodos de tempo limite (*timeouts*) pode ser um desafio.

Passando por *firewalls* ◊ Os serviços web se destinam a serem usados por clientes de uma organização que acessam servidores de outra pela Internet. A maioria das organizações utiliza um *firewall* para proteger os recursos presentes em suas próprias redes e os protocolos de transporte, como aqueles usados pela RMI Java ou CORBA, normalmente não seriam capazes de atravessar um *firewall*. Entretanto, os *firewalls* normalmente permitem que mensagens HTTP e SMTP passem por eles. Portanto, é conveniente usar um desses protocolos para transportar mensagens SOAP.

19.2.2 Uma comparação de serviços web com o modelo de objeto distribuído

Um serviço web tem uma interface que pode fornecer operações para acessar e atualizar os recursos de dados que gerencia. Em um nível superficial, a interação entre cliente e servidor é muito parecida com a RMI, onde um cliente usa uma referência de objeto remoto para invocar uma operação em um objeto remoto. Para um serviço web, o cliente usa um URI para invocar uma operação no recurso

Hidden page

Hidden page

```
import java.util.Vector;
public class ShapeListImpl implements ShapeList{
    private Vector theList = new Vector();
    private int version = 0;
    private Vector theVersions = new Vector();

    public int newShape(GraphicalObject g) throws RemoteException{
        version++;
        theList.addElement(g);
        theVersions.addElement(new Integer(version));
        return theList.size();
    }
    public int numberOfShapes(){}
    public int getVersion() {}
    public int getGOVersion(int i){ }
    public GraphicalObject getAllState(int i) {}
}
```

Figura 19.8 Implementação em Java do servidor *ShapeList*.

O Tomcat [Apache 2004] é um contêiner de *servlet* comumente usado. Quando o Tomcat está em execução, sua interface de gerenciamento está disponível em um URL para visualização com um navegador. Essa interface mostra os nomes dos *servlets* que estão correntemente distribuídos e fornece operações para gerenciá-los e para acessar informações sobre cada um, incluindo sua descrição de serviço. Uma vez que um *servlet* seja distribuído no Tomcat, os clientes podem acessá-lo e os efeitos combinados de suas operações serão armazenados em suas variáveis de instância. Em nosso exemplo, uma lista de objetos *GraphicalObjects* será construída, à medida que cada um deles for adicionado, como resultado de uma requisição cliente para a operação *newShape*. Se um *servlet* for interrompido pela interface de gerenciamento do Tomcat, os valores das variáveis de instância serão reconfigurados quando ele for reiniciado.

O Tomcat também dá acesso a uma descrição de cada um dos serviços que contém, para permitir que os programadores projetem programas clientes e para facilitar a compilação automática dos *proxies* exigidos pelo código do cliente. A descrição do serviço é legível para seres humanos, pois é expressa em notação XML.

O programa cliente ◊ O programa cliente pode usar *proxies* estáticos, *proxies* dinâmicos ou uma interface de invocação dinâmica. Em todos os casos, a descrição do serviço relevante pode ser usada para obter as informações exigidas pelo código do cliente. Em nosso exemplo, a descrição do serviço pode ser obtida do Tomcat.

Proxies estáticos: A Figura 19.9 mostra o cliente *ShapeList* fazendo uma chamada por meio de um *proxy* – um objeto local que passa mensagens para o serviço remoto. O código do *proxy* é gerado por *wscompile* a partir da descrição do serviço. O nome da classe do *proxy* é formado pela adição de '*_Impl*' ao nome do serviço – neste caso, a classe do *proxy* é chamada de *MyShapeListService_Impl*. Esse nome é específico da implementação, pois a especificação do SOAP não fornece uma regra para atribuição de nomes de classes de *proxy*.

Na linha 1, o método *createProxy* é chamado. Esse método aparece na linha 5, onde vai para a linha 6 para criar um *proxy*, usando a classe *MyShapeListService_Impl*; em seguida, ele retorna o *proxy* (note que às vezes os *proxies* são chamados de *stubs*, daí o nome da classe *Stub*). Na linha 2, o URL do serviço é fornecido para o *proxy* por intermédio do argumento dado na linha de comando. Na linha 3, o tipo do *proxy* é ajustado para estar de acordo com o tipo da interface – *ShapeList*. A linha 4 faz uma chamada para o procedimento remoto *getAllState*, solicitando ao serviço para que retorne o objeto que está no elemento 0 no vetor de *GraphicalObjects*.

Hidden page

SOAP. Um método de esqueleto executa as seguintes tarefas: ele analisa o envelope SOAP presente na mensagem de requisição e extrai seus argumentos, chama o método correspondente e monta um envelope de resposta SOAP contendo os resultados.

Erros, falhas e correção no SOAP/XML: falhas podem ser relatadas pelo módulo HTTP, pelo despachante, pelo esqueleto ou pelo próprio serviço. O serviço pode relatar seus erros por meio de um valor de retorno ou de parâmetros de falha especificados na descrição do serviço. O esqueleto é responsável por verificar se o envelope SOAP contém uma requisição e se o código XML no qual ele está escrito é bem formado. Tendo estabelecido que o código XML é válido, o esqueleto usará o espaço de nomes XML presente no envelope para verificar se a requisição corresponde ao serviço oferecido e se a operação e seus argumentos são apropriados. Se a validação da requisição falhar em um desses níveis, então um erro será retornado para o cliente. Verificações semelhantes são feitas pelo *proxy*, quando ele recebe o envelope SOAP contendo o resultado.

19.2.4 Comparação de serviços web com CORBA

A principal diferença entre serviços web e o CORBA, ou outro *middleware* semelhante, é o contexto no qual eles se destinam a serem usados. O CORBA foi projetado para uso dentro de uma única organização, ou entre um pequeno número de organizações colaboradoras. Isso resultou em certos aspectos do projeto sendo centralizados demais para uso cooperativo por parte de organizações independentes ou para uso *ad hoc* sem arranjos anteriores, conforme será explicado a partir de agora.

Problemas de atribuição de nomes ◊ No CORBA, cada objeto remoto é referenciado por meio de um nome que é gerenciado por uma instância do serviço de atribuição de nomes CORBA (Seção 20.3.1). Esse serviço, assim como o DNS, fornece um mapeamento de um nome para um valor, a ser usado como endereço (um IOR, no CORBA). Mas, ao contrário do DNS, o serviço de atribuição de nomes CORBA é projetado para uso dentro de uma organização, em vez de ser usado por toda a Internet.

No serviço de atribuição de nomes CORBA, cada servidor gerencia um grafo de nomes com um contexto de atribuição de nomes inicial e é inicialmente independente de quaisquer outros servidores. Embora organizações separadas possam confederar (unir) seus serviços de atribuição de nomes, isso não é automático. Antes que um servidor possa se unir com outro, ele precisa conhecer o contexto de atribuição de nomes inicial deste. Assim, o projeto do serviço de atribuição de nomes CORBA restringe efetivamente o compartilhamento de objetos CORBA dentro de um pequeno conjunto de organizações que tenham confederado seus serviços de atribuição de nomes.

Problemas de referência ◊ Agora, consideraremos se uma referência de objeto remoto CORBA, que é chamada de IOR (Seção 20.2.4), poderia ser usada como uma referência de objeto em nível de Internet, da mesma maneira que um URL. Cada IOR contém um *slot* que especifica o identificador de tipo da interface do objeto que referencia. Entretanto, esse identificador de tipo é entendido apenas pelo repositório de interfaces que armazena a definição do tipo correspondente. Isso tem a implicação de que o cliente e o servidor precisam usar o mesmo repositório de interfaces, o que na realidade não é prático em escala global.

Em contraste, no modelo de serviços web, um serviço é identificado por meio de um URL, permitindo que um cliente em qualquer parte da Internet faça uma requisição para um serviço que pode pertencer a qualquer organização de qualquer lugar. Isto é, um serviço web pode ser compartilhado por clientes de toda a Internet. O DNS é o único serviço exigido para acesso de URL – e é projetado para funcionar eficientemente no âmbito da Internet.

Separação de ativação e localização ◊ As tarefas de localizar e ativar serviços web são bem separadas. Em contraste, uma referência persistente do CORBA se refere a um componente da plataforma (o repositório de implementação) que ativa o objeto correspondente sob demanda em qualquer computador conveniente e também é responsável por localizar o objeto quando ele tiver sido ativado.

Facilidade de uso ◊ A infra-estrutura HTTP e XML de serviços web é bem entendida e conveniente para uso e já está instalada em todos os sistemas operacionais comumente usados, embora o usuário

Hidden page

Em contraste, na estratégia do vinculador, o cliente usa um nome para pesquisar a referência do serviço em tempo de execução, possibilitando que as referências de serviço mudem com o passar do tempo. Mas essa estratégia exige um procedimento indireto de um nome para uma referência de serviço para todos os serviços, mesmo que muitos deles possam sempre permanecer no mesmo local.

No contexto dos serviços web, a WSDL (ou *Web Services Description Language*) é comumente usada para descrições de serviço. Quando este livro foi escrito, a WSDL 2.0 [www.w3.org XI] era um esboço funcional do W3C. Ela define um esquema XML para representar os componentes de uma descrição de serviço, os quais incluem, por exemplo, os nomes de elemento *definitions*, *types*, *message*, *interface*, *bindings* e *services*.

A WSDL separa a parte abstrata de uma descrição de serviço da parte concreta, como se vê na Figura 19.10.

A parte abstrata da descrição inclui um conjunto de definições dos tipos usados pelo serviço, em particular os tipos dos valores trocados nas mensagens. O exemplo em Java da Seção 19.2.3, cuja interface Java aparece na Figura 19.7, usa os tipos Java *int* e *GraphicalObject*. O primeiro (assim como qualquer tipo básico) pode ser transformado diretamente no equivalente da XML. Mas *GraphicalObject* é definido na linguagem Java em termos dos tipos *int*, *String* e *boolean*. *GraphicalObject* é representado na XML, para uso comum por clientes heterogêneos, como *complexType*, consistindo em uma seqüência de tipos XML nomeados, incluindo, por exemplo:

```
<element name="isFilled" type="boolean"/>
<element name="originx" type="int"/>
```

O conjunto de nomes definido dentro da seção *types* de uma definição WSDL é chamado de *espaço de nomes de destino*. A seção *message* da parte abstrata contém uma descrição do conjunto de mensagens trocadas. Para o estilo documento de interação, essas mensagens serão usadas diretamente. Para o estilo de interação requisição e resposta, existem duas mensagens para cada operação, as quais são usadas para descrever as operações na seção *interface*. A parte concreta especifica como e onde o serviço pode ser contatado.

A modularidade inerente de uma definição WSDL permite que seus componentes sejam combinados de diferentes maneiras; por exemplo, a mesma interface pode ser usada com diferentes vínculos ou localizações. Os tipos podem ser definidos dentro do elemento *types*, ou em um documento separado referenciado por um URI do elemento *types*. Neste último caso, as definições de tipo podem ser referenciadas a partir de vários documentos WSDL diferentes.

Mensagens ou operações ◊ Nos serviços web, tudo que o cliente e o servidor precisam é ter uma idéia comum sobre a mensagem a ser trocada. Para um serviço baseado na troca de um pequeno número de tipos diferentes de documento, a WSDL descreve apenas os tipos das diferentes mensagens a serem trocadas. Quando um cliente envia uma dessas mensagens para um serviço web, este decide qual operação vai efetuar e qual tipo de mensagem vai enviar de volta para o cliente, de acordo com o tipo da mensagem que recebeu. Em nosso exemplo em Java, duas mensagens serão definidas para cada uma das operações na interface – uma para a requisição e uma para a resposta. Por exemplo, a Figura 19.11 mostra as mensagens de requisição e resposta da operação *newShape*, que tem um único argumento de entrada de tipo *GraphicalObject* e um único argumento de saída de tipo *int*.

Figura 19.10 Os principais elementos em uma descrição WSDL.

Hidden page

Hidden page

- o esquema XML dos formatos de mensagem. O padrão é o envelope SOAP;
- o esquema XML da representação de dados externa. O padrão é a codificação SOAP da XML.

A Figura 19.14 também mostra os detalhes dos vínculos de uma das operações (*newShape*), especificando que a mensagem de *entrada* e a de *saída* devem passar em um corpo SOAP, usando um estilo de codificação em particular e, além disso, que a operação deve ser transmitida como uma *Action* soap.

Serviço: cada elemento *service* em um documento WSDL especifica o nome do serviço e um ou mais *pontos finais* (ou portas) onde uma instância do serviço pode ser contatada. Cada um dos elementos *endpoint* se refere ao nome do vínculo em uso e, no caso de um vínculo SOAP, utiliza um elemento *soap:address* para especificar o URI da localização do serviço.

Documentação ◊ Informações legíveis para seres humanos e pela máquina podem ser inseridas em um elemento *documentation* na maioria dos pontos dentro de um documento WSDL. Essas informações podem ser removidas antes que a WSDL seja usada para processamento automático, por exemplo, por compiladores de *stub*.

Uso de WSDL ◊ Documentos WSDL completos podem ser acessados por meio de seus URIs por clientes e servidores, direta ou indiretamente, por intermédio de um serviço de diretório como o UDDI. Estão disponíveis ferramentas para gerar definições de WSDL a partir das informações fornecidas por meio de uma interface gráfica com o usuário, eliminando a necessidade de envolvimento dos usuários com os detalhes complexos e com a estrutura da WSDL. Por exemplo, a *Web Services Description Language* do *toolkit* Java permite a criação, representação e manipulação de documentos WSDL descrevendo serviços [WSDL4J 2003]. As definições WSDL também podem ser geradas a partir de definições de interface escritas em outras linguagens, como JAX-RPC Java, discutida anteriormente na Seção 19.2.1.

19.4 Um serviço de diretório para uso com serviços web

Existem muitas maneiras pelas quais os clientes podem obter descrições de serviço; por exemplo, qualquer um que forneça um serviço web de mais alto nível, como o serviço de agente de viagens, discutido na Seção 19.1, quase certamente faria uma página web anunciando o serviço e os clientes em potencial se depariariam com a página ao procurar serviços desse tipo.

Entretanto, qualquer organização que pretenda basear suas aplicações em serviços web achará mais conveniente usar um serviço de diretório para tornar esses serviços disponíveis para os clientes. Esse é o objetivo do UDDI (*Universal Directory and Discovery Service*) [Bellwood *et al.* 2003], que fornece um serviço de nome e um serviço de diretório (veja a Seção 9.3). Isto é, as descrições de serviço WSDL podem ser pesquisadas pelo nome (um serviço de catálogo telefônico) ou pelo atributo (um serviço de páginas amarelas). Elas também podem ser acessadas diretamente por meio de seus URLs, o que é conveniente para desenvolvedores que estejam projetando programas clientes que utilizam o serviço.

Os clientes podem usar a estratégia das páginas amarelas para pesquisar uma categoria de serviço em particular, como um agente de viagens ou uma livraria, ou podem usar a estratégia do catálogo telefônico para pesquisar um serviço com referência à organização que o fornece.

Estruturas de dados ◊ As estruturas de dados que suportam UDDI são projetadas de forma a permitir todos os estilos de acesso anteriores e podem incorporar qualquer volume de informações legíveis para seres humanos. Os dados são organizados em termos das quatro estruturas mostradas na Figura 19.15, cada uma das quais podendo ser acessada individualmente, por meio de um identificador chamado de *chave* (além de *tModel*, que pode ser acessado por um URL):

businessEntity: descreve a organização que fornece esses serviços web, dando seu nome, endereço, atividades, etc.;

Hidden page

enviadas ao seu proprietário – isto é, o servidor em que ela foi publicada pela primeira vez. É possível um proprietário transmitir sua posse para outro servidor UDDI no mesmo registro.

Esquema de replicação: os membros de um registro propagam cópias das estruturas de dados uns para os outros, como segue: um servidor que fez alterações notifica os outros servidores no registro, os quais então solicitam as alterações. Uma forma de indicação de tempo vetorial é usada para determinar quais das alterações devem ser propagadas e aplicadas. O esquema é simples, em comparação com outros esquemas de replicação que usam indicações de tempo vetoriais, como Gossip (Seção 15.4.1) ou Coda (Seção 15.4.3), pois:

1. todas as alterações em uma estrutura de dados em particular são feitas no mesmo servidor;
2. as atualizações de um servidor em particular são recebidas em ordem seqüencial pelos outros membros, mas nenhuma ordenação em particular é imposta entre as operações de atualizações executadas por diferentes servidores.

Interação entre servidores: conforme descrito anteriormente, os servidores interagem uns com os outros para transmitir o esquema de replicação. Eles também podem interagir para transferir a posse de estruturas de dados. Entretanto, a resposta a uma operação de pesquisa é dada por um único servidor, sem nenhuma interação com outros servidores no registro, ao contrário do serviço de diretório X.500 (Seção 9.5), no qual os dados são particionados entre os servidores, que cooperam uns com os outros na busca do servidor relevante para uma requisição em particular.

19.5 Aspectos de segurança da XML

A segurança da XML consiste em um conjunto de projetos relacionados do W3C para assinatura, gerenciamento de chaves e criptografia. Ela se destina a ser usada no trabalho cooperativo pela Internet, envolvendo documentos cujo conteúdo talvez precise ser autenticado ou cifrado. Normalmente, os documentos são criados, trocados, armazenados e depois novamente trocados, possivelmente após serem modificados por uma série de usuários diferentes.

A WS-Security [Kaler 2002] é outra estratégia de segurança relativa à aplicação de integridade de mensagem, confidencialidade da mensagem e autenticação de mensagem no SOAP.

Como exemplo de um contexto no qual a segurança da XML seria útil, considere um documento contendo os registros médicos de um paciente. Diferentes partes desse documento são usadas no consultório médico local e em várias clínicas e hospitais especializados visitados pelo paciente. Ele será atualizado por médicos, especialistas e enfermeiras que estejam tomando notas sobre condições e tratamento, por administradores que estejam marcando compromissos e por farmacêuticos para fornecer remédios. Diferentes partes do documento serão visíveis pelas diferentes funções mencionadas anteriormente e possivelmente também pelo paciente. É fundamental que certas partes do documento, por exemplo, as recomendações quanto ao tratamento, possam ser atribuídas à pessoa que as fez e possa haver garantias de que não tenham sido alteradas.

Essas necessidades não podem ser atendidas pelo TLS (anteriormente conhecido como SSL, Seção 7.6.3), usado para criar um canal seguro para a comunicação de informações. Ele permite que os processos nos dois extremos do canal negociem a necessidade de autenticação, de criptografia e as chaves e os algoritmos a serem usados, tanto quando um canal for estabelecido, quanto durante seu tempo de vida. Por exemplo, os dados sobre uma transação financeira poderiam ser assinados e enviados à vontade, até que informações sigilosas, como detalhes do cartão de crédito, fossem fornecidas, no ponto em que a criptografia seria aplicada.

Para possibilitar o novo tipo de utilização mencionado anteriormente, a segurança deve ser especificada e aplicada no próprio documento, em vez de ser uma propriedade do canal que o transmitirá de um usuário para outro.

Isso é possível na XML, ou em outros formatos de documento estruturados, nos quais podem ser usados metadados. Marcas (*tags*) XML podem ser usadas para definir as propriedades dos dados no documento. Em particular, a segurança da XML depende de novas *tags* que possam ser usadas para indicar o início e o fim de seções de dados cifrados ou assinados e de assinaturas. Uma vez que

a segurança necessária tenha sido aplicada dentro de um documento, ele pode ser enviado para uma variedade de usuários diferentes, até por meio de *multicast*.

Requisitos básicos ◊ A segurança da XML deve fornecer pelo menos o mesmo nível de proteção do TLS. Isto é:

Ser capaz de cifrar um documento inteiro ou apenas algumas partes selecionadas dele: por exemplo, considere as informações sobre uma transação financeira, as quais incluem o nome de uma pessoa, o tipo de transação e os detalhes sobre o cartão de crédito ou débito que está sendo usado. Em um caso, apenas os detalhes do cartão poderiam ser ocultos, tornando possível identificar a transação antes de decifrar o registro. No outro caso, o tipo de transação também poderia ser oculto, para que intrusos não pudessem identificar, por exemplo, se é um pedido ou um pagamento.

Ser capaz de assinar um documento inteiro ou apenas algumas partes selecionadas dele: quando um documento se destina a ser usado para trabalho cooperativo por um grupo de pessoas, podem existir algumas partes críticas do documento que devem ser assinadas para garantir que foram feitas por uma pessoa em particular ou que não foram alteradas. Mas também é útil poder ter outras partes que possam ser alteradas durante o uso do documento – essas não devem ser assinadas.

Requisitos básicos adicionais ◊ Mais requisitos surgem da necessidade de armazenar documentos, possivelmente para modificá-los e enviá-los para uma variedade de destinatários diferentes:

Para fazer adições em um documento que já está assinado e para assinar o resultado: por exemplo, Alice pode assinar um documento e passá-lo para Bob, que "atesta a assinatura dela" adicionando uma observação nesse sentido e depois assina o documento inteiro.

Para fazer adições em um documento que já contém seções cifradas e cifrar parte da nova versão, possivelmente incluindo algumas das seções já cifradas.

Para autorizar vários usuários diferentes a ver partes distintas de um documento: no caso de um registro médico, um pesquisador pode ver alguma seção em particular dos dados médicos, um administrador pode ver detalhes pessoais e um médico pode ver ambos.

A flexibilidade e os recursos de estruturação da notação XML tornam possível fazer tudo o que foi descrito anteriormente, sem quaisquer adições no esquema derivado dos requisitos básicos.

Requisitos relativos aos algoritmos ◊ Os documentos XML seguros são assinados e/ou cifrados bem antes de qualquer consideração com relação a quem os acessará. Se o criador não estiver mais envolvido, não será possível negociar os protocolos e o uso de autenticação ou criptografia. Portanto:

O padrão deve especificar um conjunto de algoritmos a serem fornecidos em qualquer implementação de segurança da XML: pelo menos um algoritmo de criptografia e um de assinatura devem ser obrigatórios para permitir a maior interoperabilidade possível. Outros algoritmos opcionais devem ser fornecidos para uso dentro de grupos menores.

Os algoritmos usados para criptografia e autenticação de um documento em particular devem ser selecionados a partir desse conjunto e os nomes dos algoritmos em uso devem ser referenciados dentro do próprio documento XML: se os lugares onde o documento será usado não podem ser previstos, então um dos protocolos exigidos deve ser usado.

A segurança da XML define os nomes dos elementos que podem ser usados para especificar o URI do algoritmo em uso para assinatura ou criptografia. Portanto, para ser capaz de selecionar uma variedade de algoritmos dentro do mesmo documento XML, geralmente um elemento especificando um algoritmo é aninhado dentro de um elemento que contém informações assinadas ou cifradas.

Requisitos para localização de chaves ◊ Quando um documento é criado, e sempre que ele é atualizado, as chaves apropriadas devem ser escolhidas, sem qualquer negociação com as partes que poderão acessar o documento no futuro. Isso leva aos seguintes requisitos:

Ajudar os usuários de documentos seguros na localização das chaves necessárias: por exemplo, um documento que inclui dados assinados deve conter informações quanto a chave pública a ser

Hidden page

Hidden page

19.6 Coordenação de serviços web

A infra-estrutura SOAP suporta interações requisição-resposta entre clientes e serviços web. Entretanto, muitas aplicações úteis envolvem várias requisições que precisam ser executadas em uma ordem em particular. Por exemplo, ao se fazer reservas para um vôo, são reunidas as informações sobre preço e disponibilidade, antes que as reservas sejam feitas. Quando um usuário interage com páginas web por intermédio de um navegador, por exemplo, para fazer reserva em um vôo, ou para dar um lance em um leilão, a interface fornecida pelo navegador, que é baseada nas informações fornecidas pelo servidor, controla a seqüência em que as operações são executadas.

Entretanto, se for um serviço web que estiver fazendo reservas, como o serviço de agente de viagens mostrado na Figura 19.2, então esse serviço precisará trabalhar a partir de uma descrição da maneira apropriada para prosseguir ao interagir com outros serviços que realizam, por exemplo, aluguel de carros e reservas em hotel, assim como reservas em vôos. A Figura 19.18 mostra um exemplo de tal descrição.

Esses exemplos ilustram a necessidade de serviços web, como clientes, para receberem uma descrição de um protocolo em particular a ser seguido ao interagir com outros serviços web. Mas também existe o problema da manutenção da consistência nos dados do servidor quando ele está recebendo e respondendo as requisições de vários clientes. Os Capítulos 13 e 14 discutiram as transações, ilustrando os problemas por meio de uma série de transações bancárias. Como um exemplo simples, em uma transferência de dinheiro entre duas contas bancárias, a consistência exige que o depósito em uma conta e o saque da outra devam ser realizados. O Capítulo 14 apresentou o protocolo de efetivação em duas fases, que é usado por servidores em cooperação para garantir a consistência de transações.

Em alguns casos, transações atômicas são adequadas aos requisitos de aplicações que usam serviços web. Entretanto, atividades como aquelas do agente de viagens demoram um longo tempo para terminar e seria impraticável usar um protocolo de efetivação em duas fases para executá-las, pois envolve manter recursos bloqueados por longos períodos de tempo. Uma alternativa é usar um protocolo mais relaxado, no qual cada participante faz alterações no estado persistente, quando elas ocorrem. No caso de falha, um protocolo em nível de aplicação é usado para desfazer essas ações.

No *middleware* convencional, a infra-estrutura fornece um protocolo de requisição e resposta simples, deixando outros serviços, como transações, persistência e segurança, serem implementados

1. O cliente solicita ao serviço de agente de viagens informações sobre um conjunto de serviços; por exemplo, vôos, aluguel de carro e reservas em hotel.
2. O serviço de agente de viagens reúne as informações sobre preço e disponibilidade e as envia para o cliente, o qual escolhe uma das opções a seguir em nome do usuário:
   (a) refinar a consulta, possivelmente envolvendo mais provedores para obter mais informações; e, em seguida, repetir o passo 2;
   (b) fazer reservas;
   (c) sair.
3. O cliente solicita uma reserva e o serviço de agente de viagens verifica a disponibilidade.
4. *Ou* todos estão disponíveis;
   ou para os serviços que não estão disponíveis;
   ou são oferecidas alternativas para o cliente, que volta para o passo 3;
   ou o cliente volta para o passo 1.
5. Faz o depósito.
6. Fornece ao cliente um número de reserva como confirmação.
7. Durante o período decorrido até o pagamento final, o cliente pode modificar ou cancelar as reservas.

Figura 19.18 Cenário do agente de viagens.

Hidden page

Um modelo baseado nesses requisitos está descrito em outro documento de minuta de trabalho do W3C [www.w3.org XVI].

Linguagens de coreografia ◊ A intenção é produzir uma linguagem declarativa, baseada na XML, para definir coreografias que possam usar definições WSDL. Os primeiros estágios do trabalho sobre a *Choreography Definition Language* estão relatados em [www.w3.org XVII]. Antes disso, um grupo de empresas enviou para o W3C uma especificação para a interface de coreografia de serviços web [www.w3.org XVIII].

19.7 Estudo de caso: a Grade

O termo grade (*grid*) é usado para se referir ao *middleware* que é projetado para permitir o compartilhamento de recursos como arquivos, computadores, software, dados e sensores em uma escala muito grande. Normalmente, os recursos são compartilhados por grupos de usuários em diferentes organizações, os quais estão colaborando na solução de problemas que exigem grandes números de computadores para resolvê-los, ou pelo compartilhamento de dados ou pelo compartilhamento de poder de computação. Esses recursos são necessariamente suportados por hardware, sistemas operacionais, linguagens de programação e aplicações heterogêneas. É necessário gerenciamento para coordenar o uso de recursos para garantir que os clientes obtenham o que precisam e que os serviços possam fornecê-los. Em alguns casos, são exigidas técnicas de segurança sofisticadas para garantir o uso correto dos recursos nesse tipo de ambiente.

A Seção 19.7.1 apresentará o World-Wide Telescope, uma aplicação de uso intensivo de dados que é um exemplo do tipo de solução de problema para o qual a grade é projetada. Ele ilustra um padrão de compartilhamento típico e a distribuição geográfica de usuários, a partir da qual podem ser derivadas as características de uma família de aplicações científicas, o que sugere um conjunto de requisitos para uma grade (veja a Seção 19.7.2). Usamos esses requisitos para motivar a arquitetura, a qual especifica uma grade que funciona em serviços web (veja a Seção 19.7.3). A última seção apresenta o *toolkit* Globus, que é uma implementação da arquitetura de grade.

19.7.1 O World-Wide Telescope – uma aplicação de grade

Esse projeto está relacionado à distribuição de recursos de dados compartilhados pela comunidade de astronomia. Ele está descrito no trabalho de Szalay e Gray [2004], Szalay e Gray [2001] e Gray e Szalay [2002]. Os dados astronômicos consistem em repositórios de arquivo de observações, cada um dos quais abrangendo um período de tempo em particular, uma parte do espectro eletromagnético (ótico, raios X, rádio) e uma área em particular do céu. Essas observações são feitas por diferentes instrumentos instalados em vários lugares do mundo.

Um estudo sobre como os astrônomos compartilham seus dados é útil para extrair as características de uma aplicação típica de grade, pois eles compartilham livremente seus resultados uns com os outros e os problemas de segurança podem ser omitidos, tornando esta discussão mais simples.

Os astrônomos fazem estudos que precisam combinar dados sobre os mesmos objetos celestes, mas que envolvem vários períodos de tempo diferentes e diversas partes do espectro. A capacidade de usar observações de dados independentes é importante para a pesquisa. A visualização permite aos astrônomos verem os dados como mapas de dispersão bi ou tridimensionais.

As equipes que reúnem os dados os armazenam em imensos repositórios de arquivo (atualmente, na ordem de terabytes), os quais são gerenciados de forma local por cada equipe. Os instrumentos usados na coleta de dados estão sujeitos à lei de Moore. Por isso, o volume dos dados reunidos cresce exponencialmente. À medida que são reunidos, os dados brutos são analisados e processados em uma seqüência de passos e armazenados como dados derivados para uso pelos astrônomos de todo o mundo. Mas antes que os dados possam ser usados por outros pesquisadores, os cientistas que trabalham em um campo em particular precisam concordar com uma maneira comum de rotular seus dados.

Szalay e Gray [2004] mencionam que, no passado, os dados de pesquisa científica eram incluídos pelos autores em artigos e publicados em periódicos que ficavam em bibliotecas. Mas, hoje em dia,

Hidden page

R2: processamento de dados no local onde eles são armazenados e gerenciados, ou quando são reunidos ou em resposta a uma requisição. Uma consulta típica poderia resultar em uma visualização baseada nos dados reunidos para uma região do céu, gravados por diferentes instrumentos, em diferentes momentos. Isso envolverá selecionar um pequeno volume de dados de cada repositório de arquivo de dados maciço.

R3: O gerenciador de recursos de um repositório de arquivo de dados deve ser capaz de criar instâncias de serviço dinamicamente para tratar com a seção de dados em particular exigida, exatamente como no modelo de objeto distribuído, onde serventes são criados, quando necessário, para manipular diferentes recursos gerenciados por um serviço.

R4: Metadados para descrever:

- características dos dados em um repositório de arquivo, por exemplo, para astronomia: a área do céu, a data e a hora da coleta e os instrumentos utilizados;
- as características de um serviço que esteja gerenciando esses dados, por exemplo, seu custo, sua localização geográfica, seu gerador (anunciante) ou sua carga ou espaço disponível.

R5: Serviços de diretório baseados nos metadados acima.

R6: Software para gerenciar consultas, transferências de dados e reserva antecipada de recursos, levando em conta que os recursos geralmente são gerenciados pelos projetos que geram os dados e que o acesso a eles talvez precise ser racionado.

Os serviços web podem tratar dos dois primeiros requisitos, fornecendo uma maneira conveniente de permitir que os cientistas acessem operações sobre dados em repositórios de arquivo remotos. Isso exigirá que cada aplicação em particular forneça uma descrição do serviço que inclua um conjunto de métodos para acessar seus dados. O *middleware* de grade trata dos requisitos restantes.

Embora o World-Wide Telescope seja uma aplicação típica de uso intensivo de dados, as Grades também são usadas para aplicações de uso intensivo de poder computacional, como a análise de imagens e outros exemplos discutidos na Seção 19.7.4. Onde aplicações de uso intensivo de poder computacional são distribuídas em uma Grade, o gerenciamento de recursos se preocupará com a alocação de recursos de computação e com a harmonização das cargas.

Finalmente, será necessária segurança para muitas aplicações de grade. Por exemplo, a Grade está em uso para pesquisa médica e para aplicações comerciais. Mesmo quando a privacidade dos dados não é problema, será importante estabelecer a identidade das pessoas que criaram os dados.

19.7.3 Arquitetura aberta de serviços de grade

A arquitetura aberta de serviços de grade (OGSA – *Open Grid Services Architecture*) é um padrão para aplicações baseadas em grade [Foster *et al.* 2002 e 2001]. Ela fornece uma estrutura onde os requisitos anteriores podem ser satisfeitos. Ela é baseada em serviços web. Os recursos são gerenciados por serviços de grade específicos da aplicação. O *toolkit* Globus, que implementa a arquitetura, será discutido na Seção 19.7.5.

A Figura 19.19 mostra os principais componentes da arquitetura de grade. Ela ilustra dois aspectos importantes dos *serviços de grade em nível de aplicação*:

1. Eles são serviços web que implementam interfaces de serviço de grade padrão, além de suas interfaces específicas da aplicação. Especificamente, eles implementam as seguintes interfaces de serviço de grade e funcionalidade adicional:
   - uma interface para um conjunto de dados (chamados de *dados de serviço*) que contém metadados sobre o serviço. Os metadados incluem, por exemplo, a hora a ser terminada, mas pode incluir qualquer um dos itens mencionados no requisito R4 anterior, assim como valores da aplicação, como conjunto de resultados recentes ou valores médios;
   - o contexto no qual um serviço é executado deve fornecer uma *fábrica* com a capacidade de criar novas instâncias do serviço e de interrompê-las quando seu tempo acabar, conforme

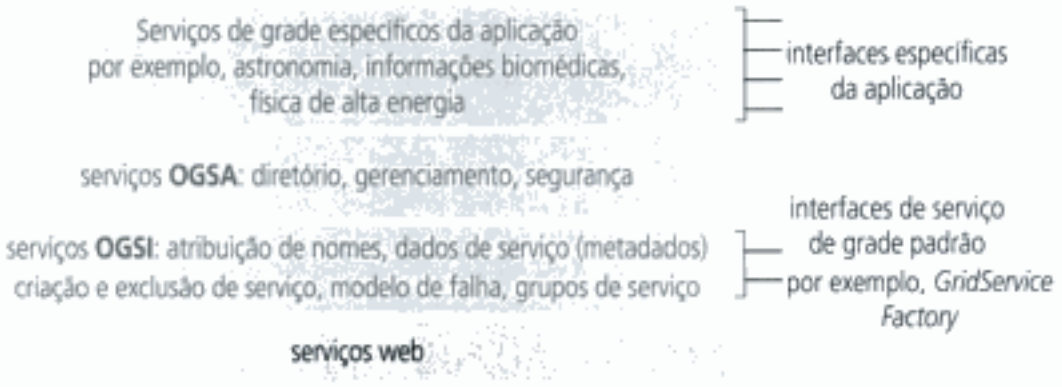


Figura 19.19 Arquitetura aberta de serviços de grade (OGSA).

sugerido pelo requisito R3. A fábrica conta com os recursos de atribuição de nomes da camada OGSI para gerenciar os nomes das instâncias de serviço que cria.

2. Eles utilizam os serviços de grade padrão que são fornecidos como duas camadas separadas, chamadas OGSI e OGSA, acima dos serviços web. A camada de serviços OGSA inclui:
   - serviço de diretório – permitindo que software cliente selecione instâncias de serviço convenientes para suas necessidades, com base nos metadados reunidos das instâncias de serviço que se registraram no serviço de diretório. Isso trata do requisito R5;
   - serviço de gerenciamento – para monitorar serviços a partir das informações presentes em seus dados de serviço, para tratar com falhas; e para controlar os tempos de vida de instâncias de serviço, configurando valores nos dados de serviço. Isso trata do requisito R6;
   - serviços de segurança – fornecendo *logins* únicos e delegação, assim como autenticação e criptografia de dados.

A camada da infra-estrutura aberta de serviços de grade (OGSI – *Open Grid Services Infrastructure*) inclui o seguinte:

- a implementação de um esquema para a atribuição de nomes de instâncias de serviço;
- a definição de elementos de *dados de serviço* padrão que devem ser implementados por cada instância de serviço de grade no nível da aplicação, junto com operações para configurar e obter seus valores. Esses elementos incluem, por exemplo, os nomes das interfaces suportadas, uma referência à fábrica que criou a instância ou o tempo de término;
- definições da interface para a fábrica, para criar novas instâncias de serviço, e operações para configurar os tempos de término ou para destruí-las;
- um modelo de falha para uso por todos os serviços de grade;
- serviços de notificação – para permitir que os serviços se tornem anunciantes de informações sobre dados de serviço e que outros serviços se tornem assinantes;
- grupos de serviço – operações para adicionar e remover membros, para uso de grupos em cooperação que estejam fornecendo um serviço.

As definições e serviços anteriores são fornecidos nas interfaces *GridService*, *Factory* e em outras interfaces da camada OGSI.

Infra-estrutura aberta de serviços de grade ◊ Agora, apresentaremos o esquema de atribuição de nomes de dois níveis que relaciona os nomes de alto nível das instâncias de serviço com nomes de baixo nível e depois apresentaremos mais detalhes dos outros serviços na camada OGSI.

Esquema de atribuição de nomes em dois níveis: os serviços de grade podem ser criados dinamicamente. Portanto, para atender a necessidade de distinguir uma instância de outra, a infra-estrutura inclui um esquema de atribuição de nomes que fornece a cada instância um identificador de longa

duração e globalmente exclusivo, chamado de identificador de serviço de grade (GSH – *Grid Service Handle*). Quando uma instância de um serviço é reiniciada com o mesmo estado de antes, o mesmo GSH pode ser usado.

Um GSH é representado por um URI, o qual pode ser mapeado em um nome de vida curta, chamado de referência de serviço de grade (GSR – *Grid Service Reference*), o qual é usado para se referir ao destino de uma invocação. A infra-estrutura inclui um serviço de transformação de identificador para realizar esse mapeamento. A GSR é um nome cuja estrutura depende do mecanismo de requisição e resposta em uso, por exemplo, no caso de SOAP, a GSR se referirá a um documento WSDL contendo elementos <*service*> e <*binding*>. A GSR se torna inválida quando a instância do serviço a que ela se refere falha.

Se um URN for usado para o GSH, ele poderá ser transformado em diferentes instâncias de serviço em diferentes momentos, permitindo que este último seja migrado ou mesmo replicado em diferentes recursos.

Dados de serviço (metadados): a idéia é que os clientes possam pedir a uma instância de um serviço para que retorne informações sobre seu estado corrente, por exemplo, sua capacidade, espaço livre, carga ou erros correntes, ou mesmo seus resultados recentes. Isso exige que cada instância de um serviço de grade em nível de aplicação implemente armazenamento para dados de serviço e um conjunto de operações para acessá-los.

Um conjunto de operações padrão é fornecido na interface *GridService* – isso inclui operações para acessar os nomes das interfaces suportadas pelo serviço, os nomes dos elementos de dados do serviço, a identidade da fábrica que os criou, seu GSH e sua GSR e seu tempo de término. Todo serviço de grade deve suportar a interface *GridService*, mas também pode incluir interfaces OGSI para notificação ou para grupos de serviço.

Criação e exclusão de serviço: esta parte da infra-estrutura define uma interface *Factory* padrão que especifica operações para criar instâncias de serviço transientes sob demanda, com nomes novos na forma de um par GSH e GSR. A interface *Factory* deve ser implementada por qualquer contêiner para executar serviços de grade em nível de aplicação.

Uma requisição típica para criar uma nova instância de serviço é feita por uma aplicação que especifica como argumentos sua descrição do serviço e o final de seu tempo de vida. Em resposta, a fábrica cria uma nova instância de serviço e a registra em um serviço de transformação de identificador, retornando finalmente o GSH e a GSR corrente para o cliente.

Uma instância de serviço terminará quando sua tarefa estiver concluída ou a pedido do cliente que a criou. Cada instância de serviço recebe um tempo de vida para impedir que sobreviva para sempre, no caso em que mensagens são perdidas. Mas o criador pode solicitar prolongamentos do tempo de vida, enviando mensagens de "manter vivo" (*keep alive*). No caso de falha do cliente, não haverá mais mensagens *keep alive* e a instância do serviço finalmente terminará. Para fornecer autonomia ao serviço, uma instância pode decidir alterar seu próprio tempo de vida. Todos os recursos usados por uma instância de serviço são recuperados quando seu tempo de vida expira. As instâncias de serviço podem ser compartilhadas pelos clientes.

Modelo de falha: a infra-estrutura define uma estratégia comum para o relato de falhas, a qual é usada por todos os serviços em nível de OGSI e é recomendada para uso pelos serviços de grade em nível de aplicação. Ela é definida como um esquema XML que exige pelo menos dois elementos para especificar o serviço de origem e uma indicação de tempo. Ela oferece elementos opcionais para descrever uma falha em linguagem direta, sua causa e um código de falha. Existe uma abrangência para estender o relatório de falha de forma a incluir informações específicas para um serviço OGSI ou específicas da aplicação.

Extensões da interface WSDL: como a WSDL padrão não inclui mecanismos para definir dados de serviço, o OGSI fornece uma extensão para ela, para associar definições dos nomes e tipos de elementos de dados de serviço a uma interface em particular em uma descrição de serviço. Os elementos de dados poderiam, por exemplo, conter sua capacidade, o espaço livre, a carga ou indicações de erro correntes.

Serviços OGSA ◊ Os serviços de nível mais alto são construídos sobre os serviços OGSI. De acordo com Foster *et al.* [2004], diferentes desenvolvedores fornecerão uma variedade de serviços OGSA diferentes, para satisfazer os requisitos específicos de serviços em nível de aplicação. Eles listam os componentes OGSA que são aplicáveis de maneira suficientemente ampla para serem incluídos em qualquer sistema de grade; por exemplo, serviços de diretório, gerenciamento e monitoramento, e segurança. A disponibilidade de interfaces padrão para esses serviços é uma base para garantir que diferentes implementações possam trabalhar umas com as outras. Deixamos a discussão sobre serviços de diretório e segurança para a Seção 19.7.5, na qual os apresentaremos no contexto do *toolkit* Globus.

Serviços de gerenciamento: o gerenciamento na grade se preocupa com qualquer tipo de recurso que possa ser compartilhado ou explorado. Isso significa gerenciamento de serviços, o qual se preocupa com sua organização para uso e com o monitoramento de seu estado. Os problemas tratados incluem envio de tarefa, acordo para fornecer alguma qualidade de serviço em particular e reserva antecipada de recursos. Isso pode ser fornecido com a ajuda de *acordos do nível de serviço* (SLAs – *Service-Level Agreements*) [Hauch e Reiser 2000], os quais dão ao cliente de um recurso a garantia quanto ao tipo de serviço que está sendo fornecido e permitem ao proprietário do recurso manter o controle sobre como ele é usado e sobre o volume de informação exposta para o cliente.

São propostos três tipos diferentes de SLA: em nível de tarefa, que é usado para acordo sobre o desempenho de uma atividade; em nível de recurso, que é usado para acordo sobre o direito de consumir um recurso, por exemplo, por reserva antecipada; e vínculo, que é usado para acordo sobre o uso de um recurso em uma tarefa.

19.7.4 Alguns exemplos de aplicações de grade

A Figura 19.20 mostra alguns exemplos de aplicações que estão usando tecnologia de grade. Os três primeiros têm características semelhantes ao World-Wide Telescope, pois os dados são reunidos por instrumentos científicos e armazenados no local onde são coletados. No exemplo 1, dados de vibração são coletados por intermédio de um sensor em um motor de avião. No exemplo 2, os dados são coletados de estruturas de teste que estão sendo sujeitas a um abalo violento para simular um terremoto. No exemplo 3, instrumentos de ressonância e tomografia são usados para coletar imagens do cérebro. Em todos os casos, o volume de dados brutos aumenta com o passar do tempo, à medida que mais estudos são feitos. Além disso, o volume de dados processados aumenta à medida que as análises são realizadas.

Nesses três exemplos, os dados são gerenciados por equipes de cientistas ou engenheiros pertencentes a organizações separadas. As análises dos dados brutos e as simulações são feitas de forma local e estão disponíveis para colegas de todo o mundo.

Esses exemplos confirmam que o World-Wide Telescope é uma aplicação típica de uso intensivo de dados. Entretanto, as aplicações de grade também são usadas para aplicações com uso intensivo de poder computacional, como:

- O exemplo 4 se refere ao novo detector do CERN que se tornará operacional em 2007. Antes disso, equipes de físicos estão realizando simulações dos resultados esperados do detector. Essa é uma tarefa de uso intensivo de poder computacional, que está sendo realizada por vários computadores em cooperação.
- O exemplo 5 é sobre filtragem virtual – o teste de um banco de dados de milhões de moléculas de droga para ver se elas bloqueiam a atividade de cada uma de um grande número de proteínas em potencial. É usado o poder de computação de um conjunto de computadores desktop executando software de grade. A tarefa é executada em paralelo em cada um dos computadores que está testando o efeito de uma molécula de droga específica sobre uma proteína em particular. Ela usa a capacidade ociosa de computadores *desktop*.
- O exemplo 6 é sobre análise de imagens. A empresa GlobeExplorer fornece imagens de satélite e fotografias aéreas de boa qualidade, extraídas de um repositório de arquivo de milhares de imagens brutas, cada uma das quais exigindo melhoramento. Ela usa a capacidade ociosa de um agrupamento de servidores web.

Hidden page

Hidden page

Relação futura com serviços web ◊ Tem havido algumas discussões sobre a possibilidade de integrar recursos OGSI nos serviços web – sob o título de estrutura WS-Resource [Globus 2004]. A Globus Alliance sugere que a capacidade de criar instâncias de serviço e fazer uso de estado de serviço poderia ser usada mais amplamente do que no contexto de aplicações de grade. Entretanto, sua estratégia de objeto distribuído pode não ser bem-sucedida para muitos tipos de aplicações de baixo acoplamento normalmente executadas como serviços web.

19.8 Resumo

Neste capítulo, mostramos que os serviços web surgiram da necessidade de fornecer uma infra-estrutura para suportar interligação em rede entre diferentes organizações. Essa infra-estrutura geralmente usa o conhecido protocolo HTTP para transportar mensagens entre clientes e servidores pela Internet e é baseada no uso de URIs para se referir aos recursos. A XML, um formato textual, é usada para representação e empacotamento de dados.

Duas influências separadas levaram à aparição de serviços web. Uma delas é a adição de interfaces de serviço em servidores web com o objetivo de permitir que os recursos de um site fossem acessados por programas clientes que não os navegadores, além de usar uma forma de interação mais rica. A outra é o desejo de fornecer algo como RPC na Internet, com base nos protocolos existentes. Os serviços web resultantes fornecem interfaces com conjuntos de operações que podem ser chamadas de forma remota. Assim como qualquer outra forma de serviço, um serviço web pode ser cliente de outro serviço web, permitindo assim que um serviço web integre ou combine com um conjunto de outros serviços web.

SOAP é o protocolo de comunicação geralmente usado pelos serviços web e seus clientes. Ele pode ser usado para transmitir mensagens de requisição e suas respostas entre cliente e servidor, ou pela troca assíncrona de documentos ou por uma forma de protocolo de requisição e resposta baseado em um par de trocas de mensagens assíncronas. Nos dois casos, a mensagem de requisição ou resposta é incluída em um documento formatado em XML, chamado de envelope. Geralmente, o envelope SOAP é transmitido por meio do protocolo HTTP síncrono, embora outros transportes possam ser usados.

Processadores de XML e SOAP estão disponíveis para todas as linguagens de programação e sistemas operacionais amplamente usados. Isso permite que os serviços web e seus clientes sejam implantados em quase qualquer lugar. Essa forma de interligação em rede é possível pelo fato de que os serviços web não estão ligados a nenhuma linguagem de programação em particular nem suportam o modelo de objeto distribuído.

Nos *middlewares* convencionais as definições de interface fornecem aos clientes os detalhes dos serviços. Entretanto, no caso de serviços web, são usadas descrições de serviço. Uma descrição de serviço especifica o protocolo de comunicação a ser usado (por exemplo, SOAP) e o URI do serviço, assim como descreve sua interface. A interface pode ser descrita como um conjunto de operações, ou como um conjunto de mensagens, a serem trocadas entre cliente e servidor.

A segurança da XML foi projetada para fornecer a proteção necessária para o conteúdo de um documento trocado por membros de um grupo de pessoas, as quais têm tarefas diferentes para realizar nesse documento. Diferentes partes do documento estarão disponíveis para diferentes pessoas, algumas com a capacidade de adicionar ou alterar o conteúdo e outras de lê-lo. Para permitir uma flexibilidade completa em seu uso futuro, as propriedades de segurança são definidas dentro do próprio documento. Isso é obtido por meio da XML, que tem um formato auto-descritivo. Elementos da XML são usados para especificar as partes do documento que são cifradas ou assinadas, assim como os detalhes dos algoritmos usados e informações para ajudar a encontrar chaves.

O *middleware* de grade é executado em serviços web para suportar colaborações entre cientistas ou engenheiros em organizações de diferentes partes do mundo. Muito freqüentemente, seu trabalho é baseado no uso de dados brutos coletados por instrumentos em diferentes lugares e depois processados de forma local. A estipulação de interfaces de serviço web permite que esses dados sejam acessados de forma remota. Entretanto, o requisito de iniciar o processamento de dados em nome de clientes

remotos leva à necessidade de criar instâncias de serviços sob demanda e de gerenciar seus tempos de vida. A arquitetura de grade adiciona fábricas nos serviços web, junto com um esquema de atribuição de nomes de dois níveis para referenciar instâncias de serviço. Além disso, ela fornece suporte para metadados que descrevem as características das instâncias de serviço. Ela também especifica serviços de diretório, gerenciamento e segurança. O *toolkit* Globus é uma implementação da arquitetura que tem sido usada em uma variedade de aplicações de uso intenso de dados e poder computacional.

Exercícios

19.1 Compare o protocolo de requisição e resposta descrito na Seção 4.4 com a implementação de comunicação cliente-servidor no SOAP. Cite dois motivos pelos quais o uso de mensagens assíncronas pelo SOAP é mais apropriado para uso na Internet. Até que ponto o uso de HTTP pelo SOAP reduz a diferença entre as duas estratégias? *páginas 678–679*

19.2 Compare a estrutura dos URLs, conforme usados pelos serviços web, com a das referências de objeto remoto, conforme especificadas na Seção 4.3.4. Cite, em cada caso, como elas são usadas para executar um pedido do cliente. *páginas 681–682*

19.3 Ilustre o conteúdo de uma mensagem SOAP de requisição e de sua mensagem de resposta correspondente para o serviço de eleição dado no Exercício 5.1. Use na sua resposta, uma versão pictórica da XML, como mostrado nas Figuras 19.4 e 19.5. *página 679*

19.4 Descreva em linhas gerais os cinco principais elementos de uma descrição do serviço WSDL. No caso do serviço *Election* definido no Exercício 5.1, cite o tipo de informação a ser usada pelas mensagens de requisição e resposta – algum deles precisa ser incluído no espaço de nomes de destino? Para a operação *vote*, desenhe diagramas semelhantes às Figuras 19.11 e 19.13. *páginas 689–690*

19.5 Continuando com o exemplo do serviço *Election*, explique por que a parte da WSDL definida no Exercício 19.4 é referida como abstrata. O que precisaria ser adicionado na descrição do serviço para torná-lo completamente concreto? *páginas 686–687*

19.6 Defina uma interface Java para o serviço *Election*, conveniente para uso como um serviço web. Diga por que você acha que a interface que definiu é conveniente. Explique como um documento WSDL para o serviço é gerado e como se torna disponível para os clientes. *página 683*

19.7 Descreva o conteúdo de um *proxy* de cliente Java para o serviço *Election*. Explique como os métodos de empacotamento e desempacotamento podem ser obtidos para um *proxy* estático. *páginas 683–685*

19.8 Explique a função de um contêiner de *servlet* na distribuição de um serviço web e na execução de uma requisição de cliente. *página 683*

19.9 No exemplo em Java ilustrado nas Figuras 19.8 e 19.9, o cliente e o servidor estão tratando com objetos, embora os serviços web não suportem objetos distribuídos. Como pode ser isso? Quais são as limitações impostas sobre as interfaces de serviços web em Java? *página 683*

19.10 Descreva em linhas gerais o esquema de replicação usado no UDDI. Supondo que indicações de tempo vetoriais são usadas para suportar esse esquema, defina duas operações para uso por registros que precisem trocar dados. *páginas 692–693*

19.11 Explique por que o UDDI pode ser descrito como serviço de nome e como serviço de diretório, mencionando os tipos de perguntas que podem ser feitas. O segundo 'D' no nome UDDI se refere a descoberta – o UDDI é realmente um serviço de descoberta? *Capítulo 9 e página 691*

19.12 Descreva em linhas gerais a principal diferença entre TLS e segurança da XML. Explique por que a XML é particularmente conveniente para a função que desempenha, em termos dessas diferenças. *páginas 692–693*

19.13 Os documentos protegidos pela segurança da XML podem ser assinados ou cifrados muito tempo antes que alguém possa prever quem serão os destinatários finais. Quais medidas são adotadas para garantir que estes últimos tenham acesso aos algoritmos usados pelo primeiro? *páginas 692–693*

19.14 Explique a relevância da XML canônica nas assinaturas digitais. Quais informações contextuais podem ser incluídas na forma canônica? Dê um exemplo de brecha de segurança onde o contexto é omitido da forma canônica. *páginas 695–696*

Hidden page

Hidden page

20.1 Introdução

O OMG (*Object Management Group*) foi formado em 1989 com o objetivo de estimular a adoção de sistemas de objetos distribuídos para usufruir das vantagens da programação orientada a objetos no desenvolvimento de software para sistemas distribuídos, os quais estavam se tornando muito difundidos. Para atingir seus objetivos, o OMG defendeu o uso de sistemas abertos baseados em interfaces orientadas a objetos padrão. Esses sistemas seriam construídos a partir de hardware, redes de computadores, sistemas operacionais e linguagens de programação heterogêneos.

Uma motivação importante foi permitir que os objetos distribuídos fossem implementados em qualquer linguagem de programação e pudessem se comunicar uns com os outros. Portanto, foi projetada uma linguagem de interface independente de qualquer linguagem de implementação específica.

Foi introduzida uma metáfora, o ORB (ou *object request broker* – agente de requisição de objeto), cuja função é ajudar um cliente a invocar um método em um objeto. Essa função envolve localizar o objeto, invocá-lo, se necessário, e então comunicar a requisição do cliente para o objeto, o qual o executa e responde.

Em 1991, um grupo de empresas estabeleceu uma especificação para uma arquitetura de agente de requisição de objeto, conhecida como CORBA (*Common Object Request Broker Architecture*). Em seguida, em 1996, surgiu a especificação CORBA 2.0, que definia padrões permitindo que implementações feitas por diferentes desenvolvedores se comunicassem. Esses padrões são chamados de *General Inter-ORB Protocol* ou GIOP. Pretendia-se que o protocolo GIOP pudesse ser implementado sobre qualquer camada de transporte orientado a conexões. A implementação do GIOP para a Internet usa o protocolo TCP e é chamada de *Internet Inter-ORB Protocol*, ou IIOP [OMG 2004a]. O CORBA 3 apareceu no final de 1999 e um modelo de componente foi adicionado recentemente.

Os principais componentes da estrutura de RMI independente de linguagem do CORBA são os seguintes:

- Uma linguagem de definição de interface conhecida como IDL, que será ilustrada inicialmente na Seção 20.2 e descrita mais completamente na Seção 20.2.3.
- Uma arquitetura, que será discutida na Seção 20.2.2.
- O protocolo GIOP define uma representação externa de dados, chamada CDR, que foi descrita na Seção 4.3. Ele também define formatos específicos para as mensagens em um protocolo de requisição e resposta. Além das mensagens de requisição e resposta, ele especifica mensagens para fazer perguntas sobre a localização de um objeto, cancelar requisições e relatar erros.
- O IIOP, uma implementação de GIOP, define uma forma padrão para referências de objeto remoto, que será descrita na Seção 20.2.4.

A arquitetura CORBA também admite serviços CORBA – um conjunto de serviços genéricos que podem ser úteis para aplicações distribuídas. Eles serão apresentados na Seção 20.3, que inclui uma discussão mais detalhada sobre os serviços de nomes (*Naming Service*), de eventos (*Event Service*), de notificação (*Notification Service*) e o de segurança (*Security Service*). Para ver um conjunto interessante de artigos sobre CORBA, consulte a edição especial do *CACM* [Seetharamanan 1998].

Antes de discutirmos esses componentes do CORBA, apresentaremos a RMI do CORBA do ponto de vista do programador.

20.2 CORBA RMI

Programar em um sistema RMI de múltiplas linguagens, como o RMI CORBA, exige mais do programador se comparado com o desenvolvimento em um sistema RMI de linguagem única, como o RMI do Java. Os seguintes novos conceitos precisam ser aprendidos:

- o modelo de objeto oferecido pelo CORBA;
- a linguagem de definição de interface e seu mapeamento para a linguagem de implementação.

Outros aspectos da programação em CORBA são semelhantes àqueles discutidos no Capítulo 5. Em particular, o programador define interfaces remotas para os objetos remotos e depois usa um compilador de interface para produzir os *proxies* e esqueletos correspondentes. Mas, no CORBA, os *proxies* são gerados na linguagem do cliente e os esqueletos, na linguagem do servidor. Usaremos o exemplo do quadro compartilhado apresentado na Seção 5.5 para ilustrar como se faz para escrever uma especificação de IDL e para construir programas servidores e clientes.

Modelo de objeto do CORBA ◊ O modelo de objeto do CORBA é semelhante àquele descrito na Seção 5.2, mas os clientes não são necessariamente objetos – um cliente pode ser qualquer programa que envie mensagens de requisição para objetos remotos e receba respostas. O termo *objeto CORBA* é usado para se referir aos objetos remotos. Assim, um objeto CORBA implementa uma interface IDL, tem uma referência de objeto remoto e é capaz de responder às invocações de métodos em sua interface IDL. Um objeto CORBA pode ser implementado por uma linguagem que não seja orientada a objetos sem o conceito de classe, por exemplo. Como as linguagens de implementação terão diferentes noções de classe, ou mesmo nenhuma noção, o conceito de classe não existe no CORBA. Portanto, classes não podem ser definidas na IDL do CORBA, o que significa que instâncias de classes não podem ser passadas como argumentos. Entretanto, estruturas de dados de vários tipos e complexidade arbitrária podem ser passadas como argumentos.

IDL CORBA ◊ A IDL do CORBA especifica um nome e um conjunto de métodos que os clientes podem solicitar. A Figura 20.1 mostra duas interfaces, chamadas *Shape* (linha 3) e *ShapeList* (linha 5), que são versões de IDL das interfaces definidas na Figura 5.12. Elas são precedidas por definições de duas estruturas (*structs*), as quais são usadas como tipos de parâmetro na definição dos métodos. Note, em particular, que *GraphicalObject* é definido como *struct*, embora fosse uma classe no exemplo de RMI em Java. Um componente cujo tipo é *struct* tem um conjunto de campos contendo valores de diferentes tipos, como as variáveis de instância de um objeto, mas não tem métodos. Há mais informações sobre IDL na Seção 20.2.3.

Parâmetros e resultados na IDL CORBA: cada parâmetro é marcado como sendo de entrada, saída ou ambos, usando-se as palavras-chaves *in*, *out* ou *inout*. A Figura 5.2 ilustra um exemplo simples do uso dessas palavras-chaves. Na Figura 20.1, linha 7, o parâmetro de *newShape* serve para indicar que o argumento deve ser passado do cliente para o servidor na mensagem de requisição. O valor de retorno fornece um parâmetro *out* adicional – ele pode ser indicado como *void*, caso não exista nenhum parâmetro *out*.

Os parâmetros podem ser qualquer um dos tipos primitivos, como *long* e *boolean*, ou um dos tipos construídos, como *struct* ou *array*. Os tipos primitivos e estruturados serão descritos com mais detalhes na Seção 20.2.3. Nosso exemplo mostra as definições de dois tipos *struct*, nas linhas 1 e 2. Seqüências e arrays são definidos em *typedefs*, como se vê na linha 4, que mostra uma seqüência de elementos de tipo *Shape* de comprimento 100. A semântica da passagem de parâmetros é a seguinte:

> *Passagem de objetos CORBA*: qualquer parâmetro cujo tipo seja especificado pelo nome de uma interface IDL, como o valor de retorno *Shape* na linha 7, é uma referência para um objeto CORBA e é passado o valor de uma referência de objeto remoto.
>
> *Passagem de tipos primitivos e construídos CORBA*: os argumentos de tipos primitivos e construídos são copiados e passados por valor. Na chegada, um novo valor é criado no processo do destinatário. Por exemplo, o *struct GraphicalObject* passado como argumento (na linha 7) produz uma nova cópia desse *struct* no servidor.

Essas duas formas de passagem de parâmetro são combinadas no método *allShapes* (na linha 8), cujo tipo de retorno é um array de tipo *Shape* – isto é, um array de referências de objeto remoto. O valor de retorno é uma cópia do array na qual cada um dos elementos é uma referência de objeto remoto.

Tipo *Object*. *Object* é o nome de um tipo cujos valores são referências de objeto remoto. Ele é efetivamente um supertipo comum de todos os tipos da interface IDL, como *Shape* e *ShapeList*.

Hidden page

Hidden page

isso não for feito, o objeto CORBA não poderá receber invocações remotas. Os leitores que estudaram o Capítulo 5 cuidadosamente poderão perceber que o registro do objeto no POA faz com que ele seja registrado no equivalente CORBA da tabela de objetos remotos.

Em nosso exemplo, o servidor contém implementações das interfaces *Shape* e *ShapeList* na forma de duas classes serventes, junto com uma classe servidora que contém uma seção *initialization* (veja a Seção 5.2.5) em seu método *main*.

As classes serventes: cada classe servente estende a classe esqueleto correspondente e implementa os métodos de uma interface IDL usando as assinaturas de método definidas na interface Java equivalente. A classe servente que implementa a interface *ShapeList* é chamada *ShapeListServant*, embora qualquer outro nome pudesse ser escolhido. Seu esboço aparece na Figura 20.3. Considere o método *newShape* na linha 1, que é um método de fábrica, pois cria objetos *Shape*. Para transformar um objeto *Shape* em um objeto CORBA, ele é registrado no POA por intermédio de seu método *servant_to_reference*, como mostrado na linha 2, que faz uso da referência ao POA raiz, que foi passado por meio do construtor quando a classe servente foi criada. Versões completas da interface IDL e das classes cliente e servidor deste exemplo estão disponíveis no endereço www.cdk4.net/corba.

O servidor: o método *main* na classe servidora *ShapeListServer* aparece na Figura 20.4. Primeiramente, ele cria e inicializa o ORB (linha 1). Ele obtém uma referência para o POA raiz e ativa o POAManager (linhas 2 e 3). Em seguida, ele cria uma instância de *ShapeListServant*, que é apenas um objeto Java (linha 4) e, ao fazer isso, passa uma referência para o POA raiz. Então, ele o transforma em um objeto CORBA, registrando-o no POA (linha 5). Depois disso, ele registra o servidor no *Naming Service*. Então, ele espera receber requisições de clientes (linha 10).

```
import org.omg.CORBA.*;
import org.omg.PortableServer.POA;
class ShapeListServant extends ShapeListPOA {
    private POA theRootpoa;
    private Shape theList[];
    private int version;
    private static int n=0;
    public ShapeListServant(POA rootpoa){
        theRootpoa = rootpoa;
            // inicializa as outras variáveis de instância
    }
    public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException {      1
        version++;
        Shape s = null;
        ShapeServant shapeRef = new ShapeServant(g, version);
        try {
        org.omg.CORBA.Object ref = theRoopoa.servant_to_reference(shapeRef);              2
        s = ShapeHelper.narrow(ref);
        } catch (Exception e) {}
        if(n >=100) throw new ShapeListPackage.FullException();
        theList[n++] = s;
        return s;
    }
    public Shape[] allShapes(){... }
    public int getVersion() {... }
}
```

Figura 20.3 Classe *ShapListServant* do programa servidor em Java da interface CORBA *ShapeList*.

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
public class ShapeListServer {
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);                                          1
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA")); 2
            rootpoa.the_POAManager().activate();                                     3
            ShapeListServant SLSRef = new ShapeListServant(rootpoa);                 4
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(SLSRef);         5
            ShapeList SLRef = ShapeListHelper.narrow(ref);
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);                6
            NameComponent nc = new NameComponent("ShapeList", "");                   7
            NameComponent path[] = {nc};                                             8
            ncRef.rebind(path, SLRef);                                               9
            orb.run();                                                               10
        } catch (Exception e) {... }
    }
}
```

Figura 20.4 Classe Java *ShapeListServer*.

Os servidores que usam o *Naming Service*, primeiro obtêm um contexto de atribuição de nomes raiz (linha 6), depois fazem um *NameComponent* (linha 7), definem um caminho (linha 8) e, finalmente, usam o método *rebind* (linha 9) para registrar o nome e a referência de objeto remoto. Os clientes executam os mesmos passos, mas utilizam o método *resolve*, como mostrado na Figura 20.5, linha 2.

O programa cliente ◊ Um exemplo de programa cliente aparece na Figura 20.5. Ele cria e inicializa um ORB (linha 1) e depois entra em contato com o *Naming Service* para obter uma referência para o objeto remoto *ShapeList*, usando seu método *resolve* (linha 2). Depois disso, ele invoca seu método *allShapes* (linha 3) para obter uma seqüência de referências de objeto remoto para todos os objetos *Shape* correntemente mantidos no servidor. Em seguida, ele invoca o método *getAllState* (linha 4), fornecendo como argumento a primeira referência de objeto remoto da seqüência retornada; o resultado é fornecido como uma instância da classe *GraphicalObject*.

O método *getAllState* parece contradizer nossa afirmação anterior de que objetos não podem ser passados por valor no CORBA, pois tanto o cliente como o servidor transacionam em instâncias da classe *GraphicalObject*. Entretanto, não há nenhuma contradição: o objeto CORBA retorna uma *struct* e os clientes que estiverem usando uma linguagem diferente poderão vê-la de forma diferente. Por exemplo, na linguagem C++, o cliente a veria como uma *struct*. Mesmo em Java, a classe *GraphicalObject* gerada é mais parecida com uma *struct*, pois não tem métodos.

Os programas clientes sempre devem capturar exceções CORBA *SystemExceptions*, as quais relatam erros decorrentes da distribuição (veja a linha 5). Os programas clientes também devem capturar as exceções definidas na interface IDL, como a *FullException* disparada pelo método *newShape*.

Esse exemplo ilustra o uso da operação *narrow*: a operação *resolve* do *Naming Service* retorna um valor de tipo *Object*; esse tipo é reduzido para estar de acordo com o tipo em particular exigido – *ShapeList*.

Callbacks ◊ *Callbacks* podem ser implementadas no CORBA de uma maneira semelhante àquela descrita para RMI Java, na Seção 5.5.1. Por exemplo, a interface *WhiteboardCallback* pode ser definida como segue:

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ShapeListClient{
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);                                    1
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path [] = { nc };
            ShapeList shapeListRef =
            ShapeListHelper.narrow(ncRef.resolve(path));                       2
            Shape[] sList = shapeListRef.allShapes();                          3
            GraphicalObject g = sList[0].getAllState();                        4
        } catch(org.omg.CORBA.SystemException e) {...}
    }
}
```

Figura 20.5 Programa cliente em Java para as interfaces CORBA *Shape* e *ShapeList*.

```
interface WhiteboardCallback {
    oneway void callback(in int version);
};
```

Essa interface é implementada como um objeto CORBA pelo cliente, permitindo que o servidor envie ao cliente um número de versão quando novos objetos são adicionados. Mas antes que o servidor possa fazer isso, o cliente precisa informá-lo da referência de objeto remoto de seu objeto. Para tornar isso possível, a interface *ShapeList* exige métodos adicionais, como *register* e *deregister*, como segue:

```
int register(in WhiteboardCallback callback);
void deregister(in int callbackId);
```

Após um cliente ter obtido uma referência para o objeto *ShapeList* e criado uma instância de *WhiteboardCallback*, ele usa o método *register* de *ShapeList* para informar o servidor de que está interessado em receber *callbacks*. O objeto *ShapeList* no servidor é responsável por manter uma lista de clientes interessados e notificar a todos eles sempre que seu número de versão aumentar, quando um novo objeto for adicionado. O método *callback* é declarado como *oneway* para que o servidor possa usar chamadas assíncronas para evitar atraso ao notificar cada cliente.

20.2.2 A arquitetura CORBA

A arquitetura é projetada para suportar a função de um agente de requisição de objeto (ORB – *Object Request Broker*) que permite aos clientes invocarem métodos em objetos remotos, onde clientes e servidores podem ser implementados em uma variedade de linguagens de programação. Os principais componentes da arquitetura CORBA estão ilustrados na Figura 20.6.

Essa figura deve ser comparada com a Figura 5.7, no caso em que será notado que a arquitetura CORBA contém três componentes adicionais: o adaptador de objeto, o repositório de implementações e o repositório de interfaces.

O CORBA fornece invocações estáticas e dinâmicas. As invocações estáticas são usadas quando a interface remota do objeto CORBA é conhecida no momento da compilação, permitindo o uso de *stubs* de cliente e esqueletos de servidor. Se a interface remota não for conhecida no momento da compilação, a invocação dinâmica deverá ser usada. A maioria dos programadores prefere usar invocação estática, pois ela fornece um modelo de programação mais natural.

Hidden page

O POA permite que objetos CORBA sejam instanciados de forma transparente e, além disso, ele separa a criação de objetos CORBA da criação dos *serventes* que implementam esses objetos. Aplicações servidoras, como bancos de dados, com grandes números de objetos CORBA, podem criar serventes sob demanda, apenas quando os objetos são acessados. Nesse caso, elas podem usar chaves de banco de dados para os nomes de objeto, Como alternativa, elas podem usar um único servente para suportar todos esses objetos.

Além disso, é possível especificar políticas para o POA; por exemplo, se ele deve fornecer uma *thread* para cada invocação, se as referências de objeto devem ser persistentes ou transientes e se deve haver um servente separado para cada objeto CORBA. O padrão é que um único servente pode representar todos os objetos CORBA para seu POA.

Esqueletos ◊ As classes de esqueleto são geradas na linguagem do servidor por um compilador de IDL. Como antes, as invocações de método remotas são enviadas por meio do esqueleto apropriado para um servente em particular e o esqueleto desempacota os argumentos nas mensagens de requisição e empacota exceções e resultados nas mensagens de resposta.

***Stubs/proxies* de cliente** ◊ Eles são escritos na linguagem do cliente. A classe de um *proxy* (para linguagens orientadas a objetos) ou um conjunto de procedimentos de *stub* (para linguagens procedurais) é gerada a partir de uma interface IDL por um compilador de IDL para a linguagem do cliente. Como antes, os *stubs/proxies* de cliente empacotam os argumentos nas requisições de invocação e desempacotam exceções e resultados nas respostas.

Repositório de implementações ◊ Um repositório de implementações é responsável por ativar servidores registrados sob demanda e por localizar os servidores que estão correntemente em execução. O nome do adaptador de objeto é usado para se referir aos servidores ao registrá-los e ativá-los.

Um repositório de implementações armazena um mapeamento dos nomes de adaptadores de objeto para nomes de caminho de arquivos contendo implementações de objeto. As implementações de objeto e os nomes de adaptadores de objeto geralmente são registrados no repositório de implementações quando os programas servidores são instalados. Quando as implementações de objeto são ativadas nos servidores, o nome de computador servidor (*hostname*) e o seu número da porta são adicionados no mapeamento.

Entrada do repositório de implementações:

nome do adaptador de objeto	nome de caminho da implementação de objeto	*hostname* e número da porta do servidor

Nem todos os objetos CORBA precisam ser ativados sob demanda. Alguns objetos, por exemplo, objetos de *callback* criados pelos clientes, são executados uma vez e deixam de existir quando não são mais necessários. Eles não usam o repositório de implementações.

Geralmente, um repositório de implementações permite que sejam armazenadas informações extras sobre cada servidor, por exemplo, informações de controle de acesso sobre quem pode ativá-lo ou executar suas operações. É possível replicar informações nos repositórios de implementação para fornecer disponibilidade ou tolerância a falhas.

Repositório de interfaces ◊ A função do repositório de interfaces é fornecer informações sobre as interfaces IDL registradas para clientes e servidores que as exigirem. Para uma interface de determinado tipo, ele pode fornecer os nomes dos métodos e, para cada método, os nomes e tipos dos argumentos e exceções. Assim, o repositório de interfaces adiciona um recurso de reflexão no CORBA. Suponha que um programa cliente receba uma referência remota para um novo objeto CORBA. Além disso, suponha que o cliente não tenha *proxy* para ele; então, ele pode perguntar ao repositório de interfaces sobre os métodos do objeto e os tipos de parâmetro que cada um deles exige.

Quando um compilador de IDL processa uma interface, ele atribui um identificador de tipo a cada tipo de IDL que encontra. Para cada interface registrada nele, o repositório de interfaces fornece um mapeamento entre o identificador de tipo dessa interface e a interface em si. Assim, o identificador de

tipo de uma interface às vezes é chamado de *ID de repositório*, pois pode ser usado como uma chave para as interfaces IDL registradas no repositório de interfaces.

Toda referência de objeto remoto CORBA inclui um *slot* que contém o identificador de tipo de sua interface, permitindo que os clientes que o possuem perguntem sobre seu tipo no repositório de interfaces. Essas aplicações que usam invocação estática (normal) com *proxies* clientes e esqueletos de IDL não exigem um repositório de interfaces. Nem todos os ORBs fornecem um repositório de interfaces.

Interface de invocação dinâmica ◊ Conforme sugerido na Seção 5.5, em algumas aplicações pode ser necessário construir um programa cliente sem conhecer todas as classes *proxy* que ele precisará no futuro. Por exemplo, talvez um navegador de objeto precise exibir informações sobre todos os objetos CORBA disponíveis nos vários servidores em um sistema distribuído. Não é exeqüível que tal programa inclua *proxies* para todos esses objetos, particularmente porque novos objetos podem ser adicionados no sistema, à medida que o tempo passa. O CORBA não permite o *download* de classes de *proxies* em tempo de execução, como na RMI Java. A interface de invocação dinâmica é a alternativa do CORBA.

A interface de invocação dinâmica permite que os clientes façam invocações dinâmicas em objetos remotos CORBA. Ela é usada quando não é possível empregar *proxies*. O cliente pode obter do repositório de interfaces as informações necessárias sobre os métodos disponíveis para determinado objeto CORBA. O cliente pode usar essas informações para construir uma invocação com argumentos convenientes e enviá-la para o servidor.

Esqueletos dinâmicos ◊ Novamente, conforme explicado na Seção 5.5, pode ser necessário adicionar em um servidor um objeto CORBA cuja interface era desconhecida quando o servidor foi compilado. Se um servidor usa esqueletos dinâmicos, então ele pode aceitar invocações na interface de um objeto CORBA para as quais não possui esqueleto. Quando um esqueleto dinâmico recebe uma invocação, ele inspeciona o conteúdo da requisição para descobrir seu objeto de destino, o método a ser invocado e os argumentos. Então, ele ativa o destino.

Código legado ◊ O termo *código legado* se refere a código existente que não foi projetado tendo-se em mente os objetos distribuídos. Um código legado pode ser transformado em um objeto CORBA definindo-se uma interface IDL para ele e fornecendo-se a implementação de um adaptador de objeto apropriado e os esqueletos necessários.

20.2.3 Linguagem de definição de interface CORBA

A linguagem de definição de interface, ou simplesmente IDL (*Interface Definition Language*), CORBA fornece recursos para a definição de módulos, interfaces, tipos, atributos e assinaturas de método. Mostramos exemplos de tudo isso, fora os módulos, nas Figuras 5.2 e 20.1. A IDL tem as mesmas regras léxicas da linguagem C++, mas possui palavras-chaves adicionais para suportar distribuição, por exemplo, *interface*, *any*, *attribute*, *in*, *out*, *inout*, *readonly* e *raises*. Ela também permite recursos de pré-processamento padrão da linguagem C++. Veja, por exemplo, o *typedef* para *All* na Figura 20.7. A gramática da IDL é um subconjunto da linguagem C++ ANSI, com construções adicionais para suportar assinaturas de método. Fornecemos aqui apenas um breve panorama da IDL. Uma visão geral útil e muitos exemplos são dados em Baker [1997] e Henning e Vinoski [1999]. A especificação completa está disponível no site do OMG [OMG 2002a].

Módulos IDL ◊ A construção do módulo permite que as interfaces e outras definições de tipo da IDL sejam agrupadas em unidades lógicas. Um *módulo* define um escopo de atribuição de nomes, o qual impede que os nomes definidos dentro de um módulo colidam com os nomes definidos fora dele. Por exemplo, as definições das interfaces *Shape* e *ShapeList* poderiam pertencer a um módulo chamado *Whiteboard*, como mostrado na Figura 20.7.

Interfaces IDL ◊ Conforme já vimos, uma interface IDL descreve os métodos que estão disponíveis nos objetos CORBA que implementam essa interface. Os clientes de um objeto CORBA podem ser desenvolvidos apenas a partir do conhecimento de sua interface IDL. A partir do estudo de nossos

Hidden page

Hidden page

Hidden page

Hidden page

transiente contém os detalhes do endereço do servidor que contém o objeto CORBA, enquanto uma IOR persistente contém os detalhes do endereço do repositório de implementações no qual está registrada. Nos dois casos, o cliente ORB envia a mensagem de requisição para o servidor cujos detalhes do endereço são dados na IOR. Agora, discutiremos como a IOR é usada para localizar o servente que representa o objeto CORBA nos dois casos.

IORs transientes: o núcleo ORB servidor recebe a mensagem de requisição contendo o nome do adaptador de objeto e o nome do objeto do destino. Ele usa o nome do adaptador de objeto para localizar o adaptador de objeto, o qual utiliza o nome do objeto para localizar o servente.

IORs persistentes: um repositório de implementações recebe a requisição. Ele extrai o nome do adaptador de objeto da IOR presente na requisição. Desde que o nome do adaptador de objeto esteja em sua tabela, ele tenta, se necessário, ativar o objeto CORBA no endereço de *host* especificado em sua tabela. Uma vez que o objeto CORBA tenha sido ativado, o repositório de implementações retorna seus detalhes de endereço para o cliente ORB, o qual os utiliza como destino das mensagens de requisição RMI, que inclui o nome do adaptador de objeto e o nome do objeto. Isso permite que o núcleo ORB servidor localize o adaptador de objeto, o qual, como antes, utiliza o nome do objeto para localizar o servente.

O segundo campo de uma IOR pode ser repetido, de modo a especificar o nome de domínio do *host* e o número da porta de mais de um destino, para permitir que um objeto ou um repositório de implementações seja replicado em vários locais diferentes.

A mensagem de resposta no protocolo de requisição e resposta inclui informações de cabeçalho que permitem a execução do procedimento das IORs persistentes. Em particular, ela inclui uma entrada de status que pode indicar se a requisição deve ser encaminhada para um servidor diferente, no caso em que o corpo da resposta inclui uma IOR que contém o endereço do servidor do objeto recentemente ativado.

20.2.5 Mapeamentos de linguagem CORBA

Vimos, a partir de nossos exemplos, que o mapeamento dos tipos na IDL para tipos Java é muito simples. Os tipos primitivos na IDL são mapeados nos tipos primitivos correspondentes em Java. *Structs*, *enums* e *unions* são mapeados em classes Java; seqüências e arrays na IDL são mapeados em arrays em Java. Uma exceção da IDL é mapeada em uma classe Java que fornece variáveis de instância para os campos da exceção e construtores. Os mapeamentos em C++ são semelhantemente simples.

Entretanto, vimos que surgem algumas dificuldades no mapeamento da semântica da passagem de parâmetros da IDL para a da linguagem Java. Em particular, a IDL permite que os métodos retornem diversos valores separados por meio de parâmetros de saída, enquanto a linguagem Java só pode ter um único resultado. As classes *Holder* são fornecidas para suprir essa diferença, mas isso exige que o programador faça uso delas, o que não é nada simples. Por exemplo, o método *getPerson* da Figura 5.2 é definido na IDL como segue:

```
void getPerson(in string name, out Person p);
```

e o método equivalente na interface Java seria definido como:

```
void getPerson(String name, PersonHolder p);
```

e o cliente deve fornecer uma instância de *PersonHolder* como argumento de sua invocação. A classe portadora tem uma variável de instância que contém o valor do argumento para o cliente acessar quando a invocação retornar. Ela também tem métodos para transmitir o argumento entre servidor e cliente.

Embora as implementações em C++ do CORBA possam manipular parâmetros *out* e *inout* de forma bastante natural, os programadores de C++ sofrem com um conjunto de problemas diferentes nos parâmetros, relacionados ao gerenciamento do armazenamento. Essas dificuldades surgem quando referências de objeto e entidades de comprimento variável, como strings ou seqüências, são passados como argumentos.

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

- O protocolo IIOP (*Internet Inter-ORB Protocol*) implementa o protocolo de requisição e resposta sobre TCP. As referências de objeto remoto IIOP incluem o nome de domínio e o número da porta de um servidor.

Um objeto CORBA implementa as operações em uma interface IDL. Tudo que os clientes precisam conhecer para acessar um objeto CORBA são as operações disponíveis em sua interface. O programa cliente acessa objetos CORBA por meio de *proxies* ou *stubs*, os quais são gerados automaticamente a partir de suas interfaces IDL, na linguagem do cliente. Esqueletos de servidor para objetos CORBA são gerados automaticamente a partir de suas interfaces IDL, na linguagem do servidor. O adaptador de objeto é um componente importante dos servidores CORBA. Suas funções incluem ativar e desativar serventes, criar referências de objeto remoto e encaminhar mensagens de requisição para os serventes apropriados.

A arquitetura CORBA permite que objetos CORBA sejam ativados sob demanda. Isso é obtido por meio de um componente chamado repositório de implementações, o qual mantém um banco de dados de implementações indexadas por seus nomes de adaptador de objeto. Quando um cliente invoca um objeto CORBA, ele pode ser ativado, se necessário, para realizar essa invocação.

Um repositório de interfaces é um banco de dados de definições de interface IDL indexadas pelas IDs de repositório. Como a IOR de um objeto CORBA contém a ID de repositório de sua interface, o repositório de interfaces apropriado pode ser usado para se obter as informações sobre os métodos de sua interface que são exigidas para invocações dinâmicas a métodos.

Os serviços CORBA fornecem funcionalidade acima da RMI, o que pode ser exigido por aplicações distribuídas, permitindo que eles utilizem serviços adicionais, como os serviços de atribuição de nomes e de diretório, notificações de evento, transações ou segurança, conforme for exigido.

Exercícios

20.1 A *Task Bag* (sacola de tarefas) é um objeto que armazena pares de (chave e valor). Uma chave é um string e um valor é uma seqüência de bytes. Sua interface fornece os seguintes métodos remotos:

pairOut: com dois parâmetros, através dos quais o cliente especifica uma *chave* e um *valor* a ser armazenado.

pairIn: cujo primeiro parâmetro permite que o cliente especifique a *chave* de um par a ser removido da *Task Bag*. O valor no par é fornecido para o cliente por intermédio de um segundo parâmetro. Se nenhum par correspondente estiver disponível, uma exceção será lançada.

readPair: é igual a *pairIn*, exceto que o par permanece na *Task Bag*.

Use IDL do CORBA para definir a interface da *Task Bag*. Defina uma exceção que possa ser disparada quando qualquer uma das operações não puder ser executada. Sua exceção deve retornar um valor inteiro indicando o número do problema e um string descrevendo o problema. A interface da *Task Bag* deve definir um único atributo, fornecendo o número de tarefas na sacola. *página 720–721*

20.2 Defina uma assinatura alternativa para os métodos *pairIn* e *readPair*, cujo valor de retorno indica quando nenhum par correspondente está disponível. O valor de retorno deve ser definido como um tipo enumerado cujos valores podem ser *ok* e *wait*. Discuta os méritos relativos das duas estratégias alternativas. Qual estratégia você usaria para indicar um erro como uma chave contendo caracteres inválidos? *página 721*

20.3 Qual dos métodos na interface da *Task Bag* poderia ter sido definido como uma operação *oneway*? Forneça uma regra geral a respeito dos parâmetros e exceções de métodos *oneway*. De que maneira o significado da palavra-chave *oneway* difere do restante da IDL? *página 721*

20.4 O tipo *union* da IDL pode ser usado para um parâmetro que precisará passar um tipo de um pequeno número de tipos. Utilize-o para definir o tipo de um parâmetro que às vezes é vazio e às vezes tem o tipo *Value*. *página 722–723*

20.5 Na Figura 20.1, o tipo *All* foi definido como uma seqüência de comprimento fixo. Redefina isso como um array de mesmo comprimento. Dê algumas recomendações quanto à escolha entre arrays e seqüências em uma interface IDL. *página 722–723*

20.6 A *Task Bag* se destina a ser usada por clientes em cooperação, alguns dos quais adicionam pares (descrevendo tarefas) e outros os removem (e executam as tarefas descritas). Quando um cliente é informado de que não há nenhum par correspondente disponível, ele não pode continuar com seu trabalho até que um par se torne disponível. Defina uma interface de *callback* apropriada para uso nessa situação. *página 716–717*

20.7 Descreva as modificações necessárias na interface da *Task Bag* para permitir o uso de *callbacks*. *página 716–717*

20.8 Quais dos parâmetros dos métodos na interface da *Task Bag* são passados por valor e quais são passados por referência? *página 712–713*

20.9 Use o compilador de IDL Java para processar a interface que você definiu no Exercício 20.1. Inspecione a definição das assinaturas dos métodos *pairIn* e *readPair* no equivalente em Java gerado da interface IDL. Procure também, na definição gerada do método portador, o argumento de valor dos métodos *pairIn* e *readPair*. Agora, dê um exemplo mostrando como o cliente invocará o método *pairIn*, explicando como adquirirá o valor retornado por meio do segundo argumento. *página 725*

20.10 Dê um exemplo para mostrar como um cliente Java acessará o atributo que fornece o número de tarefas no objeto *Task Bag*. Sob que aspectos um atributo difere de uma variável de instância de um objeto? *página 721–723*

20.11 Explique por que as interfaces de objetos remotos em geral e os objetos CORBA em particular não fornecem construtores. Explique como os objetos CORBA podem ser criados na ausência de construtores. *Capítulo 5 e página 715*

20.12 Redefina a interface da *Task Bag* do Exercício 20.1 na IDL, de modo que ela faça uso de uma *struct* para representar um *Par*, o qual consiste em uma *Chave* e em um *Valor*. Note que não há necessidade de usar um *typedef* para definir uma *struct*. *página 722–723*

20.13 Discuta as funções do repositório de implementações do ponto de vista da capacidade de mudança de escala e da tolerância a falhas. *página 719–720, página 724–725*

20.14 Até que ponto os objetos CORBA podem ser migrados de um servidor para outro? *página 719–720, página 724–725*

20.15 Discuta as vantagens e desvantagens dos nomes de duas partes ou *NameComponents* no serviço de atribuição de nomes do CORBA. *página 727–728*

20.16 Forneça um algoritmo que descreva como um nome de várias partes é solucionado no serviço de atribuição de nomes do CORBA. Um programa cliente precisa solucionar um nome de várias partes com componentes "A", "B" e "C", relativo a um contexto de atribuição de nomes inicial. Como ele especificaria os argumentos da operação *resolve* no serviço de atribuição de nomes? *página 727–728*

20.17 Uma empresa virtual consiste em um conjunto de empresas que estão colaborando umas com as outras para executar um projeto em particular. Cada empresa deseja fornecer às outras acesso apenas aos seus objetos CORBA relevantes para o projeto. Descreva uma maneira apropriada para o grupo aliar seus *Naming Services*. *página 729–730*

20.18 Discuta como se faz para usar fornecedores e consumidores diretamente conectados do serviço de evento CORBA no contexto da aplicação do quadro compartilhado. As interfaces *PushConsumer* e *PushSupplier* são definidas na IDL como segue:

```
interface PushConsumer {
        void push(in any data) raises (Disconnected);
        void disconnect_push_consumer();
}
interface PushSupplier {
        void disconnect_push_supplier();
}
```

O fornecedor ou o consumidor pode decidir terminar a comunicação de eventos, chamando *disconnect_push_supplier*() ou *disconnect_push_consumer*(), respectivamente. *página 730*

20.19 Descreva como se faz a interposição de um canal de evento entre o fornecedor e os consumidores em sua solução para o Exercício 20.18. Um canal de evento tem a seguinte interface IDL:

Hidden page

Hidden page

Abadi and Gordon 1999 — Abadi, M. e Gordon, A.D. (1999). A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, Vol. 148, No. 1, pgs. 1–70, Janeiro.

Abadi *et al.* 1998 — Abadi, M., Birrell, A.D., Stata, R. e Wobber, E.P. (1998). Secure Web tunneling. Em *Proceedings 7th International World Wide Web Conference*, pgs. 531–9. Elsevier, em *Computer Networks and ISDN Systems*, Volume 30, Nos 1–7.

Abrossimov *et al.* 1989 — Abrossimov, V., Rozier, M. e Shapiro, M. (1989). Generic virtual memory management for operating system kernels. *Proceedings of 12th ACM Symposium on Operating System Principles*, pgs. 123–36, Dezembro.

Accetta *et al.* 1986 — Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. e Young, M. (1986). Mach: A new kernel foundation for UNIX development. Em *Proceedings Summer 1986 USENIX Conference*, pgs. 93–112.

Adjie-Winoto *et al.* 1999 — Adjie-Winoto, W., Schwartz, E., Balakrishnan, H. e Lilley, J. (1999). The design and implementation of an intentional naming system. Em *Proceedings 17th ACM Symposium on Operating System Principles*, publicado como *Operating Systems Review*, Vol. 34, No. 5, pgs. 186–201.

Adve e Hill 1990 — Adve, S. e Hill, M. (1990). Weak ordering – a new definition. Em *Proceedings 17th. Annual Symposium on Computer Architecture*, IEEE, pgs. 2–14.

Agrawal *et al.* 1987 — Agrawal, D., Bernstein, A., Gupta, P. e Sengupta, S. (1987). Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, Vol. 2: pgs. 45–59. Springer-Verlag.

Ahamad *et al.* 1992 — Ahamad, M., Bazzi, R., John, R., Kohli, P. e Neiger, G. (1992). *The Power of Processor Consistency*. Technical report GIT-CC-92/34, Georgia Institute of Technology, Atlanta.

Al-Muhtadi *et al.* 2002 — Al-Muhtadi, J., Campbell, R., Kapadia, A., Mickunas, D. e Yi, S. (2002). Routing Through the Mist: Privacy Preserving Communication in Ubiquitous Computing Environments. *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Viena, Áustria, Julho, pgs. 74–83.

Alonso *et al.* 2004 — Alonso, G., Casata, C., Kuno, H. e Machiraju, V. (2004) *Web Services, Concepts, Architectures and Applications*. Berlim, Heidelberg: Springer-Verlag.

Anderson 1993 — Anderson, D.P. (1993). Meta-scheduling for distributed continuous media. *ACM Transactions on Computer Systems*, Vol. 11, No. 3.

Anderson 1996 — Anderson, R. J. (1996). The Eternity Service. Em *Proceedings of Pragocrypt'96*.

Anderson 2001 — Anderson, R.J. (2001). *Security Engineering*. John Wiley & Sons.

Anderson *et al.* 1990a — Anderson, D.P., Herrtwich, R.G. e Schaefer, C. (1990). SRP – *A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet*. Technical report 90-006, International Computer Science Institute, Berkeley, CA.

Anderson *et al.* 1990b — Anderson, D.P., Tzou, S., Wahbe, R., Govindan, R. e Andrews, M. (1990). Support for continuous media in the DASH System. *Tenth International Conference on Distributed Computing Systems*, Paris.

Anderson *et al.* 1991 — Anderson, T., Bershad, B., Lazowska, E. e Levy, H. (1991). Scheduler activations: efficient kernel support for the user-level management of parallelism. Em *Proceedings 13th ACM Symposium on Operating System Principles*, pgs. 95–109.

Anderson *et al.* 1995 — Anderson, T., Culler, D., Patterson, D. e a equipe NOW. (1995). A case for NOW (Networks Of Workstations), *IEEE Micro*, Vol. 15, No. 1.

Anderson *et al.* 1996 — Anderson, T.E., Dahlin, M.D., Neefe, J.M., Patterson, D.A., Roselli, D.S. e Wang, R.Y. (1996). Serverless Network File Systems. *ACM Trans. on Computer Systems*, Vol. 14, No. 1, pgs. 41–79. Fevereiro.

Anderson *et al.* 2002 — Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M. e Werthimer, D. (2002). SETI@ home: An experiment in public-resource computing. *Communications of the ACM*, Nov. de 2002, Vol. 45, No. 11, pgs. 56–61.

Anderson *et al.* 2004 — Anderson, R., Chan, H. e Perrig, A. (2004). Key Infection: Smart Trust for Smart Dust. *Proceedings of IEEE 12th International Conference on Network Protocols (ICNP 2004)*, Berlim, Alemanha, Outubro, pgs. 206–215.

ANSA 1989 — ANSA (1989). *The Advanced Network Systems Architecture (ANSA) Reference Manual*. Castle Hill, Cambridge, Inglaterra: Architecture Project Management.

ANSI 1985 — American National Standards Institute (1985). *American National Standard for Financial Institution Key Management*, Standard X9.17 (revisado).

Apache 2004 — The Apache foundation (2004). *Apache Tomcat*.

Apple Computer 2004 — Apple Computer (2004). iChat AV *Video conferencing for the rest of us*.

Armand *et al*. 1989 — Armand, F., Gien, M., Herrman, F. e Rozier, M. (1989). Distributing UNIX brings it back to its original virtues. *Proc.Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pgs. 153–174, Outubro.

Arnold *et al*. 1999 — Arnold, K., O'Sullivan, B., Scheifler, R.W., Waldo, J. e Wollrath, A. (1999). *The Jini Specification*, Reading, MA: Addison-Wesley.

associates.amazon.com — Amazon Web Service FAQs.

Attiya e Welch1998 — Attiya, H. e Welch, J. (1998). *Distributed Computing – Fundamentals, Simulations and Advanced Topics*. McGraw-Hill.

Babaoglu *et al*. 1998 — Babaoglu, O., Davoli, R., Montresor, A. e Segala, R. (1998). System support for partition-aware network applications. Em *Proceedings 18th InternationalConference on Distributed Computing Systems* (ICDCS 98), pgs. 184–191.

Bacon 2002 — Bacon, J. (2002). *Concurrent Systems*, terceira edição. Harlow, Inglaterra: Addison-Wesley.

Baghwan *et al*. 2003 — Bhagwan, R., Savage, S. e Voelker, G. (2003). Understanding availability. Em *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, Fev. de 2003.

Baker 1997 — Baker, S. (1997). *CORBA Distributed Objects Using Orbix*. Harlow, Inglaterra: Addison-Wesley.

Bal *et al*. 1990 — Bal, H.E., Kaashoek, M.F. e Tanenbaum, A.S. (1990). Experience with distributed programming in Orca. Em *Proceedings International Conference on Computer Languages'90*, IEEE, pgs. 79–89.

Balakrishnan *et al*. 1995 — Balakrishnan, H., Seshan, S. e Katz, R.H. (1995). Improving reliable transport and hand-off performance in cellular wireless networks. Em *Proceedings ACM Mobile Computing and Networking Conference*, ACM, pgs. 2–11.

Balakrishnan *et al*. 1996 — Balakrishnan, H., Padmanabhan, V., Seshan, S. e Katz, R. (1996). A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. *Proceedings of the ACM SIGCOMM'96 Conference*, pgs. 256–69.

Balan *et al*. 2003 — Balan, R.K., Satyanarayanan, M., Park, S., Okoshi, T. (2003). Tactics-Based Remote Execution for Mobile Computing. *Proceedings First USENIX International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, São Francisco, CA, EUA, Maio, pgs. 273–286.

Balazinska *et al*. 2002 — Balazinska, M., Balakrishnan, H. e Karger, D. (2002). INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. *Proceedings Pervasive 2002 – International Conference on Pervasive Computing*, Zurique, Suíça, Agosto, pgs. 195–210.

Balfanz *et al*. 2002 — Balfanz, D., Smetters, D.K., Stewart, P. e Wong, H.C. (2002). Talking to strangers: authentication in ad-hoc wireless networks. *Proceedings Network and Distributed System Security Symposium*, San Diego, CA, EUA, Fevereiro.

Banerjea e Mah 1991 — Banerjea, A. e Mah, B.A. (1991). The real-time channel administration protocol. *Second International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg.

Baran 1964 — Baran, P. (1964). *On Distributed Communications*. Research Memorandum RM-3420-PR, Rand Corporation.

Barborak *et al*. 1993 — Barborak, M., Malek, M. e Dahbura, A. (1993). The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, Vol. 25, No. 2, pgs. 171–220.

Barghouti e Kaiser 1991 — Barghouti, N.S. e Kaiser, G.E. (1991). Concurrency control in advanced database applications. *Computing Surveys*, Vol. 23, No. 3, pgs. 269–318.

Barham *et al*. 2003 — Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. e Warfield, A. (2003). Xen and the art of virtualization. *Proceedings*

nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, EUA, Outubro, pgs. 164–177.

Barr e Asanovic 2003 — Barr, K. e Asanovic, K. (2003). Energy Aware Lossless Data Compression. *Proceedings First USENIX International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, São Francisco, CA, EUA, Maio, pgs. 231–244.

Bartoli *et al.* 1993 — Bartoli, A., Mullender, S.J. e van der Valk, M. (1993). Wide-address spaces – exploring the design space. *ACM Operating Systems Review*, Vol. 27, No. 1, pgs. 11–17.

Barton *et al.* 2002 — Barton, J., Kindberg, T. e Sadalgi, S. (2002). Physical Registration: Configuring Electronic Directories using Handheld Devices. *IEEE Wireless Communications*, Vol. 9, No. 1, Fevereiro, pgs. 30–38.

Bates *et al.* 1996 — Bates, J., Bacon, J., Moody, K. e Spiteri, M. (1996). Using events for the scalable federation of heterogeneous components, *European SIGOPS Workshop*.

Bell e LaPadula 1975 — Bell, D.E. e LaPadula, L.J. (1975). *Computer Security Model: Unified Exposition and Multics Interpretation*. Mitre Corporation, 1975.

Bellman 1957 — Bellman, R.E. (1957). *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Bellovin e Merritt 1990 — Bellovin, S.M. e Merritt, M. (1990). Limitations of the Kerberos authentication system. *ACM Computer Communications Review*, Vol. 20, No. 5, pgs. 119–32.

Bellwood *et al.* 2003 — Bellwood, T., Clément, L. e von Riegen, C. (eds) (2003). *UDDI Version 3.0.1*. Oasis Corporation.

Beresford e Stajano 2003 — Beresford, A. e Stajano, F. (2003). Location Privacy in Pervasive Computing. *IEEE Pervasive Computing*, Vol. 2, No. 1, Jan.–Mar., pgs. 46–55.

Berners-Lee 1991 — Berners-Lee, T. (1991). World Wide Web Seminar.

Berners-Lee 1999 — Berners-Lee, T. (1999). *Weaving The Web*. HarperCollins.

Berners-Lee *et al.* 2005 — Berners Lee, T., Fielding, R. e Masinter, L. (2005). Uniform Resource Identifiers (URI): Generic Syntax, Agosto. Internet RFC 3986.

Bernstein *et al.* 1980 — Bernstein, P.A., Shipman, D.W. e Rothnie, J.B. (1980). Concurrency control in a system for distributed databases (SDD-1). *ACM Transactions Database Systems*, Vol. 5, No. 1, pgs. 18–51.

Bernstein *et al.* 1987 — Bernstein, P., Hadzilacos, V. e Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley. Texto disponível on-line.

Bershad *et al.* 1990 — Bershad, B., Anderson, T., Lazowska, E. e Levy, H. (1990). Lightweight remote procedure call. *ACM Transactions Computer Systems*, Vol. 8, No. 1, pgs. 37–55.

Bershad *et al.* 1991 — Bershad, B., Anderson, T., Lazowska, E. e Levy, H. (1991). User-level interprocess communication for shared memory multiprocessors. *ACM Transactions Computer Systems*, Vol. 9, No. 2, pgs. 175–198.

Bershad *et al.* 1993 — Bershad, B., Zekauskas, M. e Sawdon, W. (1993). The Midway distributed shared memory system. Em *Proceedings IEEE COMPCON Conference, IEEE*, pgs. 528–37.

Bershad *et al.* 1995 — Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Chambers, C. e Eggers, S. (1995). Safety and performance in the SPIN operating system. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pgs. 267–84.

Bhatti e Friedrich 1999 — Bhatti, N. e Friedrich, R. (1999). *Web Server Support for Tiered Services*. Hewlett-Packard Corporation Technical Report HPL-1999-160.

Birman 1993 — Birman, K.P. (1993). The process group approach to reliable distributed computing. *Comms. ACM*, Vol. 36, No. 12, pgs. 36–53.

Birman 1996 — Birman, K.P. (1996). *Building Secure and Reliable Network Applications*. Greenwich, CT: Manning.

Birman 2004 — Birman, K.P. (2004). Like it or not, Web Services are Distributed Objects! *Comm. of the ACM*. Vol. 47, No. 12, pgs. 60–62. Dezembro.

Birman e Joseph 1987a — Birman, K.P. e Joseph, T.A. (1987). Reliable communication in the presence of failures. *ACM Transactions Computer Systems*, Vol. 5, No. 1, pgs. 47–76.

Birman e Joseph 1987b — Birman, K. e Joseph, T. (1987). Exploiting virtual synchrony in distributed systems. Em *Proceedings 11th ACM Symposium on Operating Systems Principles*, pgs. 123–38.

Birman *et al.* 1991 — Birman, K.P., Schiper, A. e Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Transactions Computer Systems*, Vol. 9, No. 3, pgs. 272–314.

Birrell e Needham 1980 — Birrell, A.D. e Needham, R.M. (1980). A universal file server. *IEEE Transactions Software Engineering*, Vol. SE-6, No. 5, pgs. 450–3.

Birrell e Nelson 1984 — Birrell, A.D. e Nelson, B.J. (1984). Implementing remote procedure calls. *ACM Transactions Computer Systems*, Vol. 2, pgs. 39–59.

Birrell *et al.* 1982 — Birrell, A.D., Levin, R., Needham, R.M. e Schroeder, M.D. (1982). Grapevine: an exercise in distributed computing. *Comms. ACM*, Vol. 25, No. 4, pgs. 260–73.

Birrell *et al.* 1995 — Birrell, A., Nelson, G. e Owicki, S. (1993). Network objects. Em *Proceedings 14th ACM Symposium on Operating Systems Principles*, pgs. 217–30.

Bishop e Warren 2003 — Bishop, P. e Warren, N. (2003). *JavaSpaces in Practice*. Pearson Education.

Bisiani e Forin 1988 — Bisiani, R. e Forin, A. (1988). Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions Computers*, Vol. 37, No. 8, pgs. 930–45.

Bisiani e Ravishankar 1990 — Bisiani, R. e Ravishankar, M. (1990). Plus: a distributed shared memory system. Em *Proceedings 17th International Symposium on Computer Architecture*, pgs. 115–24.

Black 1990 — Black, D. (1990). Scheduling support for concurrency and parallelism in the Mach operating system, *IEEE Computer*, Vol. 23, No. 5, pgs. 35–43.

Black e Artsy 1990 — Black, A. e Artsy, Y. (1990). Implementing location independent invocation, *IEEE Transactions Parallel and Distributed Systems*, Vol. 1, No. 1.

Blair e Stefani 1997 — Blair, G.S. e Stefani, J.-B. (1997). *Open Distributed Processing and Multimedia*. Harlow, Inglaterra: Addison-Wesley.

Blakley 1999 — Blakley, R. (1999). *CORBA Security – An Introduction to Safe Computing with Objects*. Reading, MA: Addison-Wesley.

Bolosky *et al.* 1996 — Bolosky, W., Barrera, J., Draves, R., Fitzgerald, R., Gibson, G., Jones, M., Levi, S., Myhrvold, N. e Rashid, R. (1996). The Tiger video fileserver, *6th NOSSDAV Conference*, Zushi, Japão, Abril.

Bolosky *et al.* 1997 — Bolosky, W., Fitzgerald, R. e Douceur, J. (1997). Distributed schedule management in the Tiger video fileserver, *16th ACM Symposium on Operating System Principles*, pgs. 212–23, St Malo, França, Outubro.

Bolosky *et al.* 2000 — Bolosky, W.J., Douceur, J.R., Ely, D. e Theimer, M. (2000). Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs, em *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2000, pgs. 34–43.

Bonnaire *et al.* 1995 — Bonnaire, X., Baggio, A. e Prun, D. (1995). Intrusion free monitoring: an observation engine for message server based applications. Em *Proceedings of the 10th International Symposium on Computer and Information Sciences* (ISCIS X), pgs. 541–48.

Borisov *et al.* 2001 — Borisov, N., Goldberg, I. e Wagner, D. (2001) Intercepting mobile communications: The insecurity of 802.11. Em *Proceedings of MOBICOM 2001*.

Bowman *et al.* 1990 — Bowman, M., Peterson, L. e Yeatts, A. (1990). Univers: an attribute-based name server. *Software–Practice and Experience*, Vol. 20, No. 4, pgs. 403–24.

Box 1998 — Box, D. (1998). *Essential COM*. Reading, MA: Addison-Wesley.

Box and Curbera 2004 — Box, D. e Curbera, F. (2004). *Web Services Addressing (WS-Addressing)*, BEA Systems, IBM and Microsoft. Agosto.

Boykin *et al.* 1993 — Boykin, J., Kirschen, D., Langerman, A. e LoVerso, S. (1993). *Programming under Mach*. Reading, MA: Addison-Wesley.

Bray e Sturman 2002 — Bray, J. e Sturman. C.F. (2002). *Bluetooth: Connect Without Cables*, 2ª ed., Prentice Hall.

Buford 1994 — Buford, J.K. (1994). *Multimedia Systems*. Addison-Wesley.

Burns e Wellings 1998 — Burns, A. e Wellings, A. (1998). *Concurrency in Ada*, Cambridge University Press.

Burrows *et al.* 1989 — Burrows, M., Abadi, M. e Needham, R. (1989). *A logic of authentication*. Technical Report 39, Palo Alto, CA: Digital Equipment Corporation Systems Research Center.

Burrows *et al.* 1990 — Burrows, M., Abadi, M. e Needham, R. (1990). A logic of authentication. *ACM Transactions Computer Systems*, Vol. 8, pgs. 18–36.

Bush 1945 — Bush, V. (1945). As we may think. *The Atlantic Monthly*, Julho.

Callaghan 1996a — Callaghan, B. (1996). *WebNFS Client Specification*, Internet RFC 2054, Outubro.

Callaghan 1996b — Callaghan, B. (1996). *WebNFS Server Specification*, Internet RFC 2055, Outubro.

Callaghan 1999 — Callaghan, B. (1999). *NFS Illustrated*, Reading, MA: Addison-Wesley.

Callaghan *et al.* 1995 — Callaghan, B., Pawlowski, B. e Staubach, P. (1995). *NFS Version 3 Protocol Specification*, Internet RFC 1813, Sun Microsystems, Junho.

Campbell 1997 — Campbell, R. (1997). *Managing AFS: The Andrew File System*, Prentice-Hall.

Campbell *et al.* 1993 — Campbell, R., Islam, N., Raila, D. e Madany, P. (1993). Designing and implementing Choices: an object-oriented system in C++. *Comms. ACM*, Vol. 36, No. 9, pgs. 117–26.

Canetti e Rabin 1993 — Canetti, R. e Rabin, T. (1993). Fast asynchronous byzantine agreement with optimal resilience. Em *Proceedings 25th ACM Symposium on Theory of Computing*, pgs. 42–51.

Carriero e Gelernter 1989 — Carriero, N. e Gelernter, D. (1989). Linda in context. *Comms. ACM*, Vol. 32, No. 4, pgs. 444–58.

Carter *et al.* 1991 — Carter, J.B., Bennett, J.K. e Zwaenepoel, W. (1991). Implementation and performance of Munin. Em *Proceedings 13th ACM Symposium on Operating System Principles*, pgs. 152–64.

Carter *et al.* 1998 — Carter, J., Ranganathan, A. e Susarla, S. (1998). Khazana, An Infrastructure for Building Distributed Services. Em *Proceedings of ICDCS'98*. Amsterdam, Holanda.

Castro e Liskov 2000 — Castro, M. e Liskov, B. (2000). Proactive Recovery in a Byzantine-Fault-Tolerant System, *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'00)*, San Diego, EUA, Outubro de 2000.

Castro *et al.* 2002 — Castro, M., Druschel, P., Hu, Y.C. e Rowstron, A. (2002). Topology-aware routing in structured peer-to-peer overlay networks. *Technical Report MSR-TR-2002-82*, Microsoft Research, 2002.

Castro *et al.* 2003 — Castro, M., Costa, M. e Rowstron, A. (2003). Performance and dependability of structured peer-to-peer overlays. *Technical Report MSR-TR-2003-94*, Microsoft Research, 2003.

CCITT 1988a — CCITT (1988). *Recommendation X.500: The Directory – Overview of Concepts, Models and Service*. International Telecommunications Union, Place des Nations, 1211 Genebra, Suíça.

CCITT 1988b — CCITT (1988). *Recommendation X.509: The Directory – Authentication Framework. International Telecommunications Union*, Place des Nations, 1211 Genebra, Suíça.

CCITT 1990 — CCITT (1990). *Recommendation I.150: B-ISDN ATM Functional Characteristics. International Telecommunications Union*, Place des Nations, 1211 Genebra, Suíça.

Ceri e Owicki 1982 — Ceri, S. e Owicki, S. (1982). On the use of optimistic methods for concurrency control in distributed databases. Em *Proceedings 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, Berkeley, CA. pgs. 117–30.

Ceri e Pelagatti 1985 — Ceri, S. e Pelagatti, G. (1985). *Distributed Databases – Principles and Systems*. McGraw-Hill.

Chalmers *et al.* 2004 — Chalmers, D., Dulay, N. e Sloman, M. (2004). Meta Data to Support Context Aware Mobile Applications. *Proceedings IEEE Intl. Conference on Mobile Data Management (MDM2004)*, Berkeley, CA, EUA, Janeiro, pgs. 199–210.

Chandra e Toueg 1996 — Chandra, T. e Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, Abr., pgs. 374–82.

Chandy e Lamport 1985 — Chandy, K. e Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pgs. 63–75.

Chang e Maxemchuk 1984 — Chang, J. e Maxemchuk, N. (1984). Reliable Broadcast Protocols, *ACM Transactions on Computer Systems*. Vol. 2. No. 3. pgs. 251–75.

Chang e Roberts 1979 — Chang, E.G. e Roberts, R. (1979). An improved algorithm for decentralized extrema-finding in circular configurations of processors. *Comms. ACM*, Vol. 22, No. 5, pgs. 281–3.

Charron-Bost 1991 — Charron-Bost, B. (1991). Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, Vol. 39, Julho, pgs. 11–16.

Chaum 1981 — Chaum, D. (1981). Untraceable Electronic Mail, Return Addresses and Digital Pseudonyms. *Comms. ACM*, Vol. 24, No. 2, Fevereiro, pgs. 84–88.

Chen *et al.* 1994 — Chen, P., Lee, E., Gibson, G., Katz, R. e Patterson, D. (1994). RAID: High-Performance, Reliable Secondary Storage. ACM Computing Surveys, Vol. 26, No. 2, Junho, pgs. 145–188.

Cheng 1998 — Cheng, C.K. (1998). *A survey of media servers*. Hong Kong University CSIS, Novembro.

Cheriton 1984 — Cheriton, D.R. (1984). The V kernel: a software base for distributed systems. *IEEE Software*, Vol. 1 No. 2, pgs. 19–42.

Cheriton 1985 — Cheriton, D.R. (1985). Preliminary thoughts on problem-oriented shared memory: a decentralized approach to distributed systems. *ACM Operating Systems Review*, Vol. 19, No. 4, pgs. 26–33.

Cheriton 1986 — Cheriton, D.R. (1986). VMTP: A protocol for the next generation of communication systems.Em *Proceedings SIGCOMM'86 Symposium on Communication Architectures and Protocols*, ACM, pgs. 406–15.

Cheriton e Mann 1989 — Cheriton, D. e Mann, T. (1989). Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions Computer Systems*, Vol. 7, No. 2, pgs. 147–83.

Cheriton e Skeen 1993 — Cheriton, D. e Skeen, D. (1993). Understanding the limitations of causally and totally ordered communication. Em *Proceedings 14th ACM Symposium on Operating System Principles*, Dez., pgs. 44–57.

Cheriton e Zwaenepoel 1985 — Cheriton, D.R. e Zwaenepoel, W. (1985). Distributed process groups in the V kernel. *ACM Transactions Computer Systems*, Vol. 3, No. 2, pgs. 77–107.

Cheswick e Bellovin 1994 — Cheswick, E.R. e Bellovin, S.M. (1994). *Firewalls and Internet Security*. Reading, MA: Addison-Wesley.

Chien 2004 — Chien, A. (2004). Massively Distributed Computing: Virtual Screening on a Desktop Grid. In Foster, I. e Kesselman, C. (eds), *The Grid 2*. São Francisco, CA: Morgan Kauffman.

Choudhary *et al.* 1989 — Choudhary, A., Kohler, W., Stankovic, J. e Towsley, D. (1989). A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Transactions Software Engineering*, Vol. 15, No. 1.

Chu *et al.* 2000 — Chu,Y.-H., Rao, S.G. e Zhang, H. (2000). A case for end system multicast. Em *Proc. of ACM Sigmetrics*, Junho de 2000, pgs. 1–12.

Clark 1982 — Clark, D.D. (1982). *Window and Acknowledgement Strategy in TCP*, Internet RFC 813.

Clark 1988 — Clark, D.D. (1988). The Design Philosophy of the DARPA Internet Protocols. *ACM SIGCOMM Computer Communication Review*, Vol. 18, No. 4, Agosto, pgs. 106–114.

Clarke *et al.* 2000 — Clarke, I., Sandberg, O., Wiley, B. e Hong, T. (2000). Freenet: A Distributed Anonymous Information Storage and Retrieval System. Em *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000.

Cohen 2003 — Cohen, B. (2003). Incentives Build Robustness in BitTorrent, Maio de 2003, Internet publication.

Comer 2000a — Comer, D.E. (2000). *Internetworking with TCP/IP, Volume 1: Principles, Protocols and Architecture*, 4ª ed. Upper Saddle River, NJ: Prentice-Hall.

Comer 2000b — Comer, D.E. (2000). *The Internet Book*, 3ª ed. Upper Saddle River, NJ: Prentice-Hall.

Condict *et al.* 1994 — Condict, M., Bolinger, D., McManus, E., Mitchell, D. e Lewontin, S. (1994). *Microkernel modularity with integrated kernel performance*. Technical report, OSF Research Institute, Cambridge, MA, Abril.

Coulouris *et al.* 1998 — Coulouris, G.F., Dollimore, J. e Roberts, M. (1998). Role and task-based access control in the PerDiS groupware platform. *Third ACM Workshop on Role-Based Access Control*, George Mason University, Washington DC, 22–23 de Outubro.

Cristian 1989 — Cristian, F. (1989). Probabilistic clock synchronization. *Distributed Computing*, Vol. 3, pgs. 146–58.

Cristian 1991a — Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Comms. ACM*, Vol. 34, No. 2.

Cristian 1991b — Cristian, F. (1991). Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, Springer Verlag, Vol. 4, pgs. 175–87.

Cristian e Fetzer 1994 — Cristian, F. e Fetzer, C. (1994). Probabilistic Internal Clock Synchronization. Em *Proceedings 13th Symposium on Reliable Distributed Systems*, IEEE Computer Society Press, 25–27 de Outubro, pgs. 22–31.

Crow *et al.* 1997 — Crow, B., Widjaja, I., Kim, J. e Sakai, P. (1997). IEEE 802.11 Wireless local area networks. *IEEE Communications Magazine*, Set. de 1997, pgs. 116–26.

cryptography.org — *North American Cryptography Archives*.

Culler *et al.* 2001 — Culler, D.E., Hill, J., Buonadonna, P., Szewczyk, R. e Woo, A. (2001). A Network-Centric Approach to Embedded Software for Tiny Devices. *Proceedings of the First International Workshop on Embedded Software*, Tahoe City, CA, EUA, Outubro, pgs. 114–130.

Culler *et al.* 2004 — Culler, D., Estrin, D. e Srivastava, M. (2004). Overview of Sensor Networks. *IEEE Computer*, Vol. 37, no. 8, Agosto, pgs. 41–49.

Curtin e Dolske 1998 — Kurtin, M. e Dolski, J. (1998). A brute force search of DES Keyspace. *:login: – the Newsletter of the USENIX Association*, Maio.

Custer 1998 — Custer, H. (1998). *Inside Windows NT*, segunda edição. Microsoft Press.

Czerwinski *et al.* 1999 — Czerwinski, S., Zhao, B., Hodes, T., Joseph, A. e Katz, R. (1999). An architecture for a secure discovery service. Em *Proceedings Fifth Annual International Conference on Mobile Computing and Networks*.

Dabek *et al.* 2001 — Dabek, F., Kaashoek, M.F., Karger, D., Morris, R. e Stoica, I. (2001). Wide-area cooperative storage with CFS. Em *Proc. of the ACM Symposium on Operating System Principles*, Outubro de 2001.

Dabek *et al.* 2003 — Dabek, F., Zhao, B., Druschel, P. e Kubiatowicz, J. (2003). Ion Stoica, Towards a Common API for Structured Peer-to-Peer Overlays, Em *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, Fevereiro de 2003.

Daemen e Rijmen 2000 — Daemen, J. e Rijmen, V. (2000). The Block Cipher Rijndael, Smart Card Research and Applications, LNCS 1820, J.-J. Quisquater e B. Schneier, (eds). Springer-Verlag, 2000, pgs. 288–296.

Daemen e Rijmen 2002 — Daemen, J. e Rijmen, V. (2002). The Design of Rijndael: *AES – The Advanced Encryption Standard*, Springer-Verlag.

Dasgupta *et al.* 1991 — Dasgupta, P., LeBlanc Jr, R.J., Ahamad, M. e Ramachandran, U. (1991). The Clouds distributed operating system. *IEEE Computer*, Vol. 24, No. 11, pgs. 34–44.

Davidson 1984 — Davidson, S.B. (1984). Optimism and consistency in partitioned database systems. *ACM Transactions Database Systems*, Vol. 9, No. 3, pgs. 456–81.

Davidson *et al.* 1985 — Davidson, S.B., Garcia-Molina, H. e Skeen, D. (1985). Consistency in partitioned networks. *Computing Surveys*, Vol. 17, No. 3, pgs. 341–70.

Davies *et al.* 1998 — Davies, N., Friday, A., Wade, S. e Blair, G. (1998). L2imbo: a distributed systems platform for mobile computing. *Mobile Networks and Applications*, Vol. 3, No. 2, Agosto, pgs. 143–156.

de Ipiña *et al.* 2002 — de Ipiña, D.L., Mendonça, P. e Hopper, A. (2002). TRIP: a Low-Cost Vision-Based Location System for Ubiquitous Computing. *Personal and Ubiquitous Computing*, Vol. 6, No. 3, Maio, pgs. 206–219.

Debaty e Caswell 2001 — Debaty, P. e Caswell, D. (2001). Uniform Web Presence Architecture for People, Places, and Things. IEEE Personal Communications, Vol. 8, No. 4, Agosto, pgs. 6–11.

DEC 1990 — Digital Equipment Corporation (1990). *In Memoriam: J. C. R. Licklider 1915–1990*, Technical Report 61, DEC Systems Research Center.

Delgrossi *et al.* 1993 — Delgrossi, L., Halstrick, C., Hehmann, D., Herrtwich, R.G., Krone, O., Sandvoss, J. e Vogt, C. (1993). Media scaling for audiovisual communication with the Heidelberg transport system. *ACM Multimedia'93*, Anaheim, CA.

Demers *et al.* 1989 — Demers, A., Keshav, S. e Shenker, S. (1989). Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM'89*.

Denning e Denning 1977 — Denning, D. e Denning, P. (1977). Certification of programs for secure information flow. *Comms. ACM*, Vol. 20, No. 7, pgs. 504–13.

Dertouzos 1974 — Dertouzos, M.L. (1974). Control robotics – the procedural control of physical processes. *IFIP Congress*.

Dierks and Allen 1999 — Dierks, T. e Allen, C. (1999). *The TLS Protocol Version 1.0*, Internet RFC 2246.

Diffie 1988 — Diffie, W. (1988). The first ten years of public-key cryptography. *Proceedings of the IEEE*, Vol. 76, No. 5, Maio de 1988, pgs. 560–77.

Diffie and Hellman 1976 — Diffie, W. e Hellman, M.E. (1976). New directions in cryptography. *IEEE Transactions Information Theory*, Vol. IT-22, pgs. 644–54.

Diffie e Landau 1998 — Diffie, W. e Landau, S. (1998). *Privacy on the Line*. Cambridge, MA: MIT Press.

Dijkstra 1959 — Dijkstra, E.W. (1959). A note on two problems in connection with graphs. *Numerische Mathematic*, Vol. 1, pgs. 269–71.

Dingledine *et al.* 2000 — Dingledine, R., Freedman, M.J. e Molnar, D. (2000). The Free Haven project: Distributed anonymous storage service. Em *Proc. Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, Julho de 2000.

Dolev e Malki 1996 — Dolev, D. e Malki, D. (1996). The Transis approach to high availability cluster communication. *Comms. ACM*, Vol. 39, No. 4, pgs. 64–70.

Dolev e Strong 1983 — Dolev, D. e Strong, H. (1983). Authenticated algorithms for byzantine agreement. *SIAM Journal of Computing*, Vol. 12, No. 4, pgs. 656–66.

Dolev *et al.* 1986 — Dolev, D., Halpern, J., e Strong, H. (1986). On the possibility and impossibility of achieving clock synchronization. *Journal of Computing Systems Science*, 32, 2 (Abr.), pgs. 230–50.

Dorcey 1995 — Dorcey, T. (1995). CU-SeeMe Desktop Video Conferencing Software, *Connexions*, vol. 9, no. 3 (Março).

Douceur e Bolosky 1999 — Douceur, J.R. e Bolosky, W. (1999). Improving responsiveness of a stripe-scheduled media server. *SPIE Proceedings*, Vol. 3654. Multimedia Computing and Networking, pgs. 192–203.

Douglis e Ousterhout 1991 — Douglis, F. e Ousterhout, J. (1991). Transparent process migration: design alternatives and the Sprite implementation, *Software – Practice and Experience*, Vol. 21, No. 8, pgs. 757–89.

Draves 1990 — Draves, R. (1990). A revised IPC interface, Em *Proceedings USENIX Mach Workshop*, pgs. 101–21, Outubro.

Draves *et al.* 1989 — Draves, R.P., Jones, M.B. e Thompson, M.R. (1989). *MIG - the Mach Interface Generator*. Technical Report, Dept. of Computer Science, Carnegie-Mellon University.

Druschel e Peterson 1993 — Druschel, P. e Peterson, L. (1993). Fbufs: a high-bandwidth cross-domain transfer facility. Em *Proceedings 14th ACM Symposium on Operating System Principles*, pgs. 189–202.

Druschel e Rowstron 2001 — Druschel, P. e Rowstron, A. (2001). PAST: A large-scale, persistent peer-to-peer storage utility. Em *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. Schloss Elmau, Alemanha, Maio de 2001.

Dubois *et al.* 1988 — Dubois, M., Scheurich, C. e Briggs, F.A. (1988). Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer*, Vol. 21, No. 2, pgs. 9–21.

Dwork *et al.* 1988	Dwork, C., Lynch, N. e Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, Vol. 35, No. 2, pgs. 288–323.
Eager *et al.* 1986	Eager, D., Lazowska, E. e Zahorjan, J. (1986). Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 5, pgs. 662–675.
Edney e Arbaugh 2003	Edney, J. e Arbaugh, W. (2003). *Real 802.11 Security: Wi-Fi Protected*. Pearson Education.
Edwards e Grinter 2001	Edwards, W.K. e Grinter, R. (2001). At Home with Ubiquitous Computing: Seven Challenges. *Proceedings Third International Conference on Ubiquitous Computing (Ubicomp 2001)*, Atlanta, GA, EUA, Set.–Out., Springer-Verlag, pgs. 256–272.
Edwards *et al.* 2002	Edwards, W.K., Newman, M.W., Sedivy, J.Z., Smith, T.F. e Izadi, S. (2002). Challenge: Recombinant Computing and the Speakeasy Approach. *Proceedings of the Eighth ACM International Conference on Mobile Computing and Networking (MobiCom 2002)*, Atlanta, GA. Setembro, pgs. 279–286.
EFF 1998	Electronic Frontier Foundation (1998). *Cracking DES, Secrets of Encryption Research, Wiretap Politics & Chip Design*. Sebastapol, CA: O'Reilly & Associates.
Egevang e Francis 1994	Egevang, K. e Francis, P. (1994). *The IP Network Address Translator (NAT)*. Internet RFC 1631, Maio.
Eisler *et al.* 1997	Eisler, M., Chiu, A. e Ling, L. (1997). *RPCSEC_GSS Protocol Specification*. Sun Microsystems. Internet RFC 2203. Setembro.
El Abbadi *et al.* 1985	El Abbadi, A., Skeen, D. e Cristian, C. (1985). An efficient fault-tolerant protocol for replicated data management. Em *4th Annual ACM SIGACT/SIGMOD Symposium on Principles of Data Base Systems*, Portland, OR.
Ellis *et al.* 1991	Ellis, C., Gibbs, S. e Rein, G. (1991). Groupware – some issues and experiences. *Comms. ACM*, Vol. 34, No. 1, pgs. 38–58.
Ellison 1996	Ellison, C. (1996). Establishing identity without certification authorities. Em *6th USENIX Security Symposium*, San Jose, 22–25 de Julho.
Ellison *et al.* 1999	Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B. e Ylonen, T. (1999). *SPKI Certificate Theory*. Internet RFC 2693, Setembro.
Elrad *et al.* 2001	Elrad, T., Filman, R. e Bader A. (eds) (2001). Theme Section on Aspect-Oriented Programming, *Comms. ACM*, Vol. 44, No. 10.
Evans *et al.* 2003	Evans, C. e 15 outros autores (2003). *Web Services Reliability (WS-Reliability)*, Fujitsu, Hitachi, NEC, Oracle Corporation, Sonic Software e Sun Microsystems. Janeiro.
Fall 2003	Fall, K. (2003). A delay-tolerant network architecture for challenged internets. *Proceedings of the ACM 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM 2003)*, Karlsruhe, Alemanha, Agosto, pgs. 27–34.
Farley 1998	Farley, J. (1998). *Java Distributed Computing*. Cambridge, MA: O'Reilly.
Farrow 2000	Farrow, R. (2000). Distributed denial of service attacks – how Amazon, Yahoo, eBay and others were brought down. *Network Magazine*, Abril.
Ferguson e Schneier 2003	Ferguson, N. e Schneier, B. (2003). *Practical Cryptography*, John Wiley & Sons.
Ferrari e Verma 1990	Ferrari, D. e Verma, D. (1990). A scheme for real-time channel establishment in wide-area networks, *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 4.
Ferreira *et al.* 2000	Ferreira, P., Shapiro, M., Blondel, X., Fambon, O., Garcia, J., Kloostermann, S., Richer, N., Roberts, M., Sandakly, F., Coulouris, G., Dollimore, J., Guedes, P., Hagimont, D. e Krakowiak, S. (2000). PerDiS: Design, Implementation, and Use of a PERsistent DIstributed Store. Em *LNCS 1752: Advances in Distributed Systems*. Berlim, Heidelberg, Nova York: Springer-Verlag, pgs. 427–53.
Ferris and Langworthy 2004	Ferris, C. e Langworthy, D. (eds), Bilorusets, R. e outros 22 autores (2004). *Web Services Reliable Messaging Protocol (WS-Reliable Messaging)*. BEA, IBM, Microsoft e TibCo. Março.
Fidge 1991	Fidge, C. (1991). Logical Time in Distributed Computing Systems. *IEEE Computer*, Vol. 24, No. 8, pgs. 28–33.

Fielding 2000 — Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*, PhD. Dissertation.

Fielding *et al.* 1999 — Fielding, R., Gettys, J., Mogul, J.C., Frystyk, H., Masinter, L. , Leach, P. e Berners-Lee, T. (1999). *Hypertext Transfer Protocol – HTTP/1.1*. Internet RFC 2616.

Fischer 1983 — Fischer, M. (1983). The Consensus Problem in Unreliable Distributed Systems (a Brief Survey). Em Karpinsky, M. (ed.), *Foundations of Computation Theory*, Vol. 158 de *Lecture Notes in Computer Science*, Springer-Verlag, pgs. 127–140. Yale University Technical Report YALEU/DCS/TR-273.

Fischer e Lynch 1982 — Fischer, M. e Lynch, N. (1982). A lower bound for the time to assure interactive consistency. *Inf. Process. Letters*, Vol. 14, No. 4, Junho, pgs. 183–6.

Fischer e Michael 1982 — Fischer, M.J. e Michael, A. (1982). Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. Em *Proceedings Symposium on Principles of Database Systems*, ACM, pgs. 70–5.

Fischer *et al.* 1985 — Fischer, M., Lynch, N. e Paterson, M. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, Vol. 32, No. 2, Abr., pgs. 374–82.

Fitzgerald e Rashid 1986 — Fitzgerald, R. e Rashid, R.F. (1986). The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions Computer Systems*, Vol. 4, No. 2, pgs. 147–77.

Flanagan 2002 — Flanagan, D. (2002). *Java in a Nutshell*. 4ª ed. Cambridge, Inglaterra: O'Reilly.

Fleisch e Popek 1989 — Fleisch, B. e Popek, G. (1989). Mirage: a coherent distributed shared memory design. Em *Proceedings 12th ACM Symposium on Operating System Principles*, Dezembro, pgs. 211–23.

Floyd 1986 — Floyd, R. (1986). *Short term file reference patterns in a UNIX environment*. Technical Rep. TR 177, Rochester, NY: Dept of Computer Science, University of Rochester.

Floyd e Jacobson 1993 — Floyd, S. e Jacobson, V. (1993). The Synchronization of Periodic Routing Messages. *ACM Sigcomm'93 Symposium*.

Floyd *et al.* 1997 — Floyd, S., Jacobson, V., Liu, C., McCanne, S. e Zhang, L. (1997). A Reliable Multicast Framework for Lightweight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, Vol. 5, No. 6, pgs. 784–803.

Fluhrer *et al.* 2001 — Fluhrer, S., Mantin, I. e Shamir, A. (2001). Weaknesses in the Key Scheduling Algorithm of RC4, em *Proceedings of the 8th annual workshop on Selected Areas of Cryptography (SAC)*, Toronto, Canadá.

Ford e Fulkerson 1962 — Ford, L.R. e Fulkerson, D.R. (1962). *Flows in Networks*. Princeton, NJ: Princeton University Press.

Foster e Kesselman 2004 — Foster, I. e Kesselman, C. (eds) (2004). *The Grid 2*. São Francisco, CA: Morgan Kauffman.

Foster *et al.* 2001 — Foster, I., Kesselman, C. e Tuecke, S. (2001). The Anatomy of the Grid: enabling scalable virtual organizations. *Intl. J. Supercomputer Applications*, Vol. 15, No. 3, pgs. 200–222.

Foster *et al.* 2002 — Foster, I., Kesselman, C., Nick, J. e Tuecke, S. (2002). Grid services for distributed systems integration. *IEEE Computer*, Vol. 35, No. 6, pgs. 37–46.

Foster *et al.* 2004 — Foster, I., Kesselman, C. e Tuecke, S. (2004). *The Open Grid Services Architecture*. Em Foster, I. e Kesselman, C. (eds), *The Grid 2*. São Francisco, CA: Morgan Kauffman.

Fox *et al.* 1997 — Fox, A., Gribble, S., Chawathe, Y., Brewer, E. e Gauthier, P. (1997). Cluster-based scalable network services. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pgs. 78–91.

Fox *et al.* 1998 — Fox, A., Gribble, S.D., Chawathe, Y. e Brewer, E.A. (1998). Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives. *IEEE Personal Communications*, Vol. 5, No. 4, Agosto, pgs. 10–19.

Fox *et al.* 2003 — Fox, D., Hightower, J., Liao, L., Schulz, D. e Borriello, G. (2003). Bayesian Filtering for Location Estimation. *IEEE Pervasive Computing*, Vol. 2, No. 3, Julho–Setembro, pgs. 24–33.

Freed and Borenstein 1996 — Freed, N. e Borenstein, N. (1996). *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. Setembro. Internet RFC 1521.

Freenet 2004 — The Free Network Project (2004).

FreePastry project 2004 — FreePastry project (2004). Pastry: A substrate for peer-to-peer applications.

Garay e Moses 1993 — Garay, J. e Moses, Y. (1993). Fully polynomial Byzantine agreement in $t+1$ rounds. Em *Proceedings 25th ACM symposium on theory of computing*, ACM Press, pgs. 31–41, Maio.

Garcia-Molina 1982 — Garcia-Molina, H. (1982). Elections in Distributed Computer Systems. *IEEE Transactions on Computers*, Vol. C-31, No. 1, pgs. 48–59.

Garcia-Molina e Spauster 1991 — Garcia-Molina, H. e Spauster, A. (1991). Ordered and Reliable Multicast Communication. *ACM Transactions Computer Systems*, Vol. 9, No. 3, pgs. 242–71.

Garfinkel 1994 — Garfinkel, S. (1994). *PGP: Pretty Good Privacy*. O'Reilly.

Gehrke e Madden 2004 — Gehrke, J. e Madden, S. (2004). Query processing in sensor networks. *IEEE Pervasive Computing*, Vol. 3, No. 1, Jan–Mar, pgs. 46–55.

Gharachorloo *et al.* 1990 — Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. e Hennessy, J. (1990). Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. Em *Proceedings 17th. Annual International Symposium on Computer Architecture*, Maio, pgs. 15–26.

Gibbs e Tsichritzis 1994 — Gibbs, S.J. e Tsichritzis, D.C. (1994). *Multimedia Programming*. Addison-Wesley.

Gibson *et al.* 2004 — Gibson, G., Courtial, J., Padgett, M.J., Vasnetsov, M., Pasko, V., Barnett, S.M. e Franke-Arnold, S. (2004). Free-space information transfer using light beams carrying orbital angular momentum. *Optics Express*, Vol. 12, No. 22, Novembro, pgs. 5448–5456.

Gifford 1979 — Gifford, D.K. (1979). Weighted voting for replicated data. Em *Proceedings 7th Symposium on Operating Systems Principles*, ACM, pgs. 150–62.

Globus 2004 — The Globus Alliance (2004). *WS-Resource framework*.

Gokhale e Schmidt 1996 — Gokhale, A. e Schmidt, D. (1996). Measuring the Performance of Communication Middleware on High-Speed Networks. *Proceedings of SIGCOMM'96*, ACM, pgs. 306–17.

Golding e Long 1993 — Golding, R. e Long, D. (1993). *Modeling replica divergence in a weak-consistency protocol for global-scale distributed databases*. Technical report UCSC-CRL-93-09, Computer and Information Sciences Board, University of California, Santa Cruz.

Goldschlag *et al.* 1999 — Goldschlag, D., Reed, M. e Syverson, P. (1999). Onion routing for anonymous and private internet connections. *Communications of the ACM*, Vol. 42, No. 2, pgs. 39–41.

Golub *et al.* 1990 — Golub, D., Dean, R., Forin, A. e Rashid, R. (1990). *UNIX as an application program. Proc. USENIX Summer Conference*, pgs. 87–96.

Gong 1989 — Gong, L. (1989). A Secure Identity-Based Capability System. Em *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, Maio, pgs. 56–63.

Goodman 1989 — Goodman, J. (1989). *Cache Consistency and Sequential Consistency*. Technical Report 61, SCI Committee.

Gordon 1984 — Gordon, J. (1984). *The Story of Alice and Bob*.

Govindan e Anderson 1991 — Govindan, R. e Anderson, D.P. (1991). Scheduling and IPC Mechanisms for Continuous Media. *ACM Operating Systems Review*, Vol. 25, No. 5, pgs. 68–80.

Goyal e Carter 2004 — Goyal, S. e Carter, J. (2004). A Lightweight Secure Cyber Foraging Infrastructure for Resource-Constrained Devices. Proceedings Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2004), Dezembro, pgs. 186–195.

Graumann *et al.* 2003 — Graumann, D., Lara, W., Hightower, J. e Borriello, G. (2003). Real-world implementation of the Location Stack: The Universal Location Framework. *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2003)*, Monterey, CA, EUA, Outubro, pgs. 122–128.

Gray 1978 — Gray, J. (1978). Notes on database operating systems. Em *Operating Systems: an Advanced Course. Lecture Notes in Computer Science*, Vol. 60, pgs. 394–481, Springer-Verlag.

Gray and Szalay 2002 — Gray, J. e Szalay, A. (2002). *The World-Wide Telescope, an Archetype for Online Science*. Technical Report. MSR-TR-2002-75. Microsoft Research.

Greenfield and Dornan 2004 — Greenfield, D. e Dornan, A. (2004). *Amazon: Web Site to Web Services*, Network Magazine. Outubro.

Grimm 2004 — Grimm, R. (2004). One.world: Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, Vol. 3, No. 3, Jul.–Set., pgs. 22–30.

Gruteser e Grunwald 2003 — Gruteser, M. e Grunwald, D. (2003). Enhancing location privacy in wireless LAN through disposable interface identifiers: a quantitative analysis. *Proceedings of the 1st ACM international workshop on Wireless mobile applications and services on WLAN hotspots (WMASH'03)*, San Diego, CA, EUA, Setembro, pgs. 46–55.

Guerraoui *et al*. 1998 — Guerraoui, R., Felber, P., Garbinato, B. e Mazouni, K. (1998). System support for object groups. Em *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications* (OOPSLA'98).

Gummadi *et al*. 2003 — Gummadi, K.P., Gummadi, R., Gribble, S.D., Ratnasamy, S., Shenker, S. e Stoica, I. (2003). The impact of dht routing geometry on resilience and proximity. Em *ACM SIGCOMM 2003*.

Gusella e Zatti 1989 — Gusella, R. e Zatti, S. (1989). The accuracy of clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions Software Engineering*, Vol. 15, pgs. 847–53.

Guttman 1999 — Guttman, E. (1999). Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing*, Vol. 3, No. 4, pgs. 71–80.

Haartsen *et al*. 1998 — Haartsen, J., Naghshineh, M., Inouye, J., Joeressen, O.J. e Allen, W. (1998). Bluetooth: Vision, Goals, and Architecture, *ACM Mobile Computing and Communications Review*, Out., vol. 2, no. 4, pgs. 38–45.

Hadzilacos e Toueg 1994 — Hadzilacos, V. e Toueg, S. (1994). *A Modular Approach to Fault-tolerant Broadcasts and Related Problems*, Technical report, Dept of Computer Science, University of Toronto.

Handley 1998 — Handley, M. (1998). Session Directories and Scalable Internet Multicast Address Allocation. *Proceedings of ACM SIGCOMM*, Setembro.

Handley *et al*. 1999 — Handley, M., Schulzrinne, H., Schooler, E., Rosenberg, J. (1999). *SIP: Session Initiation Protocol*, Internet RFC 2543.

Harbison 1992 — Harbison, S. P. (1992). *Modula-3*. Prentice Hall.

Härder 1984 — Härder, T. (1984). Observations on Optimistic Concurrency Control Schemes. *Information Systems*, Vol. 9, No. 2, pgs. 111–20.

Härder e Reuter 1983 — Härder, T. e Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *Computing Surveys*, Vol. 15, No. 4.

Harrenstien *et al*. 1985 — Harrenstien, K., Stahl, M. e Feinler, E. (1985). *DOD Internet Host Table Specification*. Internet RFC 952.

Harter e Hopper 1994 — Harter, A. e Hopper, A. (1994). A distributed location system for the active office. *IEEE Network*, Vol. 8, No. 1, Janeiro/Fevereiro de 1994, pgs. 62–70.

Harter *et al*. 2002 — Harter, A., Hopper, A., Steggles, P., Ward, A. e Webster, P. (2002). The Anatomy of a Context-Aware Application. Wireless Networks, Vol. 8, No. 2–3, Mar–Maio, pgs. 187–197.

Härtig *et al*. 1997 — Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., e Wolter, J. (1997). The performance of kernel-based systems. Em *Proceedings 16th ACM Symposium on Operating System Principles*, pgs. 66–77.

Hartman e Ousterhout 1995 — Hartman, J. e Ousterhout, J. (1995). The Zebra Striped Network File System. *ACM Trans. on Computer Systems*, Vol. 13 , No. 3, Agosto, pgs. 274–310.

Hauch e Reiser 2000 — Hauch, R. e Reiser, H. (2000). Monitoring Quality of Service across Organizational Boundaries. Em *Trends in Distributed Systems: Towards a Universal Service Market*. Proc. third Intl. IFIP/HGI Working conference, USM. Setembro.

Hayton *et al.* 1998 — Hayton, R., Bacon, J. e Moody, K. (1998). OASIS: Access Control in an Open, Distributed Environment. Em *Proceedings IEEE Symposium on Security and Privacy*, Oakland, CA, pgs. 3–14, Maio.

Hedrick 1988 — Hedrick, R. (1988). *Routing Information Protocol*, Internet RFC 1058.

Heidemann *et al.* 2001 — Heidemann, J., Silva, F., Intanagonwiwat, C., Govindan, R., Estrin, D. e Ganesan, D. (2001). Building efficient wireless sensor networks with low-level naming. *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Alberta, Canadá, Outubro, pgs. 146–159.

Henning 1998 — Henning, M. (1998). Binding, Migration and Scalability in CORBA, *Comms. ACM*, Outubro, Vol. 41, No. 10. pgs. 62–71.

Henning e Vinoski 1999 — Henning, M. e Vinoski, S. (1999). *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley.

Herlihy 1986 — Herlihy, M. (1986). A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions Computer Systems*, Vol. 4, No. 1, pgs. 32–53.

Herlihy e Wing 1990 — Herlihy, M. e Wing, J. (1990). On Linearizability: a correctness condition for concurrent objects. *ACM Transactions on programming languages and systems*, Vol. 12, No. 3, (Julho), pgs. 463–92.

Herrtwich 1995 — Herrtwich, R.G. (1995). Achieving Quality of Service for Multimedia Applications. *ERSADS'95, European Research Seminar on Advanced Distributed Systems*, l'Alpe d'Huez, França, Abril.

Hightower e Borriello 2001 — Hightower, J. e Borriello, G. (2001). Location Systems for Ubiquitous Computing. *IEEE Computer*, Vol. 34, No. 8, Agosto, pgs. 57–66.

Hightower *et al.* 2002 — Hightower, J., Brumitt, B. e Borriello, G. (2002). The Location Stack: A Layered Model for Location in Ubiquitous Computing. *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2002)*, Callicoon, NY, EUA, Junho, pgs. 22–28.

Hill *et al.* 2000 — Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. e Pister, K. (2000). System architecture directions for networked sensors. *Proceedings of the ninth ACM international conference on Architectural support for programming languages and operating systems (ASPLOS-IX)*, Cambridge, MA, EUA, Novembro, pgs. 93–104.

Hirsch 1997 — Hirsch, F.J. (1997). Introducing SSL and Certificates using SSLeay. *World Wide Web Journal*, Vol. 2, No. 3, Summer.

Holmquist *et al.* 2001 — Holmquist, L.E., Mattern, F., Schiele, B., Alahuhta, P., Beigl, M. e Gellersen, H.-W. (2001). Smart-Its Friends: A Technique for Users to Easily Establish Connections between Smart Artefacts. *Proceedings Third International Conference on Ubiquitous Computing (Ubicomp 2001)*, Atlanta, GA, EUA, 30 de Setembro a 2 de Outubro, Springer-Verlag, pgs. 116–122.

Housley 2002 — Housley, R. (2002). *Cryptographic Message Syntax (CMS) Algorithms*. Internet RFC 3370.

Howard *et al.* 1988 — Howard, J.H., Kazar, M.L., Menees, S.G, Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N. e West, M.J. (1988). Scale and Performance in a Distributed File System. *ACM Transactions Computer Systems*, Vol. 6, No. 1, pgs. 51–81.

Huang *et al.* 2000 — Huang, A., Ling, B., Barton, J. e Fox, A. (2000). Running the Web backwards: appliance data services. *Proceedings 9th international World Wide Web conference*.

Huitema 1998 — Huitema, C. (1998). *IPv6 – the New Internet Protocol*. Upper Saddle River, NJ: Prentice-Hall.

Huitema 2000 — Huitema, C. (2000). *Routing in the Internet*, 2ª ed., Englewood Cliffs, NJ: Prentice-Hall.

Hull *et al.* 2004 — Hull, R., Clayton, B. e Melamad, T. (2004). Rapid Authoring of Mediascapes. *Proceedings Sixth International Conference on Ubiquitous Computing (Ubicomp 2004)*, Nottingham, Inglaterra, Setembro, Springer-Verlag, pgs. 125–142.

Hunter e Crawford 1998 — Hunter, J. e Crawford, W. (1998). *Java Servlet Programming*. O'Reilly.

Hutchinson e Peterson 1991 — Hutchinson, N. e Peterson, L. (1991). The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, Vol. 17, No. 1, pgs. 64–76.

Hutchinson *et al.* 1989 — Hutchinson, N.C., Peterson, L.L., Abbott, M.B. e O'Malley, S.W. (1989). RPC in the x-Kernel: Evaluating New Design Techniques. Em *Proc. 12th ACM Symposium on Operating System Principles*, pgs. 91–101.

Hutto e Ahamad 1990 — Hutto, P. e Ahamad, M. (1990). Slow memory: weakening consistency to enhance concurrency in distributed shared memories. Em *Proceedings 10th International Conference on Distributed Computer Systems*, IEEE, pgs. 302–11.

Hyman *et al.* 1991 — Hyman, J., Lazar, A.A. e Pacifici, G. (1991). MARS – The MAGNET-II Real-Time Scheduling Algorithm. *ACM SIGCOM'91*, Zurique.

IEEE 1985a — Institute of Electrical and Electronic Engineers (1985). *Local Area Network – CSMA/CD Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.3, IEEE Computer Society.

IEEE 1985b — Institute of Electrical and Electronic Engineers (1985). *Local Area Network – Token Bus Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.4, IEEE Computer Society.

IEEE 1985c — Institute of Electrical and Electronic Engineers (1985). *Local Area Network – Token Ring Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.5, IEEE Computer Society.

IEEE 1990 — Institute of Electrical and Electronic Engineers (1990). *IEEE Standard 802: Overview and Architecture*. American National Standard ANSI/IEEE 802, IEEE Computer Society.

IEEE 1994 — Institute of Electrical and Electronic Engineers (1994). *Local and metropolitan area networks – Part 6: Distributed Queue Dual Bus (DQDB) access method and physical layer specifications*. American National Standard ANSI/IEEE 802.6, IEEE Computer Society.

IEEE 1999 — Institute of Electrical and Electronic Engineers (1999). *Local and metropolitan area networks – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, American National Standard ANSI/IEEE 802.11, IEEE Computer Society.

IEEE 2002 — Institute of Electrical and Electronic Engineers (2002). *Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs)*, American National Standard ANSI/IEEE 802.15.1, IEEE Computer Society.

IEEE 2003 — Institute of Electrical and Electronic Engineers (2003). *Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*, American National Standard ANSI/IEEE 802.15.4, IEEE Computer Society.

IEEE 2004a — Institute of Electrical and Electronic Engineers (2004). IEEE Standard for Local and Metropolitan Area Networks – Part 16: *Air Interface for Fixed Broadband Wireless Access Systems*. American National Standard ANSI/IEEE 802.16, IEEE Computer Society.

IEEE 2004b — Institute of Electrical and Electronic Engineers (2004). *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Medium Access Control (MAC) Security Enhancement*, American National Standard ANSI/IEEE 802.11i, IEEE Computer Society.

Iftode *et al.* 1996 — Iftode, L., Singh, J. e Li, K. (1996). Scope consistency: a bridge between release consistency and entry consistency. Em *Proceedings 8th annual ACM symposium on Parallel algorithms and architectures*. pgs. 277–87.

Imielinski e Navas 1999 — Imielinski, T. e Navas, J.C. (1999). GPS-based geographic addressing, routing, and resource discovery. *Comms. ACM*, Vol. 42, No. 4, pgs. 86–92.

international pgp — *The International PGP Home Page*.

Internet World Stats 2004 — Internet World Stats.

Ishii e Ullmer 1997 — Ishii, H. e Ullmer, B., (1997). Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI'97)*, Atlanta, GA, EUA, Março, pgs. 234–241.

ISO 1992 — International Organization for Standardization (1992). *Basic Reference Model of Open Distributed Processing, Part 1: Overview and guide to use*. ISO/IEC JTC1/SC212/WG7 CD 10746-1, International Organization for Standardization.

ISO 8879 — International Organization for Standardization (1986). Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML), 1986.

ITU/ISO 1997 — ITU/ISO (1997). Recommendation X.500 (08/97): *Open Systems Interconnection – The Directory: Overview of concepts, models and services*. International Telecommunication Union.

Iyer *et al*. 2002 — Iyer, S., Rowstron, A. e Druschel, P. (2002). Squirrel: A decentralized peer-to-peer web cache. Em *12th ACM Symposium on Principles of Distributed Computing (PODC 2002)*, Julho.

java.sun.com I — Sun Microsystems. *Java Remote Method Invocation*.

java.sun.com II — Sun Microsystems. *Java Object Serialization Specification*.

java.sun.com III — Sun Microsystems. *Servlet Tutorial*.

java.sun.com IV — Jordan, M. e Atkinson, M. (1999). *Orthogonal Persistence for the Java Platform - Draft Specification*. Sun Microsystems Laboratories. Palo Alto, CA.

java.sun.com V — Sun Microsystems, *Java Security API*.

java.sun.com VI — Sun Microsystems (1999). *JavaSpaces technology*

java.sun.com VII — Sun Microsystems. *The Java Web Services Tutorial*.

java.sun.com VIII — Sun Microsystems (2003). *Java Data Objects (JDO)*.

java.sun.com IX — Sun Microsystems. *Java Remote Object Activation Tutorial*.

Johanson e Fox 2004 — Johanson, B. e Fox, A. (2004). Extending Tuplespaces For Coordination in Interactive Workspaces.*Journal of Systems and Software*, Vol. 69, No. 3, Janeiro, pgs. 243–266.

Johnson e Zwaenepoel 1993 — Johnson, D. e Zwaenepoel, W. (1993). The Peregrine High-performance RPC System. *Software–Practice and Experience*, Vol. 23, No. 2, pgs. 201–21.

Jordan 1996 — Jordan, M. (1996). Early Experiences with Persistent Java. Em *Proceedings first international workshop on persistrence and Java*. Glasgow, Escócia.

Joseph *et al*. 1997 — Joseph, A., Tauber, J. e Kaashoek, M. (1997). Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, Vol. 46, No. 3, pgs. 337–52.

Jul *et al*. 1988 — Jul, E., Levy, H., Hutchinson, N. e Black, A. (1988). Fine-grained Mobility in the Emerald System. *ACM Transactions Computer Systems*, Vol. 6, No. 1, pgs. 109–33.

Kaashoek e Tanenbaum 1991 — Kaashoek, F. e Tanenbaum, A. (1991). Group Communication in the Amoeba Distributed Operating System. Em *Proceedings 11th International Conference on Distributed Computer Systems*, pgs. 222–30.

Kaashoek *et al*. 1989 — Kaashoek, F., Tanenbaum, A., Flynn Hummel, S. e Bal, H. (1989). An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, Vol. 23, No. 4, pgs. 5–20.

Kaashoek *et al*. 1997 — Kaashoek, M., Engler, D., Ganzer, G., Briceño, H., Hunt, R., Mazi_res, D., Pinckney, T., Grimm, R., Jannotti, J. e Mackenzie, K. (1997). Application performance and flexibility on exokernel systems. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pgs. 52–65.

Kahn 1967 — Kahn, D. (1967). *The Codebreakers: The Story of Secret Writing*. Nova York: Macmillan.

Kahn 1983 — Kahn, D. (1983). *Kahn on Codes*. Nova York: Macmillan.

Kahn 1991 — Kahn, D. (1991). *Seizing the Enigma*. Boston: Houghton Mifflin.

Kaler 2002 — Kaler, C. (ed.) (2002). *Specification: Web Services Security (WS-Security)*.

Kaliski and Staddon 1998 — Kaliski, B. e Staddon, J. (1998). *RSA Cryptography Specifications*, Version 2.0. Internet RFC 2437.

Kantor and Lapsley 1986 — Kantor, B. e Lapsley, P. (1986). *Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News*, Internet RFC 977, Fevereiro.

Kehne *et al*. 1992 — Kehne, A., Schonwalder, J. e Langendorfer, H. (1992). A Nonce-based Protocol for Multiple Authentications. *ACM Operating Systems Review*, Vol. 26, No. 4, pgs. 84–9.

Keith e Wittle 1993 — Keith, B.E. e Wittle, M. (1993). LADDIS: The Next Generation in NFS File Server Benchmarking, *Summer USENIX Conference Proceedings*. USENIX Association, Berkeley, CA, Junho.

Keleher *et al*. 1992 — Keleher, P., Cox, A. e Zwaenepoel, W. (1992). Lazy consistency for software distributed shared memory. Em *Proceedings 19th Annual International Symposium on Computer Architecture*. pgs. 13–21, Maio de 1992.

Kessler e Livny 1989 — Kessler, R.E. e Livny, M. (1989). An Analysis of Distributed Shared Memory Algorithms, Em *Proceedings 9th International Conference Distributed Computing Systems*. IEEE, pgs. 98–104.

Kiciman e Fox 2000 — Kiciman, E. e Fox, A. (2000). Using Dynamic Mediation to Integrate COTS Entities in a Ubiquitous Computing Environment. *Proceedings Second International Symposium on Handheld and Ubiquitous Computing (HUC2K)*, Bristol, Inglaterra, Setembro, pgs. 211–226.

Kille 1992 — Kille, S. (1992). *Implementing X.400 and X.500: The PP and QUIPU Systems*. Artech House.

Kindberg 1995 — Kindberg, T. (1995). A Sequencing Service for Group Communication (abstract), Em *Proceedings 14th annual ACM Symposium on Principles of Distributed Computing*, p. 260. Technical Report No. 698, Queen Mary e Westfield College Dept. of CS, 1995.

Kindberg 2002 — Kindberg, T. (2002). Implementing Physical Hyperlinks Using Ubiquitous Identifier Resolution. *Proceedings Eleventh International World Wide Web Conference (WWW2002)*, Honolulu, HI, EUA, Maio pgs. 191–199.

Kindberg e Barton 2001 — Kindberg, T. e Barton, J. (2001). A Web-Based Nomadic Computing System. *Computer Networks*, Vol. 35, No. 4, pgs. 443–456.

Kindberg e Fox 2001 — Kindberg, T. e Fox, A. (2001). System Software for Ubiquitous Computing. *IEEE Pervasive Computing*, Vol. 1, No. 1, Jan.–Mar., pgs. 70–81.

Kindberg e Zhang 2003a — Kindberg, T. e Zhang, K. (2003). Secure Spontaneous Device Association. *Proceedings Fifth International Conference on Ubiquitous Computing (Ubicomp 2003)*, Seattle, WA, EUA, Outubro, pgs. 124–131.

Kindberg e Zhang 2003b — Kindberg, T. e Zhang, K. (2003). Validating and Securing Spontaneous Associations between Wireless Devices. *Proceedings 6th Information Security Conference (ISC'03)*, Bristol, Inglaterra, Outubro, Springer-Verlag, pgs. 44–53.

Kindberg *et al*. 1996 — Kindberg, T., Coulouris, G., Dollimore, J. e Heikkinen, J. (1996). Sharing objects over the Internet: the Mushroom approach. Em *Proceedings IEEE Global Internet 1996*, Londres, Nov., pgs. 67–71.

Kindberg *et al*. 2002a — Kindberg, T., Barton, J., Morgan, J., Becker, G., Bedner, I., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Pering, C., Schettino, J. e Serra, B. (2002). People, Places, Things: Web Presence for the Real World. *Mobile Networks and Applications (MONET)*, Vol. 7, No. 5, Outubro, pgs. 365–376.

Kindberg *et al*. 2002b — Kindberg, T., Zhang, K. e Shanka, N. (2002). Context Authentication Using Constrained Channels. *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2002)*, Callicoon, NY, EUA, Junho, pgs. 14–21.

Kistler e Satyanarayanan 1992 — Kistler, J.J. e Satyanarayanan, M. (1992). Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pgs. 3–25.

Kleinrock 1961 — Kleinrock, L. (1961). *Information Flow in Large Communication Networks*, MIT, RLE Quarterly Progress Report, Julho.

Kleinrock 1997 — Kleinrock, L. (1997). Nomadicity: anytime, anywhere in a disconnected world. *Mobile Networks and Applications*, Vol. 1, No. 4, pgs. 351–7.

Kohl and Neuman 1993 — Kohl, J. e Neuman, C. (1993). *The Kerberos Network Authentication Service (V5)*, Internet RFC 1510, Setembro.

Konstantas *et al.* 1997 — Konstantas, D., Orlarey, Y., Gibbs, S. e Carbonel, O. (1997). Distributed Music Rehearsal. Em *Proceedings International Computer Music Conference* 97.

Kopetz e Verissimo 1993 — Kopetz, H. e Verissimo, P. (1993). Real Time and Dependability Concepts. Em Mullender (ed.), *Distributed Systems*, 2ª ed.. Addison-Wesley.

Kopetz *et al.* 1989 — Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W. Senft, C. e Zainlinger, R. (1989). Distributed Fault-Tolerant Real-Time Systems – The MARS Approach. *IEEE Micro*, Vol. 9, No. 1.

Krawczyk *et al.* 1997 — Krawczyk, H., Bellare, M. e Canetti, R. (1997). *HMAC: Keyed-Hashing for Message Authentication*. Internet RFC 2104.

Krumm *et al.* 2000 — Krumm, J., Harris, S., Meyers, B., Brumitt , B., Hale, M. e Shafer, S. (2000). Multi-Camera Multi-Person Tracking for EasyLiving. *Proceedings of the Third IEEE International Workshop on Visual Surveillance (VS'2000)*, Dublin, Irlanda, Julho, pgs. 3–10.

Kshemkalyani e Singhal 1991 — Kshemkalyani, A. e Singhal, M. (1991). Invariant-Based Verification of a Distributed Deadlock Detection Algorithm. *IEEE Transactions on Software Engineering*, Vol. 17, No. 8. Agosto.

Kshemkalyani e Singhal 1994 — Kshemkalyani, A. e Singhal, M. (1994). On Characterisation and Corrrectness of Distributed Deadlock detection, *Journal of Parallel and Distributed Computing*, Vol. 22, pgs. 44–59.

Kubiatowicz 2003 — Kubiatowicz, J. (2003). Extracting Guarantees from Chaos, *Communications of the ACM*, pgs. 33–38, vol. 46, No. 2, Fevereiro.

Kubiatowicz *et al.* 2000 — Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C. e Zhao, B. (2000). OceanStore: an architecture for global-scale persistent storage. Em *ASPLOS 2000*, pgs. 190–201, Novembro.

Kung e Robinson 1981 — Kung, H.T. e Robinson, J.T. (1981). Optimistic methods for concurrency control. *ACM Transactions on Database Systems*, Vol. 6, No. 2, pgs. 213–26.

Kurose e Ross 2000 — Kurose, J.F. e Ross, K.W. (2000). *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison Wesley Longman.

Ladin *et al.* 1992 — Ladin, R., Liskov, B., Shrira, L. e Ghemawat, S. (1992). Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, Vol. 10, No. 4, pgs. 360–91.

Lai 1992 — Lai, X. (1992). On the Design and Security of Block Ciphers, *ETH Series in Information Processing*, Vol. 1, Konstanz: Hartung-Gorre Verlag.

Lai e Massey 1990 — Lai, X. e Massey, J. (1990). A proposal for a new Block Encryption Standard. *Advances in Cryptology–Eurocrypt'90*. Em *Proceedings*, Springer-Verlag, pgs. 389–404.

Lamport 1978 — Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, Vol. 21, No. 7, pgs. 558–65.

Lamport 1979 — Lamport, L. (1979). How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions Computers*, Vol. C-28, No. 9, pgs. 690–1.

Lamport 1986 — Lamport, L. (1986). On interprocess communication, parts I and II. *Distributed Computing*, Vol. 1, No. 2, pgs. 77–101.

Lamport 1989 — Lamport, L. (1989). The part-time parliament. Technical Report 49, DEC SRC, Palo Alto. (Também em *ACM Transactions on Computer Systems*, vol. 16, no. 2, pgs. 133–169, 1998.)

Lamport 1998 — Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, Vol. 16, No. 2, Maio, pgs. 133–69.

Lamport *et al.* 1982 — Lamport, L., Shostak, R. e Pease, M. (1982). Byzantine Generals Problem. *ACM Transactions Programming Languages and Systems*, Vol. 4, No. 3, pgs. 382–401.

Lampson 1971 — Lampson, B. (1971). Protection. Em *Proceedings 5th Princeton Conference on Information Sciences and Systems*, Princeton, p. 437. Reimpresso em *ACM Operating Systems Review*. Vol. 8, No. 1, Janeiro, p. 18.

Lampson 1981 — Lampson, B.W. (1981). Atomic Transactions. Em *Distributed systems: Architecture and Implementation. Lecture Notes in Computer Science 105*, pgs. 254–9. Berlim: Springer-Verlag.

Lampson 1986 — Lampson, B.W. (1986). Designing a Global Name Service. Em *Proceedings 5th ACM Symposium Principles of Distributed Computing*, pgs. 1–10, Agosto.

Lampson *et al.* 1992 — Lampson, B.W., Abadi, M., Burrows, M. e Wobber, E. (1992). Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, Vol. 10, No. 4, pgs. 265–310.

Langheinrich 2001 — Langheinrich, M. (2001). Privacy by design – principles of privacy-aware ubiquitous systems. *Proceedings Third International Conference on Ubiquitous Computing (Ubicomp 2001)*, Atlanta, GA, EUA, Set.–Out., Springer-Verlag, pgs. 273–291.

Langworthy 2004 — Langworthy, D. (ed.) (2004) *Web Services Coordination (WS-Coordination)*, IBM, Microsoft, BEA. Novembro.

Lea *et al.* 1993 — Lea, R., Jacquemot, C. e Pillevesse, E. (1993). COOL: system support for distributed programming. *Comms. ACM*, Vol. 36, No. 9, pgs. 37–46.

Leach *et al.* 1983 — Leach, P.J., Levine, P.H., Douros, B.P., Hamilton, J.A., Nelson, D.L. e Stumpf, B.L. (1983). The architecture of an integrated local network. *IEEE J. Selected Areas in Communications*, Vol. SAC-1, No. 5, pgs. 842–56.

Lee e Thekkath 1996 — Lee, E.K. e Thekkath, C.A. (1996). Petal: Distributed Virtual Disks, Em *Proc. 7th Intl. Conf. on Architectural Support for Prog. Langs. and Operating Systems*, Outubro, pgs. 84–96.

Lee *et al.* 1996 — Lee, C., Rajkumar, R. e Mercer, C. (1996). Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. Em *Proceedings Multimedia Japan'96*.

Leffler *et al.* 1989 — Leffler, S., McKusick, M., Karels, M. e Quartermain, J. (1989). *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Reading, MA: Addison-Wesley.

Leibowitz *et al.* 2003 — Leibowitz, N., Ripeanu, M. e Wierzbicki, A. (2003). Deconstructing the Kazaa Network. Em *3rd IEEE Workshop on Internet Applications (WIAPP'03)*. Santa Clara, CA.

Leiner 1997 — Leiner, B.M., Cerf, V.G., Clark, D.D., Kahn, R.E., Kleinrock, L., Lynch, D.C., Postel, J., Roberts, L.G. e Wolff, S. (1997). A Brief History of the Internet, *Comms. ACM*, Vol. 40, No. 1, Fev., pgs. 102–108.

Leland *et al.* 1993 — Leland, W.E., Taqqu, M.S., Willinger, W. e Wilson, D.V. (1993). On the Self-Similar Nature of Ethernet Traffic. *ACM SIGCOMM'93*, São Francisco.

Lenoski *et al.* 1992 — Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.D., Gupta, A., Hennessy, J., Horowitz, M. e Lam, M.S. (1992). The Stanford Dash multiprocessor, *IEEE Computer*, Vol. 25, No. 3, pgs. 63–79.

Leslie *et al.* 1996 — Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R. e Hyden, E. (1996). The design and implementation of an operating system to support distributed multimedia applications, *ACM Journal of Selected Areas in Communication*, Vol. 14, No. 7, pgs. 1280–97.

Li e Hudak 1989 — Li, K. e Hudak, P. (1989). Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pgs. 321–59.

Liedtke 1996 — Liedtke, J. (1996). Towards real microkernels, *Comms. ACM*, Vol. 39, No. 9, pgs. 70–7.

Linux AFS — *The Linux AFS FAQ*.

Lipton e Sandberg 1988 — Lipton, R. e Sandberg, J. (1988). *PRAM: A scalable shared memory*. Technical Report CS-TR-180-88, Princeton University.

Liskov 1988 — Liskov, B. (1988). Distributed programming in Argus. *Comms. ACM*, Vol. 31, No. 3, pgs. 300–12.

Liskov 1993 — Liskov, B. (1993). Practical uses of synchronized clocks in distributed systems, *Distributed Computing*, Vol. 6, No. 4, pgs. 211–19.

Liskov e Scheifler 1982 — Liskov, B. e Scheifler, R.W. (1982). Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions Programming Languages and Systems*, Vol. 5, No. 3, pgs. 381–404.

Liskov e Shrira 1988 — Liskov, B. e Shrira, L. (1988). Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. Em *Proceedings SIGPLAN'88 Conference Programming Language Design and Implementation*. Atlanta.

Liskov *et al.* 1991 — Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., Williams, M. (1991). Replication in the Harp File System. Em *Proceedings 13th ACM Symposium on Operating System Principles*, pgs. 226–38.

Liu e Albitz 1998 — Liu, C. e Albitz, P. (1998). *DNS and BIND*, terceira edição. O'Reilly.

Liu e Layland 1973 — Liu, C.L. e Layland, J.W. (1973). Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, Vol. 20, No. 1.

Loepere 1991 — Loepere, K. (1991). *Mach 3 Kernel Principles*. Open Software Foundation e Carnegie-Mellon University.

Lundelius e Lynch 1984 — Lundelius, J. e Lynch, N. (1984). An Upper and Lower Bound for Clock Synchronization. *Information and Control* 62, 2/3 (Ago./Set.), pgs. 190–204.

Lynch 1996 — Lynch, N. (1996). *Distributed Algorithms*. Morgan Kaufmann.

Ma 1992 — Ma, C. (1992). *Designing a Universal Name Service*. Technical Report 270, University of Cambridge.

Macklem 1994 — Macklem, R. (1994). Not Quite NFS: Soft Cache Consistency for NFS. *Proceedings of the Winter'94 USENIX Conference*, São Francisco, CA, Janeiro, pgs. 261–278.

Madhavapeddy *et al.* 2003 — Madhavapeddy, A., Scott, D., Sharp, R. (2003). Context-aware computing with sound. *Proceedings Fifth International Conference on Ubiquitous Computing (Ubicomp 2003)*, Seattle, WA, EUA, Outubro, pgs. 315–332.

Maekawa 1985 — Maekawa, M. (1985). A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 2, pgs. 145–159.

Maffeis 1995 — Maffeis, S. (1995). Adding group communication and fault tolerance to CORBA. Em *Proceedings of the 1995 USENIX conference on object-oriented technologies*. Monterey, CA, EUA. pgs. 135–146.

Malkin 1993 — Malkin, G. (1993). *RIP Version 2 – Carrying Additional Information*, Internet RFC 1388.

Marsh *et al.* 1991 — Marsh, B., Scott, M., LeBlanc, T. e Markatos, E. (1991). First-class User-level *Threads*. Em *Proceedings 13th ACM Symposium on Operating System Principles*, pgs. 110–21.

Martin *et al.* 2004 — Martin, T., Hsiao, M., Ha, D. e Krishnaswami, J. (2004). Denial-of-Service Attacks on Battery-powered Mobile Computers. *Proceedings 2nd IEEE Pervasive Computing Conference*, Orlando, FL, EUA, Março, pgs. 309–318.

Marzullo e Neiger 1991 — Marzullo, K. e Neiger, G. (1991). Detection of global state predicates, In Toug, S., Spirakis, P. e Kirousis, L. (eds), *Proceedings 5th International Workshop on Distributed Algorithms*. Springer-Verlag, pgs. 254–72.

Mattern 1989 — Mattern, F. (1989). Virtual Time and Global States in Distributed Systems. Em Cosnard, M. *et al.* (eds), *Proceedings Workshop on Parallel and Distributed Algorithms*. Amsterdam: Holanda do Norte, pgs. 215–26.

Maymounkov e Mazieres 2002 — Maymounkov, P. e Mazieres, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. Em *Proceedings of IPTPS02*, Cambridge, EUA, Março.

mbone — *BIBs: Introduction to the Multicast Backbone*.

McGraw e Felden 1999 — McGraw, G. e Felden, E. (1999). *Securing Java*. John Wiley & Sons.

Melliar-Smith *et al.* 1990 — Melliar-Smith, P., Moser, L. e Agrawala, V. (1990). Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, pgs. 17–25.

Menezes 1993 — Menezes, A. (1993). *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers.

Menezes *et al*. 1997 — Menezes, A., van Oorschot, O. e Vanstone, S. (1997). *Handbook of Applied Cryptography*. CRC Press.

Metcalfe e Boggs 1976 — Metcalfe, R.M. e Boggs, D.R. (1976). Ethernet: distributed packet switching for local computer networks. *Comms. ACM*, Vol. 19, pgs. 395–403.

Milanovic *et al*. 2004 — Milanovic, N., Malek, M., Davidson, A. e Milutinovic, V. (2004). Routing and Security in Mobile Ad Hoc Networks. *IEEE Computer*, Vol. 37, No. 2, Fevereiro, pgs. 69–73.

Mills 1995 — Mills, D. (1995). Improved Algorithms for Synchronizing Computer Network Clocks, *IEEE Transactions Networks*, Junho, pgs. 245–54.

Milojicic *et al*. 1999 — Milojicic, J., Douglis, F. e Wheeler, R. (1999). *Mobility, Processes, Computers and Agents*. Reading: Addison-Wesley.

Minnich e Farber 1989 — Minnich, R. e Farber, D. (1989). The Mether System: a Distributed Shared Memory for SunOS 4.0. Em *Proceedings Summer 1989 Usenix Conference*.

Mitchell e Dion 1982 — Mitchell, J.G. e Dion, J. (1982). A comparison of two network-based file servers. *Comms. ACM*, Vol. 25, No. 4, pgs. 233–45.

Mitchell *et al*. 1992 — Mitchell, C.J., Piper, F. e Wild, P. (1992). Digital Signatures. Em Simmons, G.J. (ed.), *Contemporary Cryptology*. Nova York: IEEE Press.

Mockapetris 1987 — Mockapetris, P. (1987). *Domain names – concepts and facilities*. Internet RFC 1034. Novembro.

Mogul 1994 — Mogul, J.D. (1994). Recovery in Spritely NFS, *Computing Systems*, Vol. 7, No. 2.

Mok 1985 — Mok, A.K. (1985). SARTOR – A Design Environment for Real-Time Systems. *Ninth IEEE COMP-SAC*.

Morin 1997 — Morin, R. (ed.) (1997). *MkLinux: Microkernel Linux for the Power Macintosh*. Prime Time Freeware.

Morris *et al*. 1986 — Morris, J., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S. e Smith, F.D. (1986). Andrew: a distributed personal computing environment. *Comms. ACM*, Vol. 29, No. 3, pgs. 184–201.

Mosberger 1993 — Mosberger, D. (1993). *Memory Consistency Models*. Technical Report 93/11, University of Arizona.

Moser *et al*. 1994 — Moser, L., Amir, Y., Melliar-Smith, P. e Agarwal, D. (1994). Extended Virtual Synchrony. Em *Proceedings 14th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, pgs. 56–65.

Moser *et al*. 1996 — Moser, L., Melliar-Smith, P., Agarwal, D., Budhia, R., e Lingley-Papadopoulos, C. (1996). Totem: a Fault-Tolerant Multicast Group Communication System. *Comms. ACM*, Vol. 39, No. 4, pgs. 54–63.

Moser *et al*. 1998 — Moser, L., Melliar-Smith, P. e Narasimhan, P. (1998). Consistent object replication in the Eternal system. *Theory and practice of object systems*, Vol. 4, No. 2.

Moss 1985 — Moss, E. (1985). *Nested Transactions, An Approach to Reliable Distributed Computing*. MIT Press.

Multimedia Directory 2005 — Multimedia Directory (2005). Scala Inc.

Murphy *et al*. 2001 — Murphy, A.L., Picco, G.P. e Roman, G.-C. (2001). Lime: A Middleware for Physical and Logical Mobility. *Proceedings 21st International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix, AZ, EUA, Abril, pgs. 524–233.

Muthitacharoen *et al*. 2002 — Muthitacharoen, A., Morris, R., Gil, T.M. e Chen, B. (2002). Ivy: A Read/Write Peer-to-peer File System. Em *Fifth Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA. Dezembro.

Myers e Liskov 1997 — Myers, A.C. e Liskov, B. (1997). A Decentralized Model for Information Flow Control, *ACM Operating Systems Review*, Vol. 31, No. 5, pgs. 129–42, Dezembro.

Nagle 1984 — Nagle, J. (1984). Congestion Control in TCP/IP Internetworks, *Computer Communications Review*, Vol. 14, pgs. 11–17, Outubro.

Nagle 1987 — Nagle, J. (1987). On Packet Switches with Infinite Storage. *IEEE Transactions on Communications*, Vol. 35, No. 4.

National Bureau of Standards 1977 — National Bureau of Standards (1977). *Data Encryption Standard (DES)*. Federal Information Processing Standards No. 46, Washington DC: US National Bureau of Standards.

nbcr.sdsc.edu — National Biomedical Computation Resource, University of California, San Diego.

Needham 1993 — Needham, R. (1993). Names. In Mullender, S. (ed.), *Distributed Systems, an Advanced Course*, 2ª ed., Wokingham, Inglaterra: ACM Press/Addison-Wesley, pgs. 315–26.

Needham e Schroeder 1978 — Needham, R.M. e Schroeder, M.D. (1978). Using encryption for authentication in large networks of computers. *Comms. ACM*, Vol. 21, pgs. 993–9.

Nelson *et al*. 1988 — Nelson, M.N., Welch, B.B. e Ousterhout, J.K. (1988). Caching in the Sprite Network File System, *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pgs. 134–154.

Netscape 1996 — Netscape Corporation (1996). *SSL 3.0 Specification*.

Neuman *et al*. 1999 — Neuman, B.C., Tung, B. e Wray, J. (1999). *Public Key Cryptography for Initial Authentication in Kerberos*, Internet Draft ietf-cat-kerberos-pk-init-09, Julho.

Neumann e Ts'o 1994 — Neuman, B.C. e Ts'o, T. (1994). Kerberos: An Authentication Service for Computer Networks, *IEEE Communications*, vol. 32, no. 9, pgs. 33–38, Set.

Newcomer 2002 — Newcomer, E. (2002). *Understanding Web Services XML, WSDL, SOAP and UDDI*. Boston: Pearson.

Nielsen and Thatte 2001 — Nielsen, H.F. e Thatte, S. (2001). *Web Services Routing Protocol (WS-Routing)*. Microsoft Corporation. Outubro.

Nielson *et al*. 1997 — Nielsen, H., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H. e Lilley, C. (1997). Network Performance Effects of HTTP/1.1, CSS1, and PNG. *Proceedings SIGCOMM'97*.

NIST 1994 — National Institute for Standards and Technology (1994). *Digital Signature Standard*, NIST FIPS 186. US Department of Commerce.

NIST 1999 — National Institute for Standards and Technology (1999). *AES – a Crypto Algorithm for the Twenty-first Century*, US Department of Commerce.

NIST 2002 — National Institute for Standards and Technology (2002). *Secure Hash Standard*. NIST FIPS 180-2 + Change Notice to include SHA-224, US Department of Commerce.

NIST 2004 — National Institute for Standards and Technology (2004). *NIST Brief Comments on Recent Cryptanalytic Attacks on Secure Hashing Functions and the Continued Security Provided by SHA-1*, US Department of Commerce, Agosto.

Noble e Satyanarayanan 1999 — Noble, B. e Satyanarayanan, M. (1999). Experience with Adaptive Mobile Applications in Odyssey. *Mobile Networks and Applications*, Vol. 4 , No. 4, Dezembro, pgs. 245–254.

now.cs.berkeley.edu — *The Berkeley NOW project home page*.

Oaks e Wong 1999 — Oaks, S. e Wong, H. (1999). *Jave Threads*, 2ª ed., O'Reilly.

Ohkubo *et al*. 2003 — Ohkubo, M., Suzuki, K. e Kinoshita, S. (2003). Cryptographic approach to 'privacy-friendly' tags. *Proceedings RFID Privacy Workshop*, MIT, EUA.

Olson and Ogbuji 2002 — Olson, M. e Ogbuji, U. (2002) *Choose the best tool for the task at hand*.

OMG 2000a — Object Management Group (2000). *Trading Object Service Specification*,Vn. 1.0. Needham, MA: OMG.

OMG 2000b — Object Management Group (2000). *Concurrency Control Service Specification*. Needham, MA: OMG.

OMG 2002a — Object Management Group (2002). *The CORBA IDL Specification*. Needham, MA: OMG.

OMG 2002b — Object Management Group (2002). *CORBA Security Service Specification* Vn. 1.8. Needham, MA: OMG.

OMG 2002c — Object Management Group (2002). *Value Type Semantics*. Needham, MA: OMG.

OMG 2002d — Object Management Group (2002). *Life Cycle Service*, Vn. 1.2. Needham, MA: OMG.

OMG 2002e — Object Management Group (2002). *Persistent State Service*, Vn. 2.0. Needham, MA: OMG.

OMG 2003 — Object Management Group, (2003). *Object Transaction Service Specification*. Version 1.4. Needham, MA: OMG.

OMG 2004a — Object Management Group (2004). *CORBA/IIOP 3.0.3 Specification*. Needham, MA: OMG.

OMG 2004b — Object Management Group (2004). *Naming Service Specification*. Needham, MA: OMG.

OMG 2004c — Object Management Group (2004). *Event Service Specification*. Vn. 1.2. Needham, MA: OMG.

OMG 2004d — Object Management Group (2004). *Notification service Specification*. Vn. 1.1. Needham, MA: OMG. Technical report telecom/98-06-15.

OMG 2004e — Object Management Group (2004). *CORBA Messaging*. Needham, MA: OMG.

Omidyar e Aldridge 1993 — Omidyar, C.G. e Aldridge, A. (1993). Introduction to SDH/SONET. *IEEE Communications Magazine*, Vol. 31, pgs. 30–3, Set.

OpenNap 2001 — OpenNap: Open Source Napster Server, Beta release 0.44, Setembro de 2001.

Oppen e Dalal 1983 — Oppen, D.C. e Dalal Y.K. (1983). The Clearinghouse: a decentralized agent for locating named objects in a distributed environment. *ACM Trans. on Office Systems*, Vol. 1, pgs. 230–53.

Oram 2001 — Oram, A. (2001). *Peer-to-Peer: Harnessing the Benefits of Disruptive Technologies*, O'Reilly, Sebastapol, CA.

Orfali *et al.* 1996 — Orfali, R., Harkey, D. e Edwards, J. (1996). *The Essential Distributed Objects Survival Guide*. Nova York: Wiley.

Orfali *et al.* 1997 — Orfali, R., Harkey, D., e Edwards, J. (1997) *Instant CORBA*. Nova York: Wiley.

Organick 1972 — Organick, E.I. (1972). *The MULTICS System: An Examination of its Structure*. Cambridge, MA: MIT Press.

Orman *et al.* 1993 — Orman, H., Menze, E., O'Malley, S. e Peterson, L. (1993). A fast and general implementation of Mach IPC in a Network. Em *Proceedings Third USENIX Mach Conference*, Abril.

OSF 1997 — *Introduction to OSF DCE*. The Open Group.

Ousterhout *et al.* 1985 — Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M. e Thompson, J. (1985). A Trace-driven analysis of the UNIX 4.2 BSD file system. Em *10th ACM Symposium Operating System Principles*.

Ousterhout *et al.* 1988 — Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M. e Welch, B. (1988). The Sprite Network Operating System. *IEEE Computer*, Vol. 21, No. 2, pgs. 23–36.

Parker 1992 — Parker, B. (1992). *The PPP AppleTalk Control Protocol (ATCP)*. Internet RFC 1378.

Parrington *et al.* 1995 — Parrington, G.D., Shrivastava, S.K., Wheater, S.M. e Little, M.C. (1995). The Design and Implementation of Arjuna, *USENIX Computing Systems Journal*, Vol. 8, No. 3.

Partridge 1992 — Partridge, C. (1992). *A Proposed Flow Specification*. Internet RFC 1363.

Patel e Abowd 2003 — Patel, S.N. e Abowd, G.D. (2003). A 2-way Laser-assisted Selection Scheme for Handhelds in a Physical Environment. *Proceedings Fifth International Conference on Ubiquitous Computing (Ubicomp 2003)*, Seattle, WA, EUA, Outubro, pgs. 200–207.

Patterson *et al.* 1988 — Patterson, D., Gibson, G. e Katz, R. (1988). A Case for Redundant Arrays of Interactive Disks, ACM *International Conf. on Management of Data (SIGMOD)*, pgs. 109–116, Maio.

Pease *et al.* 1980 — Pease, M., Shostak, R. e Lamport, L. (1980). Reaching agreement in the presence of faults. *Journal of the ACM*, Vol. 27, No. 2, Abril, pgs. 228–34.

Pedone e Schiper 1999 — Pedone, F. e Schiper, A. (1999). Generic Broadcast. Em *Proceedings of the 13th International Symposium on Distributed Computing* (DISC'99), Setembro.

Perrig *et al.* 2002 — Perrig, A., Szewczyk, R., Wen, V., Culler, D. e Tygar, D. (2002). SPINS: Security Protocols for Sensor Networks. *Wireless Networks*, Vol. 8, No. 5, Setembro, pgs. 521–534.

Petersen *et al.* 1997 — Petersen, K., Spreitzer, M., Terry, D., Theimer, M. e Demers, A. (1997). Flexible update propagation for weakly consistent replication. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pgs. 288–301.

Peterson 1988 — Peterson, L. (1988). The Profile Naming Service. *ACM Transactions Computer Systems*, Vol. 6, No. 4, pgs. 341–64.

Peterson *et al.* 1989 — Peterson, L.L., Buchholz, N.C. e Schlichting, R.D. (1989). Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, Vol. 7, No. 3, pgs. 217–46.

Pike *et al.* 1993 — Pike, R., Presotto, D., Thompson, K., Trickey, H. e Winterbottom, P. (1993). The Use of Name Spaces in Plan 9. *Operating Systems Review*, Vol. 27, No. 2, Abril de 1993, pgs. 72–76.

Plaxton *et al.* 1997 — Plaxton, C.G., Rajaraman, R. e Richa, A.W. (1997). Accessing nearby copies of replicated objects in a distributed environment. *ACM Symposium on Parallel Algorithms and Architectures*, pgs. 311–320.

Ponnekanti e Fox 2004 — Ponnekanti, S. e Fox, A. (2004). Interoperability among Independently Evolving Web Services. *Proceedings ACM/Usenix/IFIP 5th International Middleware Conference*, Toronto, Canadá, Setembro.

Ponnekanti *et al.* 2001 — Ponnekanti, S.R., Lee, B., Fox, A., Hanrahan, P. e Winograd, T. (2001). ICrafter: A Service Framework for Ubiquitous Computing Environments. *Proceedings Third International Conference on Ubiquitous Computing (Ubicomp 2001)*, Atlanta, GA, EUA, Set.–Out., Springer-Verlag, pgs. 56–75.

Popek e Walker 1985 — Popek, G. e Walker, B. (eds) (1985). *The LOCUS Distributed System Architecture*. Cambridge, MA: MIT Press.

Postel 1981a — Postel, J. (1981). *Internet Protocol*. Internet RFC 791.

Postel 1981b — Postel, J. (1981). *Transmission Control Protocol*. Internet RFC 793.

Pottie e Kaiser 2000 — Pottie, G.J. e Kaiser, W.J. (2000). Embedding the internet: wireless integrated network sensors. *Comms. ACM*, Vol. 43, No. 5, Maio, pgs. 51–58.

Powell 1991 — Powell, D. (ed.) (1991). *Delta-4: a Generic Architecture for Dependable Distributed Computing*. Berlim e Nova York: Springer-Verlag.

Pradhan e Chiueh 1998 — Pradhan, P. e Chiueh, T. (1998). Real-Time Performance Guarantees over Wired and Wireless LANS, em *IEEE Conference on Real-Time Applications and Systems*, RTAS'98, Junho.

Preneel *et al.* 1998 — Preneel, B., Rijmen, V. e Bosselaers, A. (1998). Recent developments in the design of conventional cryptographic algorithms. Em *Computer Security and Industrial Cryptography, State of the Art and Evolution*, Lecture Notes in Computer Science, No. 1528, Springer-Verlag, pgs. 106–131.

privacy.nb.ca — *International Cryptography Freedom*.

Radia *et al.* 1993 — Radia, S., Nelson, M. e Powell, M. (1993). *The Spring Naming Service*. Technical Report 93–16, Sun Microsystems Laboratories, Inc.

Raman e McCanne 1999 — Raman, S. e McCanne, S. (1999). A model, analysis, and protocol framework for soft state-based communication. *Proceedings ACM SIGCOMM*, Cambridge, MA, EUA, Setembro, pgs. 15–25.

Randall e Szydlo 2004 — Randall, J. e Szydlo, M. (2004). Collisions for SHA0, MD5, HAVAL, MD4, and RIPEMD, but SHA1 still secure, *RSA Laboratories Technical Note*, 31 de Agosto.

Rashid 1985 — Rashid, R.F. (1985). Network operating systems. Em *Local Area Networks: An Advanced Course, Lecture Notes in Computer Science*, 184, Springer-Verlag, pgs. 314–40.

Rashid 1986 — Rashid, R.F. (1986). From RIG to Accent to Mach: the evolution of a network operating system. Em *Proceedings of the ACM/IEEE Computer Society Fall Joint Conference*, ACM, Novembro.

Rashid e Robertson 1981 — Rashid, R. e Robertson, G. (1981). Accent: a communications oriented network operating system kernel. *ACM Operating Systems Review*, Vol. 15, No. 5, pgs. 64–75.

Rashid *et al.* 1988 — Rashid, R., Tevanian Jr, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W.J. e Chew, J. (1988). Machine-Independent Virtual Memory Management

for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions Computers*, Vol. 37, No. 8, pgs. 896–907.

Ratnasamy *et al.* 2001 — Ratnasamy, S., Francis, P., Handley, M., Karp, R. e Shenker, S. (2001). A scalable content-addressable network. Em *Proc. ACM SIGCOMM 2001*, Agosto.

Raynal 1988 — Raynal, M. (1988). *Distributed Algorithms and Protocols*. Wiley.

Raynal 1992 — Raynal, M. (1992). About Logical Clocks for Distributed Systems. *ACM Operating Systems Review*, Vol. 26, No. 1, pgs. 41–8.

Raynal e Singhal 1996 — Raynal, M. e Singhal, M. (1996). Capturing Causality in Distributed Systems. *IEEE Computer*, Fevereiro, pgs. 49–56.

Redmond 1997 — Redmond, F.E. (1997). *DCOM: Microsoft Distributed Component Model*. IDG Books Worldwide.

Reed 1983 — Reed, D.P. (1983). Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, Vol. 1, No. 1, pgs. 3–23.

Rescorla 1999 — Rescorla, E. (1999). *Diffie-Hellman Key Agreement Method*. Internet RFC 2631.

Rether — Rether: A Real-Time Ethernet Protocol.

Rhea *et al.* 2001 — Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H. e Kubiatowicz, J. (2001). Maintenance-Free Global Data Storage, *IEEE Internet Computing*, Vol 5, No 5, Setembro/Outubro, pgs. 40–49.

Rhea *et al.* 2003 — Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B. e Kubiatowicz, J. (2003). Pond: the OceanStore Prototype, *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*.

Ricart e Agrawala 1981 — Ricart, G. e Agrawala, A.K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Comms. ACM*, Vol. 24, No. 1, pgs. 9–17.

Richardson *et al.* 1998 — Richardson, T., Stafford-Fraser, Q., Wood, K.R. e Hopper, A. (1998). Virtual Network Computing, *IEEE Internet Computing*. Vol. 2, No. 1, Jan/Fev. pgs. 33–8.

Ritchie 1984 — Ritchie, D. (1984). A Stream Input Output System. *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, pt 2, pgs. 1897–910.

Rivest 1992 — Rivest, R. (1992). *The MD5 Message-Digest Algorithm*. Internet RFC 1321.

Rivest 1992a — Rivest, R. (1992). The RC4 Encryption Algorithm, RSA Data Security Inc.

Rivest *et al.* 1978 — Rivest, R.L., Shamir, A. e Adelman, L. (1978). A method of obtaining digital signatures and public key cryptosystems. *Comms. ACM*, Vol. 21, No. 2, pgs. 120–6.

Rodrigues *et al.* 1998 — Rodrigues, L., Guerraoui, R. e Schiper, A. (1998). Scalable Atomic Multicast. Em *Proceedings IEEE IC3N'98*. Technical Report 98/257. École polytechnique fédérale de Lausanne.

Rose 1992 — Rose, M.T. (1992). The Little Black Book: Mail Bonding with OSI Directory Services. Englewood Cliffs, NJ: Prentice-Hall.

Rosenblum e Ousterhout 1992 — Rosenblum, M. e Ousterhout, J. (1992). The Design and Implementation of a Log-Structured File System, *ACM Transactions on Computer Systems*, Vol. 10, No. 1, Fevereiro, pgs. 26–52.

Rosenblum e Wolf 1997 — Rosenblum, D.S. e Wolf, A.L. (1997). A Design Framework for Internet-Scale Event Observation and Notification. Em *Proceedings sixth European Software Engineering Conference/ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering*, Zurique, Suíça.

Rowley 1998 — Rowley, A. (1998). *A Security Architecture for Groupware*, Tese de Doutorado, Queen Mary e Westfield College, University of London.

Rowstron e Druschel 2001 — Rowstron, A. e Druschel, P. (2001). Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Em *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Alemanha, Nov.

Rozier *et al.* 1988 — Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Leonard, P. e Neuhauser, W. (1988). Chorus Distributed Operating Systems. *Computing Systems Journal*, Vol. 1, No. 4, pgs. 305–70.

Rozier *et al.* 1990 — Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Leonard, P. e Neuhauser, W. (1990). *Overview*

	of the Chorus Distributed Operating System. Technical Report CS/TR-90-25.1, Chorus Systèmes, França.
RTnet	RTnet: Hard Real-Time Networking for Linux/RTAI.
Salber *et al.* 1999	Salber, D., Dey, A.K. e Abowd, G.D. (1999). The Context Toolkit: Aiding the Development of Context-Enabled Applications. *Proceedings of the 1999 Conference on Human Factors in Computing Systems (CHI'99)*, Pittsburgh, PA, Maio, pgs. 434–441.
Saltzer *et al.* 1984	Saltzer, J.H., Reed, D.P. e Clarke, D. (1984). End-to-End Arguments in System Design, *ACM Transactions on Computer Systems* Vol. 2, No. 4, pgs. 277–88.
Sandberg 1987	Sandberg, R. (1987). *The Sun Network File System: Design, Implementation and Experience*. Technical Report. Mountain View, CA: Sun Microsystems.
Sandberg *et al.* 1985	Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. e Lyon, B. (1985). The Design and Implementation of the Sun Network File System. Em *Proceedings Usenix Conference*, Portland, OR.
Sandholm and Gawor 2003	Sandholm, T. e Gawor, J. (2003). *Globus Toolkit 3 Core – A Grid Service Container Framework*. Julho.
Sandhu *et al.* 1996	Sandhu, R., Coyne, E., Felstein, H. e Youman, C. (1996). Role-Based Access Control Models, *IEEE Computer*, Vol. 29, No. 2, Fevereiro.
Sane *et al.* 1990	Sane, A., MacGregor, K. e Campbell, R. (1990). Distributed Virtual Memory Consistency Protocols: Design and Performance. *Second IEEE Workshop on Experimental Distributed Systems*, pgs. 91–6, Outubro.
Sansom *et al.* 1986	Sansom, R.D., Julin, D.P. e Rashid, R.F. (1986). *Extending a capability based system into a network environment*. Technical Report CMU-CS-86-116, Carnegie-Mellon University.
Santifaller 1991	Santifaller, M. (1991). *TCP/IP and NFS, Internetworking in a Unix Environment*. Reading, MA: Addison-Wesley.
Saroiu *et al.* 2002	Saroiu, S., Gummadi, P. e Gribble, S. (2002). A Measurement Study of Peer-to-Peer File Sharing Systems, Em *Proc. Multimedia Computing and Networking (MMCN)*, 2002.
Sastry *et al.* 2003	Sastry, N., Shankar, U. e Wagner, D. (2003). Secure Verification of Location Claims. *Proceedings ACM Workshop on Wireless Security (WiSe 2003)*, Setembro, pgs. 1–10.
Satyanarayanan 1981	Satyanarayanan, M. (1981). A study of file sizes and functional lifetimes. Em *Proceedings 8th ACM Symposium on Operating System Principles*, Asilomar, CA.
Satyanarayanan 1989a	Satyanarayanan, M. (1989). Distributed File Systems. In Mullender, S. (ed.), *Distributed Systems, an Advanced Course*, 2ª ed., Wokingham: ACM Press/ Addison-Wesley. pgs. 353–83.
Satyanarayanan 1989b	Satyanarayanan, M. (1989). Integrating Security in a Large Distributed System. *ACM Transactions on Computer Systems*, Vol. 7, No. 3, pgs. 247–80.
Satyanarayanan 2001	Satyanarayanan, M. (2001). Pervasive computing: Vision and challenges. *IEEE Personal Communications*, Vol. 8, No. 4, Agosto, pgs. 10–17.
Satyanarayanan *et al.* 1990	Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H. e Steere, D.C. (1990). Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, Vol. 39, No. 4, pgs. 447–59.
Saunders 1987	Saunders, B. (1987). The Information Structure of Distributed Mutual Exclusion Algorithms. *ACM Transactions on Computer Systems*, Vol. 3, No. 2, pgs. 145–59.
Scheifler e Gettys 1986	Scheifler, R.W. e Gettys, J. (1986). The X window system. *ACM Transactions on Computer Graphics*, Vol. 5, No. 2, pgs. 76–109.
Schilit *et al.* 1994	Schilit, B.N., Adams, N.I. e Want, R. (1994). Context-Aware Computing Applications. *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dezembro, pgs. 85–90.
Schiper e Raynal 1996	Schiper, A. e Raynal, M. (1996). From Group Communication to Transactions in Distributed Systems. *Comms. ACM*, Vol. 39, No. 4, pgs. 84–7.

Schiper e Sandoz 1993 — Schiper, A. e Sandoz, A. (1993). Uniform reliable multicast in a virtually synchronous environment. *Proceedings 13th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, pgs. 561–8.

Schlageter 1982 — Schlageter, G. (1982). Problems of Optimistic Concurrency Control in Distributed Database Systems, *SigMOD Record*, Vol. 13, No. 3, pgs. 62–6.

Schmidt 1998 — Schmidt, D. (1998). Evaluating architectures for multithreaded object request brokers, *Comms. ACM*, Vol. 44, No. 10, pgs. 54–60.

Schneider 1990 — Schneider, F.B. (1990). Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, Vol. 22, No. 4, pgs. 300–19.

Schneider 1996 — Schneider, S. (1996). Security properties and CSP. Em *IEEE Symposium, on Security and Privacy*, pgs. 174–187.

Schneier 1996 — Schneier, B. (1996). *Applied Cryptography*, segunda edição. Nova York: John Wiley.

Schroeder e Burrows 1990 — Schroeder, M. e Burrows, M. (1990). The Performance of Firefly RPC. *ACM Transactions on Computer Systems*, Vol. 8, No. 1. pgs. 1–17.

Schroeder *et al*. 1984 — Schroeder, M.D., Birrell, A.D. e Needham, R.M. (1984). Experience with Grapevine: The growth of a distributed system, *ACM Transactions on Computer Systems*, Vol. 2, No. 1.

Schulzrinne *et al*. 1996 — Schulzrinne, H., Casner, S., Frederick, D. e Jacobson, V. (1996). *RTP: A Transport Protocol for Real-Time Applications*, Internet RFC 1889, Janeiro.

Seetharamanan 1998 — Seetharamanan, K. (ed.) (1998). Special Issue: The CORBA Connection, *Comms. ACM*, Outubro, Vol. 41, No. 10.

session directory — *User Guide to sd (Session Directory)*.

Shannon 1949 — Shannon, C.E. (1949). Communication Theory of Secrecy Systems, *Bell System Technical Journal*, Vol. 28, No. 4, pgs. 656–715.

Shepler 1999 — Shepler, S. (1999). *NFS Version 4 Design Considerations*, Internet RFC 2624, Sun Microsystems, Junho.

Shih *et al*. 2002 — Shih, E., Bahl, P. e Sinclair, M. (2002). Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices. *Proceedings of the Eighth Annual ACM Conference on Mobile Computing and Networking*, Atlanta, GA, EUA, Setembro, pgs. 160–171.

Shirky 2000 — Shirky, C. (2000). What's P2P and What's Not, 11/24/2000. Internet publication.

Shoch e Hupp 1980 — Shoch, J.F. e Hupp, J.A. (1980). Measured performance of an Ethernet local network. *Comms. ACM*, Vol. 23, No. 12, pgs. 711–21.

Shoch e Hupp 1982 — Shoch, J.F. e Hupp, J.A. (1982). The 'Worm' programs – early experience with a distributed computation. *Comms. ACM*, Vol. 25, No. 3, pgs. 172–80.

Shoch *et al*. 1982 — Shoch, J.F., Dalal, Y.K. e Redell, D.D. (1982). The evolution of the Ethernet local area network. *IEEE Computer*, Vol. 15, No. 8, pgs. 10–28.

Shoch *et al*. 1985 — Shoch, J.F., Dalal, Y.K., Redell, D.D. e Crane, R.C. (1985). The Ethernet. Em *Local Area Networks: an Advanced Course, Lecture Notes in Computer Science*. No. 184, Springer-Verlag, pgs. 1–33.

Shrivastava *et al*. 1991 — Shrivastava, S., Dixon, G.N. e Parrington, G.D. (1991). An Overview of the Arjuna Distributed Programming System. *IEEE Software*, Janeiro, pgs. 66–73.

Singh 1999 — Singh, S. (1999). *The Code Book*. Londres: Fourth Estate.

Sinha e Natarajan 1985 — Sinha, M. e Natarajan, N. (1985). A Priority Based Distributed Deadlock Detection Algorithm. *IEEE Transactions on Software Engineering*. Vol. 11, No. 1, pgs. 67–80.

Spafford 1989 — Spafford, E.H. (1989). The Internet Worm: Crisis and Aftermath. *Comms. ACM*, Vol. 32, No. 6, pgs. 678–87.

Spasojevic e Satyanarayanan 1996 — Spasojevic, M. e Satyanarayanan, M. (1996). An Empirical Study of a Wide-Area Distributed File System, *ACM Transactions on Computer Systems*, Vol. 14, No. 2, Maio, pgs. 200–222.

Spector 1982 — Spector, A.Z. (1982). Performing remote operations efficiently on a local computer network. *Comms. ACM*, Vol. 25, No. 4, pgs. 246–60.

Spurgeon 2000 — Spurgeon, C.E. (2000). *Ethernet: The Definitive Guide*. O'Reilly.

Srikanth e Toueg 1987 — Srikanth, T. e Toueg, S. (1987). Optimal Clock Synchronization. *Journal ACM*. 34, 3 (Julho), pgs. 626–45.

Srinivasan 1995a — Srinivasan, R. (1995). RPC: *Remote Procedure Call Protocol Specification Version 2*. Sun Microsystems. Internet RFC 1831. Agosto.

Srinivasan 1995b — Srinivasan, R. (1995). *XDR: External Data Representation Standard*. Sun Microsystems. RFC 1832. Network Working Group. Agosto.

Srinivasan e Mogul 1989 — Srinivasan, R. e Mogul, J.D. (1989). Spritely NFS: Experiments with Cache-Consistency Protocols, 12th ACM Symposium on Operating System Principles, Litchfield Park, AZ, Dezembro, pgs. 45–57.

Srisuresh e Holdrege 1999 — Srisuresh, P. e Holdrege, M. (1999), *IP Network Address Translator (NAT) Terminology and Considerations*, Internet RFC 2663, Agosto.

Stajano 2002 — Stajano, F. (2002). *Security for ubiquitous computing*. Wiley.

Stajano e Anderson 1999 — Stajano, F. e Anderson, R. (1999). The Resurrecting Duckling: Security Issues for Adhoc Wireless Networks. *Proceedings 7th International Workshop on Security Protocols*, Springer-Verlag, pgs. 172–194.

Stallings 1998a — Stallings, W. (1998). *High Speed Networks – TCP/IP and ATM Design Principles*. Upper Saddle River, NJ: Prentice-Hall.

Stallings 1998b — Stallings, W. (1998). *Operating Systems*, terceira edição. Prentice-Hall International.

Stallings 1999 — Stallings, W. (1999). *Cryptography and Network Security – Principles and Practice*, segunda edição. Upper Saddle River, NJ: Prentice-Hall.

Steiner *et al.* 1988 — Steiner, J., Neuman, C. e Schiller, J. (1988). Kerberos: an authentication service for open network systems. Em *Proceedings Usenix Winter Conference*, Berkeley: CA.

Stelling *et al.* 1998 — Stelling, P., Foster, I., Kesselman, C., Lee, C. e von Laszewski, G. (1998). A Fault Detection Service for Wide Area Distributed Computations, *Proceedings 7th IEEE Symposium on High Performance Distributed Computing*, pgs. 268–78.

Stoica *et al.* 2001 — Stoica, I., Morris, R., Karger, D., Kaashoek, F. e Balakrishnan, H. (2001). Chord: A scalable Peer-To-Peer lookup service for internet applications. Em *ACM SIGCOMM*, Agosto.

Stojmenovic 2002 — Stojmenovic, I. (ed.) (2002). *Handbook of Wireless Networks and Mobile Computing*. Wiley.

Stoll 1989 — Stoll, C. (1989). *The Cuckoo's Egg: Tracking a Spy Through a Maze of Computer Espionage*. Nova York: Doubleday.

Stone 1993 — Stone, H. (1993). *High-performance Computer Architecture*, terceira edição. Addison-Wesley.

Sun 1989 — Sun Microsystems Inc. (1989). *NFS: Network File System Protocol Specification*. Internet RFC 1094.

Sun e Ellis 1998 — Sun, C. e Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. *Proceedings Conference on Computer Supported Cooperative Work Systems*, ACM Press, pgs. 59–68.

SWAP-CA 2002 — Shared Wireless Access Protocol (Cordless Access) Specification (SWAP-CA), Revision 2.0,1 The HomeRF Technical Committee, Julho de 2002.

Szalay e Gray 2001 — Szalay, A. e Gray, J. (2001) The World-Wide Telescope, *Science*, Vol. 293, pgs. 2037–2040.

Szalay e Gray 2004 — Szalay, A. e Gray, J. (2004). *Scientific Data Federation: The World-Wide Telescope*. Em Foster, I. e Kesselman, C. (eds), *The Grid 2*. São Francisco, CA: Morgan Kauffman.

Tanenbaum 2001 — Tanenbaum, A.S. (2001). *Modern Operating Systems*, segunda edição. Englewood Cliffs, NJ: Prentice-Hall.

Tanenbaum 2003 — Tanenbaum, A.S. (2003). *Computer Networks*, quarta edição. Prentice-Hall International.

Tanenbaum e van Renesse 1985 — Tanenbaum, A. e van Renesse, R. (1985). Distributed Operating Systems, *Computing Surveys*, ACM, Vol. 17, No. 4, pgs. 419–70.

Tanenbaum *et al.* 1990 — Tanenbaum, A.S., van Renesse, R., van Staveren, H., Sharp, G., Mullender, S., Jansen, J. e van Rossum, G. (1990). Experiences with the Amoeba Distributed Operating System. *Comms. ACM*, Vol. 33, No. 12, pgs. 46–63.

Taufer *et al.* 2003 — Taufer, M., Crowley, M., Karanicolas, J., Cicotti, P., Chien, A. e Brooks, L. (2003). *Moving towards desktop Grid solutions for large scale modelling in Computational Chemistry*. University of California, San Diego.

Terry *et al.* 1995 — Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M. e Hauser, C. (1995). Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pgs. 172–183.

TFCC — *IEEE Task Force on Cluster Computing*.

Thayer 1998 — Thayer, R. (1998). *IP Security Document Roadmap*, Internet RFC 2411, Novembro.

Thekkath *et al.* 1997 — Thekkath, C.A., Mann, T. e Lee, E.K. (1997). Frangipani: A Scalable Distributed File System, em *Proc. 16th ACM Symposium on Operating System Principles*, St Malo, França, Outubro, pgs. 224–237.

Tokuda *et al.* 1990 — Tokuda, H., Nakajima, T. e Rao, P. (1990). Real-time Mach: towards a predictable real-time system. Em *Proceedings USENIX Mach Workshop*, pgs. 73–82, Outubro.

Topolcic 1990 — Topolcic, C. (ed.) (1990). *Experimental Internet Stream Protocol*, Version 2. Internet RFC 1190.

Tzou e Anderson 1991 — Tzou, S.-Y. e Anderson, D. (1991). The performance of message-passing using restricted virtual memory remapping. *Software–Practice and Experience*, Vol. 21, pgs. 251–67.

van Renesse *et al.* 1989 — van Renesse, R., van Staveran, H. e Tanenbaum, A. (1989). The Performance of the Amoeba Distributed Operating System. *Software – Practice and Experience*, Vol. 19, No. 3, pgs. 223–34.

van Renesse *et al.* 1995 — van Renesse, R., Birman, K., Friedman, R., Hayden, M. e Karr, D. (1995). A Framework for Protocol Composition in Horus. *Proceedings PODC 1995*, pgs. 80–9.

van Renesse *et al.* 1996 — van Renesse, R., Birman, K. e Maffeis, S. (1996). Horus: a Flexible Group Communication System. *Comms. ACM*, Vol. 39, No. 4, pgs. 54–63.

van Steen *et al.* 1998 — van Steen, M., Hauck, F., Homburg, P. e Tanenbaum, A. (1998). Locating objects in wide-area systems. *IEEE Communication*, Vol. 36, No. 1, pgs. 104–109.

Vinoski 1998 — Vinoski, S. (1998). New Features for CORBA 3.0, *Comms. ACM*, Outubro de 1998, Vol. 41, No. 10. pgs. 44–52.

Vinoski 2002 — Vinoski, S. (2002). Putting the 'Web' into Web Services. *IEEE Internet Computing*. Julho–Agosto.

Vogels 2003 — Vogels, W. (2003). Web Services are not Distributed objects. *IEEE Internet Computing*. Nov–Dez 2003.

Vogt *et al.* 1993 — Vogt, C., Herrtwich, R.G. e Nagarajan, R. (1993). HeiRAT – The Heidelberg Resource Administration Technique: Design Philosophy and Goals. *Kommunikation in verteilten Systemen*, Munich, Informatik aktuell, Springer.

Volpano e Smith 1999 — Volpano, D. e Smith, G. (1999). Language Issues in Mobile Program Security. A ser publicado em *Lecture Notes in Computer Science*, Vol. 1419, pgs. 25–43. Springer.

von Eicken *et al.* 1995 — von Eicken, T., Basu, A., Buch, V. e Vogels, V. (1995). U-Net: a user-level network interface for parallel and distributed programming. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pgs. 40–53.

Wahl *et al.* 1997 — Wahl, M., Howes, T. e Kille, S. (1997). *The Lightweight Directory Access Protocol (v3)*. Internet RFC 2251.

Waldo 1999 — Waldo, J. (1999). The Jini Architecture for Network-centric Computing. *Comms. ACM*. Vol. 42, No. 7, pgs. 76–82.

Waldo *et al.* 1994 — Waldo, J., Wyant, G., Wollrath, A. e Kendall, S. (1994). A Note on Distributed Computing. Em Arnold *et al.* 1999, pgs. 307–26.

Hidden page

www.apple.com — Apple Computer Corp. Rendezvous technical resources.

www.bluetooth.com — The Official Bluetooth SIG Website.

www.butterfly.net — Butterfly.net, *The scalable, reliable and high performance online game platform. GoGrid.*

www.bxa.doc.gov — Bureau of Export Administration, US Department of Commerce, *Commercial Encryption Export Controls.*

www.cdk4.net — Coulouris, G., Dollimore, J. e Kindberg, T. (eds), *Distributed Systems, Concepts and Design: Supporting material.*

www.citrix.com — Citrix Corporation. *Server-based Computing White Paper.*

www.cren.net — Corporation for Research and Educational Networking, *CREN Certificate Authority.*

www.cryptopp.com — Crypto++® Library 5.2.1.

www.cs.york.ac.uk/dame — *Distributed Aircraft Maintenance Environment (DAME).*

www.cuseeme.com — CU-SeeMe Networks Inc. *Home page.*

www.doi.org — International DOI Foundation. *Páginas sobre identificadores de objetos digitais.*

www.dtnrg.org — Delay Tolerant Networking Research Group. *Home page.*

www.globexplorer.com — *Globexplorer, a maior biblioteca on-line do mundo sobre imagens aéreas e de satélite.*

www.handle.net — Handle system. *Home page.*

www.iana.org — Internet Assigned Numbers Authority. *Home page da IANA.*

www.ietf.org — Internet Engineering Task Force. *Internet RFC Index page.*

www.iona.com — Iona Technologies, *Orbix.*

www.ipnsig.org — InterPlaNetary Internet Project. *Home page.*

www.isoc.org — Robert Hobbes Zakon. *Hobbes' Internet Timeline.*

www.microsoft.com I — Microsoft Corporation. *Active Directory Services.*

www.microsoft.com II — Microsoft Corporation. *Windows 2000 Kerberos Authentication*, White Paper.

www.microsoft.com III — Microsoft Corporation. *NetMeeting Home page.*

www.neesgrid.org — *NEES Grid, Building the National Virtual Collaboratory for Earthquake Engineering.*

www.nfc-forum.org — Near Field Communication (NFC). *Forum Home page.*

www.omg.org — Object Management Group, *Índice para serviços CORBA*. OMG: Needham, MA.

www.opengroup.org — Open Group. *Portal to the World of DCE.*

www.openmobilealliance.org — Open Mobile Alliance. *Home page.*

www.openssl.org — OpenSSL Project. *OpenSSL: The Open Source toolkit for SSL/TLS.*

www.pgp.com — PGP. *Home page.*

www.reed.com — Read, D.P. (2000). *The End of the End-to-End Argument.*

www.rsasecurity.com — RSA Security Inc. *Home page.*

www.rsasecurity.com I — RSA Corporation (1997). *DES Challenge.*

www.rsasecurity.com II — RSA Corporation (2004). *RSA Factoring Challenge.*

www.rtj.org — *Real-Time for Java TM Experts Group.*

www.secinf.net — Network Security Library.

www.smart-its.org — The Smart-Its Project. *Home page.*

www.spec.org — *SPEC SFS97 Benchmark.*

www.upnp.org — Universal Plug and Play. *Home page.*

www.uscms.org — USCMS, *The Compact Muon Solenoid.*

www.verisign.com — Verisign Inc. *Home page.*

www.w3.org I — World Wide Web Consortium. *Home page.*

www.w3.org II — World Wide Web Consortium. *Páginas sobre a HyperText Markup Language.*

www.w3.org III — World Wide Web Consortium. *Páginas sobre Naming and Addressing.*

www.w3.org IV — World Wide Web Consortium. *Páginas sobre o HyperText Transfer Protocol.*

Hidden page

Índice

D

E

F

Hidden page

J

K

L

P

Q

R

S

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

SISTEMAS DISTRIBUÍDOS

CONCEITOS E PROJETO

George Coulouris
Jean Dollimore
Tim Kindberg

De telefones móveis à Internet, nossas vidas dependem cada vez mais dos sistemas distribuídos interligando computadores e outros dispositivos de maneira contínua e transparente. A quarta edição deste *best-seller* continua a fornecer uma fonte de material abrangente sobre os princípios e a prática de sistemas distribuídos e os novos e interessantes desenvolvimentos neles baseados usando vários estudos de caso modernos para ilustrar seu projeto e desenvolvimento.

Destaques da quarta edição:

- Três capítulos totalmente novos sobre sistemas peer-to-peer, serviços web e sistemas ubíquos
- Mais de 25 estudos de caso detalhados de sistemas conhecidos, oito deles novos, incluindo estudos sobre grade (grid computing), Cooltown, Bluetooth e a (in)segurança do protocolo WiFi WEP
- Abordagem atualizada da XML e suas extensões de segurança, o Advanced Encryption Standard e projeto de segurança para sistemas ubíquos

Este livro oferece aos estudantes de ciência da computação e de engenharia os conhecimentos necessários para projetar e manter software para aplicações distribuídas. Leitura valiosa também para engenheiros de software e projetistas de sistemas que queiram entender os novos e futuros desenvolvimentos do setor.

ISBN 978-85-60031-49-8

artmed®
EDITORA
RESPEITO PELO CONHECIMENTO

www.bookman.com.br