Objetos Distribuídos e Invocação Remota

Invocação de Métodos Remotos
com Java RMI
Cap.5 – Distributed Systems: Concepts and Design,
George Coulouris, Jean Dollimore, Tim Kindberg
4th Edition

Objetivos

- Estudar comunicação entre objetos distribuídos e a integração de invocação de métodos remotos dentro de uma linguagem de programação.

- Java, C++, C#

Objetivos

- Ser capaz para usar Java RMI to programar aplicações com objetos distribuídos.

- Estudar a extensão do modelo de programação baseada em eventos para aplicar à programas baseadosem eventos distribuídos.

Pontos a enfatizar

- Uma interface remota especifica os métodos de um objeto que está disponível para invocação por objetos remotos em outros programas.

Pontos a enfatizar

- Um objeto remoto é um objeto que pode ser invocado a partir de outro objeto em algum programa.

- Uma referência a objeto é um identificador para um objeto remoto que é usado para rferir-se a ele como o alvo de uma invocação remota e pode ser passado como um argumento ou resultado retornado de uma invocação remota.

Pontos a enfatizar

- Clientes necessitam de referências a objetos no sentido de invocar objetos remotos. Referências a objetos podem ser obtidas a partir de um binder ou como o resultado de uma invocação.

Pontos a enfatizar

- Um sistema RMI roda sobre um protocolo Request-Reply (see Coulouris, Section 4.4) e pode ser integrado transparentemente a uma linguagem de programação por meio de uma camada de middleware (Java RMI, CORBA, ou outro fornecedor) que provê clientes com proxies para objetos remotos e contém os detalhes de referências de objetos remotos, *marsharlling* e passagem de mensagem.

Pontos a enfatizar

- Sistemas baseados em eventos distribuídos usam o paradigma “publish-subscribe” (publica-assina), no qual um objeto gerando eventos publica os tipos de eventos que serão disponíveis para outros objetos.

Pontos a enfatizar

- Estes sistemas são úteis para comunicação entre componentes heterogêneos e sua natureza assíncrona permite publicadores e subscritores (assinantes) serem partes desacopladas, como componentes do sistema.

Pontos a enfatizar

- Programadores de clientes e servidores devem manipular exceções devido ao ambiente de distribuição.

Pontos a enfatizar

- O estudo-de-caso de Java ilustra um sistema RMI de uma única linguagem.
- Objetos remotos são passados por referência e objetos não-remotos são passados por valor.
- Classes podem ser downloaded de uma JVM para outra JVM.

Dificuldade Possível

- A relação entre referências a objeto local e remoto, necessitam ser entendidas porque programas de aplicação usam as primeiras, enquanto, interfaces remotas são definidas em termos da últimas.

O que é preciso saber ...

(i) A semântica de invocação de métodos locais e o uso de referências a objetos remotos.

(ii) O conceito de interface.

(iii) A manipulação de exceções.

Eventos e Notificações

Idéia de Eventos

- A idéia por trás do uso de eventos é aquela em que um objeto pode reagir a uma mudança ocorrendo em um outro objeto.

Notificações

- **Notificações de eventos** são essencialmente **assíncronas** e determinadas por seus receptores.

- Em particular, em aplicações interativas, **as ações que um usuário realiza sobre objetos**, por exemplo, por manipular um botão com um mouse ou entrar texto em uma caixa de texto via teclado, **são vistas como eventos que causam mudanças nos objetos** que mantém o estado da aplicação.

Notificações

- Os objetos que são responsáveis por disponibilizar uma visão do estado corrente, são notificados sempre que o estado muda.

Sistemas Baseados em Eventos Distribuídos

- Sistemas baseados em eventos distribuídos estendem o modelo de eventos locais, por permitir múltiplos objetos em diferentes localizações, serem notificados de eventos tomando lugar em objetos.

Sistemas Baseados em Eventos Distribuídos

- Sistemas baseados em eventos distribuídos usam o paradigma *publish-subscribe*, no qual um objeto que gera eventos publica o tipo de evento que ele tornará disponível para observação por outros objetos.

Sistemas Baseados em Eventos Distribuídos

- Objetos que desejam receber notificações de um objeto que tem publicado seus eventos, *subscribe* para os tipos de eventos que são de interesse deles.

Sistemas Baseados em Eventos Distribuídos

- Diferentes tipos de eventos podem, por exemplo, referir-se a diferentes métodos executados pelo objeto de interesse.

- Objetos que representam eventos são chamados *notificações*.

Sistemas Baseados em Eventos Distribuídos

- Notificações podem ser armazenadas, enviadas em mensagens, enfileiradas e aplicadas em uma variedade de ordens a diferentes coisas.

Sistemas Baseados em Eventos Distribuídos

- Quando um *publisher* experimenta um evento, *subscribers* que expressarem interesse naquele tipo de evento, receberão notificações.

- Subscrição para um tipo particular de evento é também chamado *registering interest* naquele tipo de evento.

Sistemas Baseados em Eventos Distribuídos

- **Eventos** e **notificações** podem ser usados em uma ampla variedade de diferentes aplicações.

- Por exemplo:
  1. comunicar uma forma adicionada a um desenho,
  2. uma modificação em um documento,
  3. o fato que uma pessoa entrou ou deixou uma sala,
  4. uma peça de equipamento ou um livro marcado eletronicamente está em uma nova localização (ou novo estado).

Sistemas Baseados em Eventos Distribuídos

- Os exemplos 3 e 4 são tornados possíveis com o uso de **crachás ativos** ou **dispositivos embutidos**.

Sistemas Baseados em Eventos Distribuídos

- Têm duas principais características:
  - **Heterogêneos**
  - **Assíncronos**

Heterogêneos

- Quando notificações de eventos são usadas como um meio de comunicação entre objetos, componentes em um sistema distribuido que não foram projetados para interoperar podem ser preparados para trabalhar juntos.

Heterogêneos

- Tudo o que é requerido é que **objetos gerando eventos publiquem os tipos de eventos que oferecem** , e que **outros objetos subscrevam o interesse em eventos e proporcionem uma interface para receber notificações.**

Assíncronas

- **Notificações** são enviadas **assincronamente** por **objetos gerando eventos** para **todos os objetos que estão subscritos a eles**.

Um Exemplo de Sistema baseado em Eventos e Notificações

Sistema de Sala de Negócios

Sistema Sala de Negócios

- Permite que negociantes usando computadores descubram as últimas informações sobre preços de mercado de estoques que eles precisam negociar.

- O preço de mercado para um único estoque nomeado é representado por um objeto com diversas variáveis de instância.

Sistema Sala de Negócios

- A informação chega na sala a partir de diversas fontes diferentes externas. Na forma de atualizações para algumas ou todas as variáveis de instâncias dos objetos representando os estoques, e é colecionada por objetos que chamamos provedores de informação.

Sistema Sala de Negócios

- Negociantes estão tipicamente interessados somente em seus estoques que são especializados.

- O sistema de sala de negócios poderia ser modelado por processos ou objetos com duas diferentes tarefas:

Sistema Sala de Negócios

- Um provedor de informação recebe continuamente novas informações de comércio, de fontes internas e aplicam essas aos objetos de estoques apropriados.

- Cada das atualizações para um objeto-estoque é considerada um evento.

Sistema Sala de Negócios

- O **objeto-estoque**, experimentando tais **eventos**, **notifica todos os negociantes que se subscreveram ao estoque** correspondente.

- Existe um provedor de informação separado para cada fonte externa.

Sistema Sala de Negócios

- Um processo negociante cria um objeto para representar cada estoque nomeado que o usuário (negociante) solicita para ter disponibilizado.

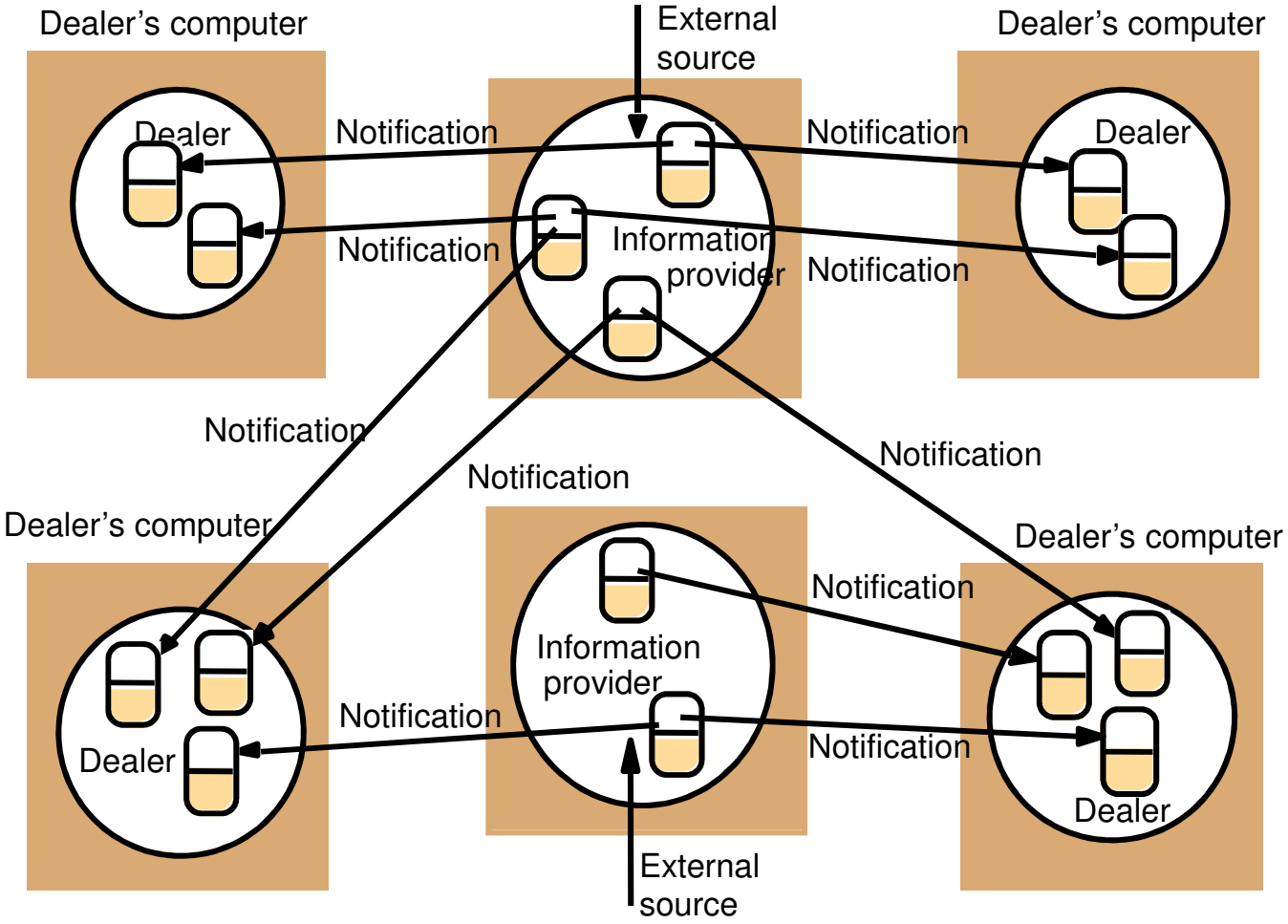
- Este objeto local (ao computador do negociante) se subscreve ao objeto representando o estoque no provedor de informação relevante.

Sistema Sala de Negócios

- Ele recebe toda a informação enviada a ele, em notificações e disponibiliza essa informação ao usuário (negociante).

- A comunicação de notificações é mostrada na Figura 5.10.

Figure 5.10
Sistema Sala de Negócios



Tipos de Eventos

- Um evento-fonte pode gerar eventos de um ou mais diferentes tipos.

- Cada evento tem atributos que especificam informação sobre aquele evento: nome, identificador do objeto que gerou ele, a operação, seus parâmetros e o tempo (ou um número de sequência).

Tipos de Eventos

- Tipos e atributos são usados ambos para subscrever a eventos e em notificações.

- Quando subscrevendo para um evento, o tipo de evento é especificado, algumas vezes modificado de acordo aos valores dos atributos.

Tipos de Eventos

- Sempre que um evento do tipo ocorre, correspondente aos atributos, as partes interessadas serão notificadas.

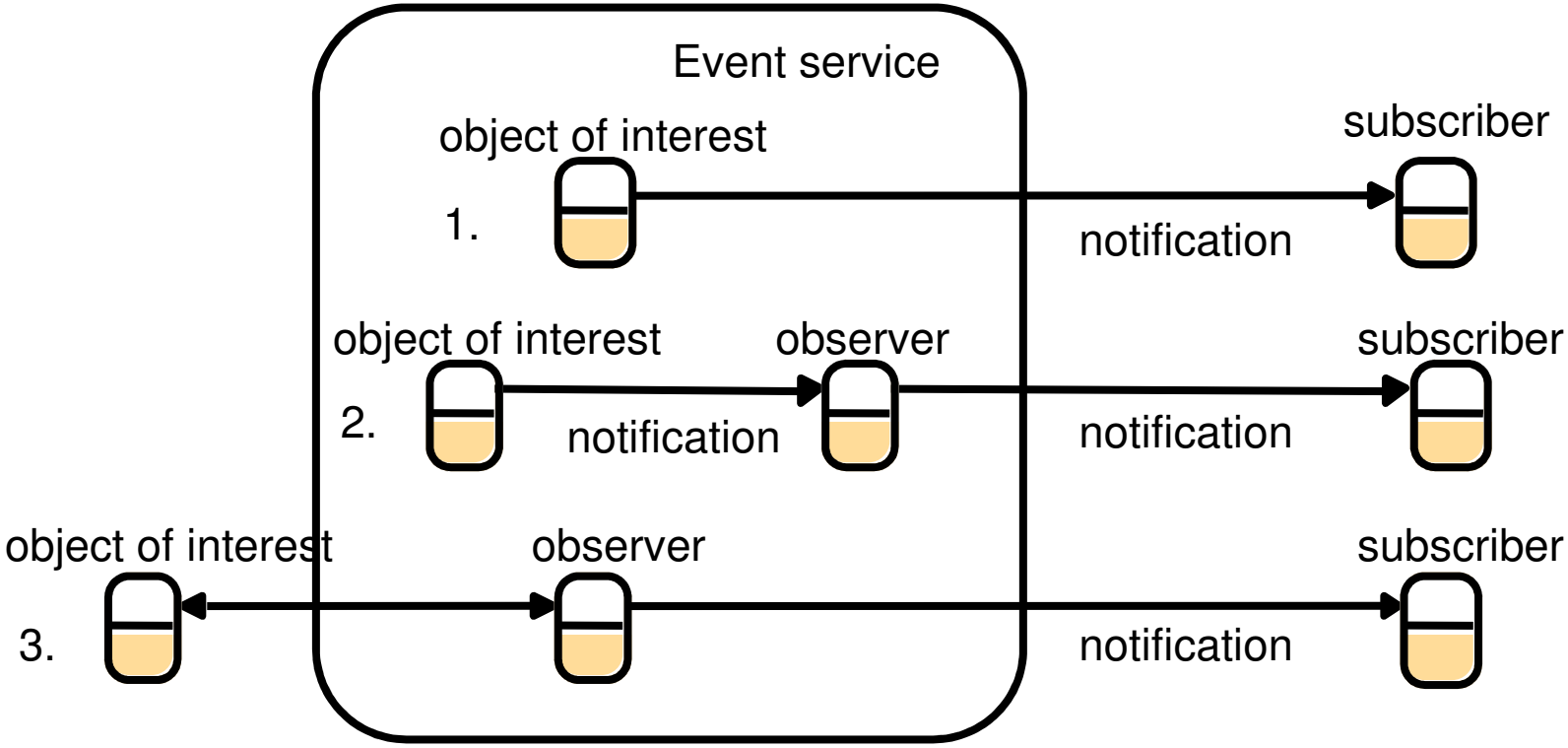
- No exemplo do Sistema Sala de Negócios, existe um tipo de evento (a chegada de atualização de um estoque) e os atributos poderiam especificar o nome de um estoque, seu preço corrente, e por fim, surgir ou falhar.

Tipos de Eventos

- Comerciantes podem especificar que eles estão interessados em todos os eventos relacionados a um estoque com um nome particular.

- A arquitetura que especifica os papéis dos participantes em notificação de eventos distribuídos. Figura 5.11.

Figure 5.11
Architecture for distributed event notification



Arquitetura para notificação de eventos distribuídos

- A Figura 5.11 mostra uma arquitetura que especifica os papéis exercidos pelos objetos que participam em sistemas baseados em eventos distribuídos.

Arquitetura para notificação de eventos distribuídos

- O principal componente é um serviço de eventos que mantém uma base de dados de eventos publicados e de subscrições de interesse (interesses de assinantes do serviço).

- Eventos em um objeto de interesse são publicados no serviço de eventos.

Arquitetura para notificação de eventos distribuídos

- Subscritores informam ao serviço de eventos sobre os tipos de eventos que eles estão interessados.

- Quando um evento ocorre em um objeto de interesse, uma notificação é enviada aos subscritores daquele tipo de evento.

Papéis de objetos participantes

- Objeto de interesse – Este é o objeto que experimenta mudanças de estado, como um resultado de suas operações sendo invocadas.

- Suas mudanças de estado podem ser de interesse de outros objetos.

- O objeto de interesse é considerado como parte do serviço de eventos, se ele transmite notificações.

Papéis de objetos participantes

- Evento – Um evento ocorre em objeto de interesse, como resultado do término de uma execução de método.

- Notificação – Uma notificação é um objeto que contém informação sobre um evento. Tipicamente contém o tipo do evento e seus atributos, os quais, geralmente, incluem a identidade do objeto de interesse, o método invocado, o tempo de ocorrência do evento ou um número de sequência.

Papéis de objetos participantes

- Subscritores (assinantes do serviço de eventos) – É um objeto que tem subscrito para alguns tipos de eventos em um outro objeto. Ele recebe notificações sobre tais eventos.

Papéis de objetos participantes

- Objeto Observador – O principal propósito de um objeto observador é desacoplar um objeto de interesse de seus subscritores.

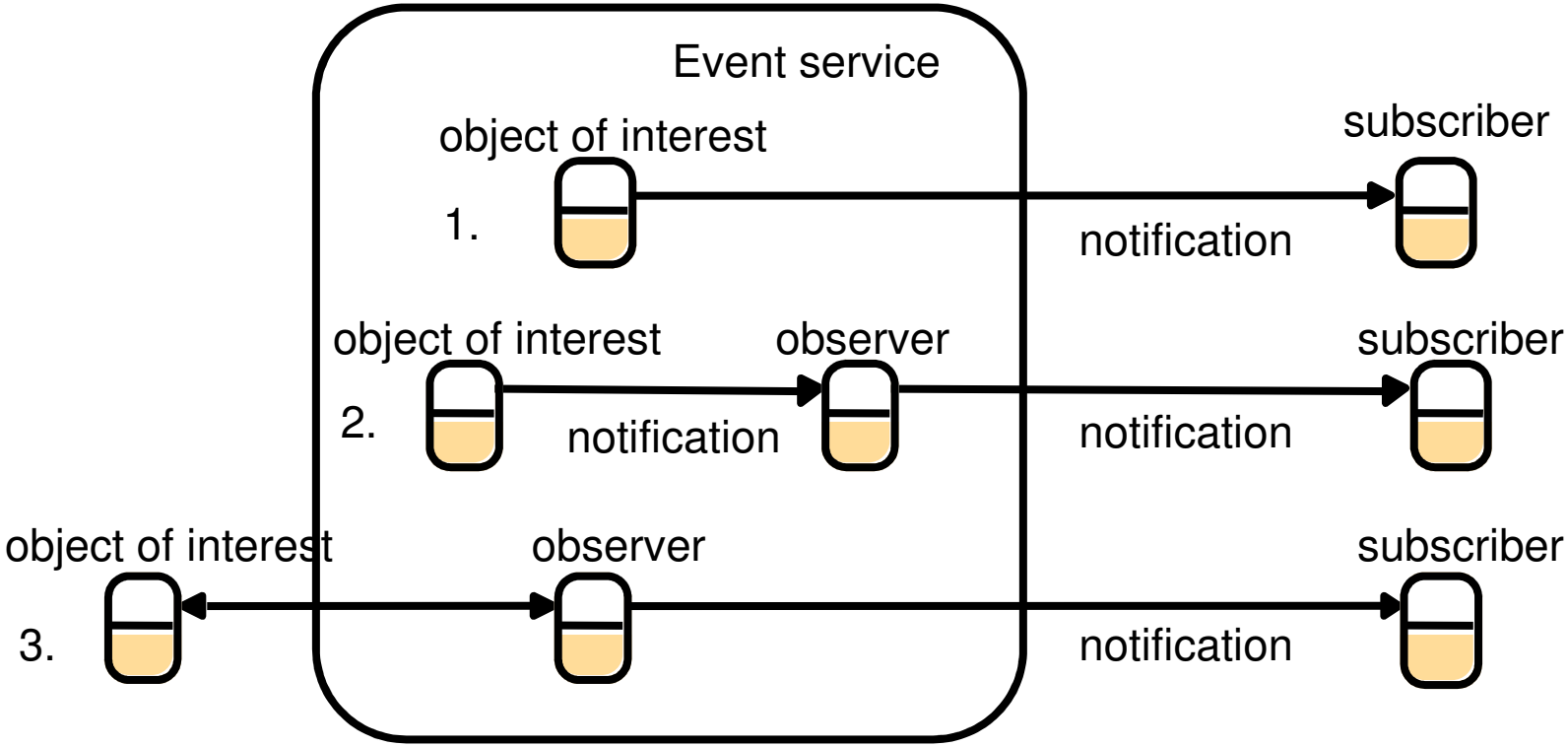
- Um objeto de interesse pode ter muitos diferentes subscritores. Subscritores podem diferir nos tipos de eventos que eles estão interessados. Ou aqueles compartilhando os mesmos requisitos quanto ao tipo, podem diferir nos valores de atributos que são de interesse.

Papéis de objetos participantes

- Publicador – Este é o objeto que declara que ele gerará notificações de tipos particulares de eventos.

- Um publicador pode ser um objeto de interesse ou um observador.

Figure 5.11
Architecture for distributed event notification



A Figura 5.11 mostra três casos: Caso 1

- Um objeto de interesse dentro do serviço de eventos, mas sem um observador. Notificações são enviadas diretamente ao subscritores. Veja a figura 5.11.

A Figura 5.11 mostra três casos: Caso 2

- Um objeto de interesse dentro do serviço de eventos, mas com um observador.

- O objeto de interesse envia notificações via o observador aos subscristores.

A Figura 5.11 mostra três casos: Caso 3

- Um objeto de interesse fora do serviço de eventos.

- Neste caso, um observador consulta o objeto de interesse, no sentido de descobrir quando eventos ocorrem. O observador envia notificações aos subscritores.

Java RMI

- Java RMI estende o modelo de objetos de Java para prover suporte para objetos distribuídos.

- Interfaces remotas em Java devem ser especificadas.

Interfaces Remotas

```
import java.rmi.*
Import java.util.???
public interface <nome> extends Remote {
  ... ... ( ... ... ) throws RemoteException;
  ... ... ( ... ... ) throws RemoteException;
}

... ...
... ...

public interface <nome> extends Remote {
  ... ... ( ... ... ) throws RemoteException;
  ... ... ( ... ... ) throws RemoteException;
}
```

Interfaces Remotas

- Interfaces remotas são definidas por estender uma interface chamada `Remote` provida no pacote `java.rmi`.

- Os métodos devem lançar `RemoteException`, porém exceções específicas da aplicação podem também ser lançadas.

Interfaces Remotas

- Um ponto importante a notar é que ambos, objetos locais e objetos remotos, podem aparecer como argumentos e resultados em uma interface remota.

- Objetos remotos são sempre denotados pelo nome de suas interfaces remotas.

Figure 5.12 – Exemplo de Interfaces Remotas Java *Shape* and *ShapeList*

```
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;
}
```

Como objetos ordinários e remotos são passados como argumentos

- Passagem de parâmetros e resultados

  Em Java RMI, os parâmetros de um método são assumidos ser parâmetros de entrada e o resultado como um único parâmetro de saída.

Como objetos ordinários e remotos são passados como argumentos

- Java Serialization

  Usado para montagem de argumentos e resultados (marshalling) em Java RMI.

- Qualquer objeto que é serializável – implementa a `interface Serializable` – pode ser passado como um argumento ou resultado de método em Java RMI.

Como objetos não-remotos e remotos são passados como argumentos

- Todos os tipos primitivos e objetos remotos são serializáveis.

- Classes for argumentos e valores de resultados são *downloaded* ao recipiente (método onde serão utilizados) pelo sistema Java RMI, onde necessário.

Passando objetos remotos

- **Referência de objeto remoto**

  A noção de referência a objeto é estendida para permitir qualquer objeto em Java RMI, ter uma referência a objeto remoto.

  É um identificador que pode ser usado através um sistema distribuído para se referir a um particular objeto remoto único.

Passando objetos remotos

- **Referência de objeto remoto**

  Sua representação é geralmente diferente de referências locais.

  Referências a objetos remotos são análogas às referências locais no que:

  -- o objeto remoto para receber uma invocação de método é especificado pelo invocador como uma referência a objeto remoto;

Passando objetos remotos

-- referências a objetos remotos podem ser passados como argumentos e resultados de invocações de métodos.

-- programas clientes geralmente requerem um meio de obter a referência remota para ao menos um dos objetos remotos no servidor.

Passando objetos remotos

-- Um *binder* em um sistema distribuído é um serviço separado que mantém uma tabela contendo mapeamentos de nomes textuais para referências a objetos remotos.

-- É usada pelos servidores para registrar seus objetos remotos pelo nome e pelos clientes para procurá-los. O *binder* em Java RMI é o RMIregistry.

Passando objetos remotos

- Exemplo, na Figura 5.12, linha 2, o valor de retorno do método `newShape` é definido como `Shape` – uma interface remota.

- Quando uma referência a objeto remoto é recebida, essa pode ser usada para fazer chamadas RMI sobre o objeto remoto para o qual essa se refere.

Passando objetos não-remotos

- Todos os objetos não-remotos serializáveis são copiados e passados por valor.

- Por exemplo, na Figura 5.2 (linhas 1 e 2) o argumento de `newShape` e o valor de retorno de `getAllState` são ambos do tipo `GraphicalObject`, o qual é serializável e passado por valor.

Passando Objetos

- Assim, no exemplo da Figura 5.2, um programa-cliente usa o método `newShape` para passar uma instância `g` de `GraphicalObject` ao servidor.

Passando Objetos

- O servidor instancia um objeto remoto do tipo `Shape` contendo o estado do `GraphicalObject` e retorna uma referência a objeto remoto de `Shape` para o programa-cliente, pois `Shape` vem de uma interface remota.

Passando Objetos

- Os argumentos e valores de retorno em invocações remotas são serializados para um *stream* como:

  1. Sempre que um objeto implementa uma interface remota é serializado, ele é substituido pela sua referência a objeto remoto, a qual contém o nome de sua classe (classe do objeto remoto).

Passando Objetos

2. Quando qualquer objeto é serializado, sua informação de classe é anotada com a localização da classe (como uma URL), habilitando a classe ser *downloaded* pelo receptor.

Baixando Classes

- Java é projetada para permitir classes ser baixadas de uma JVM para outra.

- Isto é importante para objetos distribuídos que se comunicam por meio de invocações remotas.

Baixando Classes

- Objetos não-remotos são passados por valor e objetos remotos são passado por referência, como argumentos e resultados das invocações de métodos remotos (RMI).

- Se um receptor não possui a classe de um objeto passado por valor, sei código é baixado automaticamente.

Baixando Classes

- Similarmente, se um receptor de uma referência objeto remoto não possui a classe de um proxy, seu código é baixado automaticamente.

Baixando Classes

- **Isto tem duas vantagens:**

  1. Não existe a necessidade de todo usuário guardar o mesmo conjunto de classes em seu ambiente de trabalho.

  2. Ambos, programas cliente e servidor podem tornar transparente o uso de instâncias de novas classes sempre que elas são necessárias e adicionadas.

Baixando Classes

- Como um exemplo, considere o programa "whiteboard" e suponha que sua implementação inicial de `GraphicalObject` não permite texto.

Baixando Classes

- Então um cliente com um objeto textual pode implementar uma subclasse de `GraphicalObject` que trata com texto e passar uma instância ao servidor, como um argumento do método `newSape`.

Baixando Classes

- Assim, outros clientes podem recuperar a instância usando o método `getAllState`.

- O código da nova classe será baixada automaticamente, a partir do primeiro cliente para o servidor e então, para outros clientes quando necessário.

RMIregistry

- O RMIregistry é o *binder* para o Java RMI.
- Uma instância de RMIregistry deve rodar em todo computador que hospede objetos remotos.

RMIregistry

- RMIregistry mantém uma tabela que mapeia nomes de objetos no estilo *URL names*, como *(//computerName:port/objectName)* à referências para objetos remotos, onde computerName e port referem-se à localização do RMIregistry.

- Exemplo: “`//bruno.ShapeList`”

RMIregistry

- Se são omitidos, o computador local (localhost) e a port default são assumidos.

- Suas interfaces oferecem os métodos mostrados na Figura 5.13, nos quais as exceções não estão listadas – todos os métodos podem lançar `RemoteException`.

RMIregistry

- O serviço de RMIregistry não é um serviço de *binding* amplo, que exista em toda a rede.

- Clientes devem dirigir suas consultas *lookup* à hosts particulares.

Figure 5.13
The ***Naming*** class of Java **RMIregistry**

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.14, line 2.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 15.16 line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the RMIregistry.

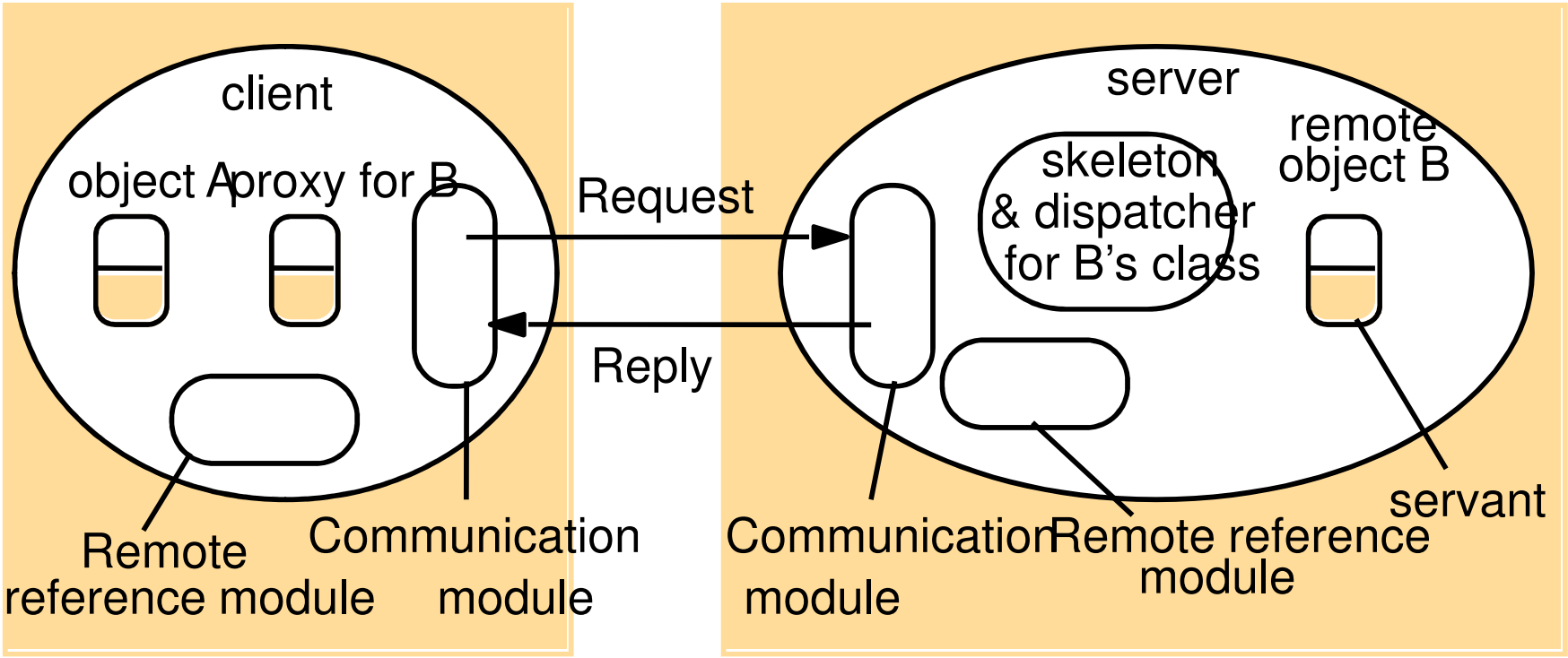
Observações

- O *server* consiste de um método `main` e uma classe *servant* para implementar cada uma de suas interfaces remotas.

- Um *servant* é uma instância de uma classe que provê o corpo de um objeto remoto.

- Ele manipula os *requests* passados pelos *skeletons*.

Figure 5.7
The role of proxy and skeleton in remote method invocation



Observações

- O método `main` de server cria uma instância de `ShapeListServant` e binds (liga) ela a um nome no RMIregistry. O valor ligado é a referência a objeto remoto e seu tipo é o tipo de sua interface remota – `ShapeList`.

Observações

- As duas classes *servants* são `ShapeListServant` que implementa `ShapeList` interface, e `ShapeServant` que implementa a interface `Shape`.

- UnicastRemoteObject provê objetos remotos que vivem somente enquanto o processo no qual eles são criados, vive.

Observações

- Na Figura 5.15 (linha 2), o método de `newShape` pode ser chamado um método factory, porque ele permite ao cliente solicitar a criação de um *servant*.

- O método `main` do servidor *server* necessita criar um gerenciador de segurança para aplicar proteção apropriada para um RMI server.

Observações

- O gerenciador de segurança `RMISecurityManger` é provido.

- Ele protege os recursos locais para garantir que as classes que são carregadas a partir de sites remotos não possam ter qualquer efeito sobre recursos como arquivos.

Observações

- Qualquer programa-cliente necessita dar inicio, usando um *binder* para procurar uma referência remota.

- O cliente estabelece um gerenciador de segurança e então procura uma referência a objeto remoto, usando a operação `lookup` no RMIregistry.

Figure 5.14
Java class *ShapeListServer* with *main* method

```
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());}
     }
}
```

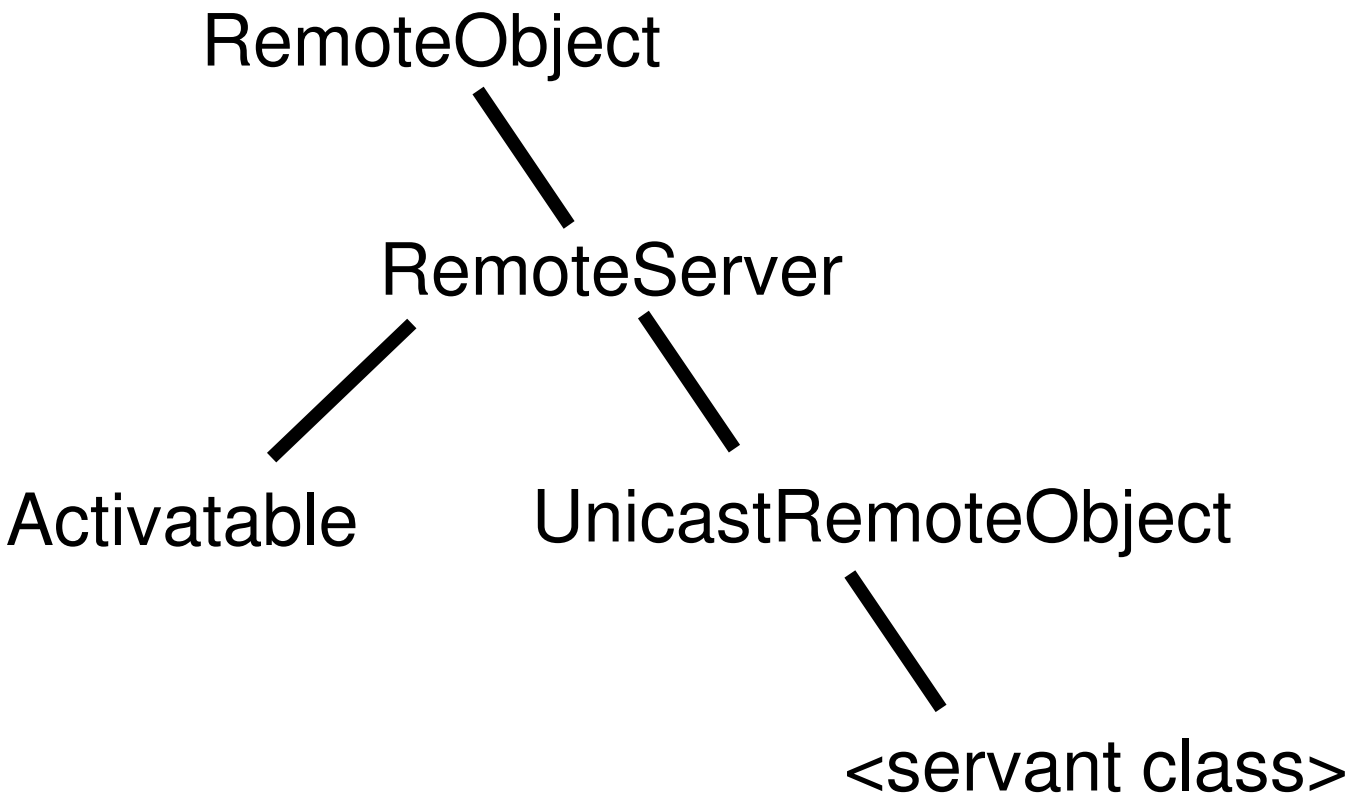
Figure 5.15
Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
      private Vector theList;              // contains the list of Shapes          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 { ... }
}
```

Figure 5.16
Java client of *ShapeList*

```
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")   ;
          Vector sList = aShapeList.allShapes();                          2
     } catch(RemoteException e) {System.out.println(e.getMessage());
     }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
   }
}
```

Figure 5.17
Classes e subclasses em Java RMI



Herança em Java RMI

- A única classe que o programador precisa estar ciente é `UnicastRemoteObject`.
- Toda classe servant simples necessitar ser estendida de `UnicastRemoteObject`.
- `UnicastRemoteObject` estende a classe abstrata `RemoteServer`.

Herança em Java RMI

- `RemoteServer` provê versões abstratas dos métodos requeridos pelos servidores remotos.
- `Activatable` provê objetos *activatable* objetos.
- `RemoteServer` é subclasse de `RemoteObject`, que tem uma variável de instância retendo a referência a objeto remoto e provê os seguintes métodos:

Herança em Java RMI

-- `equals:` este método compara referência a objetos remotos.

-- `toString`: este método dá a referência a objeto remoto como uma string.

-- `readObject`, `writeObject:` estes métodos deserializa/serializa objetos remotos.