Unidade 5

Comunicação entre Processos

- Pipes
- Sockets
- RMI
- CORBA

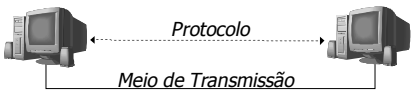
Comunicação entre Processos

- Processos e threads interagem para trabalhar conjuntamente em um sistema
  - Trocam dados / mensagens
  - Utilizam os serviços de comunicação fornecidos pela máquina e pelo S.O.
  - Seguem protocolos de comunicação para que possam entender uns aos outros

2

Comunicação entre Processos

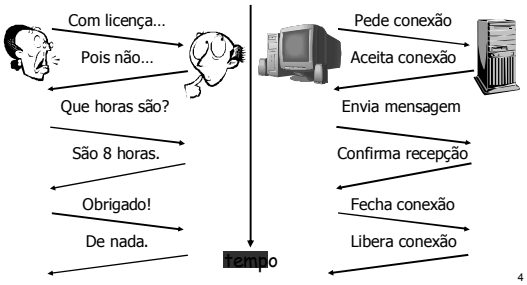
- Protocolos
  - Estabelecem caminhos virtuais de comunicação entre processos / threads
  - Duas entidades precisam usar os mesmos protocolos para trocar informações



3

Comunicação entre Processos

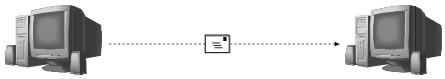
- Protocolos



4

Comunicação entre Processos

- Serviços de comunicação
  - Serviço sem Conexão: cada unidade de dados é enviada independentemente das demais



  - Serviço com Conexão: dados são enviados através de um canal de comunicação

5

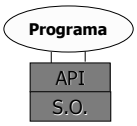
Comunicação entre Processos

- Características dos serviços de comunicação entre processos/threads:
  - Abrangência: local ou remota
  - Participantes: 1 → 1, 1 → N ou M → N
  - Tamanho das mensagens: fixo ou variável; limitado ou não
  - Sincronismo: comunicação síncrona, assíncrona ou semi-síncrona

6

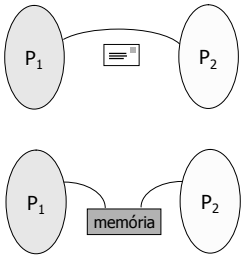
Comunicação entre Processos

- APIs de comunicação
  - Permitem que aplicações troquem dados
  - Fornecem primitivas de comunicação que podem ser chamadas a partir do código
  - Provêem acesso aos serviços de comunicação, que podem assim ser usados pelas aplicações



Comunicação entre Processos

- APIs de Comunicação de Sist. Operacionais
  - Mecanismos fornecidos pelos S.O.'s permitem enviar mensagens (trechos de memória) de um processo a outro
  - Alguns S.O.'s permitem que sejam criadas áreas de memória compartilhadas entre dois ou mais processos



Comunicação entre Processos

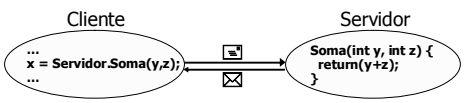
- Exemplos de APIs de comunicação:
  - Pipes: canais de comunicação locais
  - Sockets: portas de comunicação locais ou de rede (versão segura: SSL)
  - Suportes de RPC (*Remote Procedure Call*): permitem chamar procedimentos/métodos remotamente (ex.: RMI, CORBA, COM, …)
  - Canais de eventos: permitem notificar threads e processos dos eventos ocorridos no sistema (Ex.: JMS, CORBA Notification Service, …)
  - …

Comunicação entre Processos

- Mecanismos de IPC nativos do UNIX
  - *Pipes*
  - *Sockets* (comunicação local ou remota)
- Mecanismos de IPC nativos do Windows
  - *Pipes* e *Mailslots*
  - WinSock (comunicação local ou remota)
  - Microsoft RPC (comunicação local ou remota)
  - Memória compartilhada: *File Mapping* e *Dynamic Data Exchange* (DDE)
  - *Object Linking and Embedding* (OLE)

Comunicação entre Processos

- RPC – Chamada Remota de Procedimento
  - Segue o modelo Cliente/Servidor
  - Muito usado na interação entre objetos
  - Objeto servidor possui interface com métodos que podem ser chamados remotamente
  - Objetos clientes usam serviços de servidores

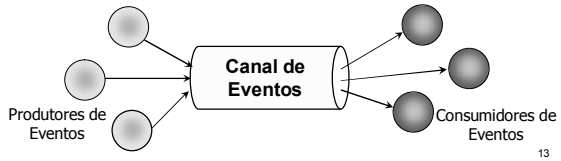


Comunicação entre Processos

- RPC – Características
  - Em geral as requisições são ponto-a-ponto e síncronas
  - Dados são tipados
    - Parâmetros da requisição
    - Retorno do procedimento/método
    - Exceções
  - Um objeto pode ser cliente e servidor em momentos diferentes

Comunicação entre Processos

- Notificação de Eventos
  - Eventos ocorridos são difundidos por produtores e entregues a consumidores
  - Canal de eventos permite o desacoplamento – produtor e consumidor não precisam se conhecer



13

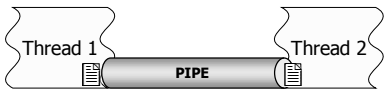
Comunicação entre Processos

- Notificação de Eventos – Características
  - Envio de eventos é completamente assíncrono
    - Produtor não precisa aguardar fim do envio
    - Evento é armazenado no canal de eventos
  - Comunicação pode ser feita através de UDP *multicast* ou fazendo múltiplos envios *unicast* com TCP, UDP ou com um suporte de RPC
  - Os eventos podem ter tamanho fixo ou variável, limitado ou ilimitado
  - Eventos podem ser tipados ou não

14

Pipes

- Pipes = canais de comunicação
  - Duas threads podem trocar dados por um pipe → comunicação de 1 para 1
  - Threads devem rodar na mesma máquina virtual → comunicação local



15

Pipes

- Disponíveis em Java através das classes PipedInputStream e PipedOutputStream
- Principais métodos:
  - Criação dos Pipes:
    is = new PipedInputStream();
    os = new PipedOutputStream(is);
  - Envio e recepção de bytes / arrays de bytes:
    is.read(dado)
    os.write(dado); os.flush();
  - Exceção: java.io.IOException

16

Pipes

- Uso de Pipes com fluxos (*streams*) de dados
  - Criação do fluxo:
    in = new java.io.DataInputStream(is);
    out = new java.io.DataOutputStream(os);
  - Envio e recepção de dados:
    <*tipo*> var = in.read<*tipo*>();
    out.write<*tipo*>(var);
    onde <*tipo*> = Int, Long, Float, Double, etc.

17

Pipes

```
public class Producer extends Thread {
  private DataOutputStream out;
  private Random rand = new Random();
  public Producer(PipedOutputStream os) {
    out = new DataOutputStream(os);   }
  public void run() {
    while (true)
      try {
        int num = rand.nextInt(1000);
        out.writeInt(num);
        out.flush();
        sleep(rand.nextInt(1000));
      } catch(Exception e) { e.printStackTrace(); }
  }
}
```

8

Pipes

```
public class Consumer extends Thread {
   private DataInputStream in;
   public Consumer(PipedInputStream is) {
      in = new DataInputStream(is); }
   public void run() {
      while(true)
         try {
            int num = in.readInt();
            System.out.println("Número recebido: " + num);
         } catch(IOException e) {  e.printStackTrace(); }
   }
}
```

9

Pipes

```
public class PipeTest {
   public static void main(String args[]) {
     try {
        PipedOutputStream out = new PipedOutputStream();
        PipedInputStream in = new PipedInputStream(pout);

        Producer prod = new Producer(out);
        Consumer cons = new Consumer(in);

        prod.start();
        cons.start();
     } catch (IOException e) { e.printStackTrace(); }
   }
}
```

10

Pipes

- Uso de Pipes com fluxos de objetos
  - Criação do fluxo:
    in = new java.io.ObjectInputStream(is);
    out = new java.io.ObjectOutputStream(os);
  - Envio e recepção de objetos:
    <*classe*> obj = (<*classe*>) in.readObject();
    out.writeObject(obj);
    onde <*classe*> = classe do objeto lido/enviado
  - Envio e recepção de Strings:
    String str = in.readUTF();
    out.writeUTF(str);

21

Sockets

- Socket
  - Abstração que representa uma porta de comunicação bidirecional associada a um processo
- Principais Tipos de Socket
  - Socket Datagrama: envia/recebe datagramas sem criar conexão; usa protocolo UDP
  - Socket Multicast: recebe as mensagens endereçadas a um grupo; usa UDP multicast
  - Socket Stream: estabelece uma conexão com outro socket; usa protocolo TCP

22

Sockets

- Operações com Sockets Datagrama
  - Criar um socket datagrama:
    DatagramSocket s = new DatagramSocket(porta);
  - Criar pacotes de dados para envio:
    DatagramPacket pack = new DatagramPacket(msg, tamanho, destino, porta);
  - Enviar dados: s.send(pack);
  - Criar pacotes de dados para recepção:
    DatagramPacket pack = new DatagramPacket(msg,tam);
  - Receber dados: s.receive(pack);
  - Ler dados do pacote: pack.getData()

23

Sockets

- Sockets Datagrama – Envio

```
try {
   DatagramSocket socket = new DatagramSocket();
   InetAddress destino = InetAddress.getByName(
      "127.0.0.1");
   String mensagem = "Hello";
   byte[] dados = mensagem.getBytes();
   int porta = 1234;
   DatagramPacket pacote = new DatagramPacket(
      dados, dados.length, destino, porta);
   socket.send(pacote);
} catch (SocketException e) { e.printStackTrace();
} catch (IOException e) {e.printStackTrace(); }
```

24

Sockets

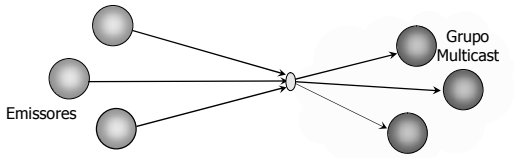
- Sockets Datagrama – Recepção

```
try {
   int porta = 1234;
   DatagramSocket socket = new DatagramSocket(porta);
   byte[] dados = new byte[100];
   DatagramPacket pacote = new DatagramPacket(
      dados, dados.length);
   socket.receive(pacote);
   String mensagem = new String(pacote.getData(), 0,
      pacote.getLength() );
   System.out.println("Mensagem: " + mensagem);
} catch (SocketException e) { e.printStackTrace();
} catch (IOException e) {e.printStackTrace(); }
```

25

Sockets

- Sockets Multicast
  - Permitem o envio simultâneo de datagramas a grupos de destinatários
  - Grupos multicast são identificados por endereços IP de 224.0.0.0 a 239.255.255.255



26

Sockets

- Sockets Multicast
  - Vários emissores podem mandar mensagens para o grupo (ou seja, mensagens vão de X emissores → Y receptores)
  - Emissor não precisa fazer parte do grupo para enviar mensagens ao grupo, e nem precisa saber quem são os seus membros; basta conhecer o endereço IP do grupo
  - O receptor entra em um grupo (se torna um membro do grupo) e passa a receber as mensagens destinadas ao grupo

27

Sockets

- Sockets Multicast – Envio

```
try {
   DatagramSocket socket = new DatagramSocket();
   InetAddress grupo = InetAddress.getByName(
      "200.1.2.3");
   String mensagem = "Hello";
   byte[] dados = mensagem.getBytes();
   int porta = 1234;
   DatagramPacket pacote = new DatagramPacket(
      dados, dados.length, grupo, porta);
   socket.send(pacote);
} catch (SocketException e) { e.printStackTrace();
} catch (IOException e) { e.printStackTrace(); }
```

28

Sockets

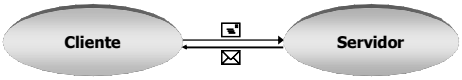
- Sockets Multicast – Recepção

```
try {
   int porta = 1234;
   MulticastSocket msocket = new MulticastSocket(porta);
   InetAddress grupo = InetAddress.getByName(
      "230.1.2.3");
   msocket.joinGroup(grupo);
   byte[] dados = new byte[100];
   DatagramPacket pacote = new DatagramPacket(
      dados, dados.length);
   msocket.receive(pacote);
   System.out.println("Mensagem: " +
      new String(pacote.getData(), 0, pacote.getLength()));
} catch (Exception e) {e.printStackTrace(); }
```

29

Sockets

- Sockets Stream
  - Estabelecem canais de comunicação entre aplicações, permitindo troca de dados pela rede
  - Adotam o paradigma cliente-servidor
    - Cliente: pede para conectar ao servidor
    - Servidor: aguarda conexões dos clientes



30

Sockets

- Operações com Sockets Stream no Servidor
  - Criar um socket servidor:
    ServerSocket s = new ServerSocket(porta, maxClientes);
  - Aguardar conexão: Socket c = s.accept();
  - Obter nome do host conectado:
    String host = c.getInetAddress().getHostName();
  - Criar fluxos de comunicação:
    ObjectInputStream in = new
    ObjectInputStream(c.getInputStream());
    ObjectOutputStream out = new
    ObjectOutputStream(c.getOutputStream());
  - Fechar conexão: in.close(); out.close(); c.close();

31

Sockets

- Operações com Sockets Stream no Cliente
  - Criar um socket cliente:
    Socket c = new Socket(InetAddress.
    getByName("*servidor.com* "), porta);
  - Criar fluxo, enviar e receber dados, e fechar: idem ao servidor
- Exceções geradas
  - SocketException
  - UnknownHostException
  - IOException

32

Sockets

- Sockets Stream – Servidor

```
try {
   ServerSocket s = new ServerSocket(1234, 10);
   Socket c = s.accept();
   ObjectOutputStream out = new ObjectOutputStream(
      c.getOutputStream());
   output.flush();
   ObjectInputStream input = new ObjectInputStream(
      c.getInputStream() );
   String mensagem = (String) input.readObject();
   String resposta = "Olá Cliente";
   output.writeObject(mensagem);
   output.flush();
} catch (Exception e) { e.printStackTrace(); }
```

33

Sockets

- Sockets Stream – Cliente

```
try {
   Socket socket = new Socket(InetAddress.getByName(
      "127.0.0.1"), 1234);
   ObjectOutputStream out = new ObjectOutputStream(
      socket.getOutputStream());
   output.flush();
   ObjectInputStream input = new ObjectInputStream(
      socket.getInputStream() );
   String mensagem = "Olá Servidor";
   output.writeObject(mensagem);
   output.flush();
   String resposta = (String) input.readObject();
} catch (Exception e) { e.printStackTrace(); }
```

34

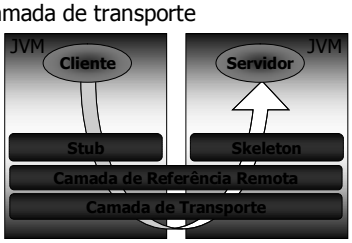
Java RMI

- Java RMI (*Remote Method Invocation*)
  - Fornece um suporte simples para RPC/RMI
  - Permite que um objeto Java chame métodos de outro objeto Java rodando em outra JVM
  - Solução específica para a plataforma Java

35

Java RMI

- Arquitetura RMI
  - *Stub* e *Skeleton*
  - Camada de referência remota
  - Camada de transporte



36

Java RMI

- *Stub*
  - Representa o servidor para o cliente
  - Efetua serialização e envio dos parâmetros
  - Recebe a resposta do servidor, desserializa e entrega ao cliente
- *Skeleton*
  - Recebe a chamada e desserializa os parâmetros enviados pelo cliente
  - Faz a chamada no servidor e retorna o resultado ao cliente

37

Java RMI

- Camada de Referência Remota
  - Responsável pela localização dos objetos nas máquinas da rede
  - Permite que referências para um objeto servidor remoto sejam usadas pelos clientes para chamar métodos
- Camada de Transporte
  - Cria e gerencia conexões de rede entre objetos remotos
  - Elimina a necessidade do código do cliente ou do servidor interagirem com o suporte de rede

38

Java RMI

- Dinâmica da Chamada RMI
  - O servidor, ao iniciar, se registra no serviço de nomes (RMI *Registry*)
  - O cliente obtém uma referência para o objeto servidor no serviço de nomes e cria a *stub*
  - O cliente chama o método na *stub* fazendo uma chamada local
  - A *stub* serializa os parâmetros e transmite a chamada pela rede para o *skeleton* do servidor

39

Java RMI

- Dinâmica da Chamada RMI (cont.)
  - O *skeleton* do servidor recebe a chamada pela rede, desserializa os parâmetros e faz a chamada do método no objeto servidor
  - O objeto servidor executa o método e retorna um valor para o *skeleton*, que o desserializa e o envia pela rede à *stub* do cliente
  - A *stub* recebe o valor do retorno serializado, o desserializa e por fim o repassa ao cliente

40

Java RMI

- Serialização de Dados
  - É preciso serializar e deserializar os parâmetros da chamada e valores de retorno para transmiti-los através da rede
  - Utiliza o sistema de serialização de objetos da máquina virtual
    - Tipos predefinidos da linguagem
    - Objetos serializáveis: implementam interface java.io.serializable

41

Java RMI

- Desenvolvimento de Aplicações com RMI
  - Devemos definir a interface do servidor
    - A interface do servidor deve estender java.rmi.Remote ou uma classe dela derivada (ex.: UnicastRemoteObject)
    - Todos os métodos da interface devem prever a exceção java.rmi.RemoteException
    - O Servidor irá implementar esta interface
  - *Stubs* e *skeletons* são gerados pelo compilador RMI (rmic) com base na interface do servidor

42

Java RMI

- Java RMI – Interface do Servidor
  - É implementada pelo servidor
  - É chamada pelo cliente
  - Estende a interface java.rmi.Remote
  - Todos os métodos devem prever a exceção java.rmi.RemoteException

```
public interface HelloWorld extends java.rmi.Remote {
   public String hello() throws java.rmi.RemoteException;
}
```

43

Java RMI

- Java RMI – Implementação do Servidor

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloServer extends UnicastRemoteObject
   implements HelloWorld {
   public HelloServer() throws RemoteException {super();}
   public String hello() throws RemoteException {
      return "Hello!!!";
   }
   public static void main(String[] args) {
      try {
         HelloServer servidor = new HelloServer();
         Naming.rebind("//localhost/HelloWorld", servidor);
      } catch (Exception e) { e.printStackTrace(); }
   }
}
```

44

Java RMI

- Java RMI – Implementação do Cliente

```
public class HelloClient {
   public static void main(String[] args) {
      try {
         HelloWorld servidor = (HelloWorld)
            java.rmi.Naming.lookup("//localhost/HelloWorld");
         String msg = servidor.hello();
         System.out.println("Mensagem do Servidor: "+msg);
      } catch (Exception e) { e.printStackTrace(); }
   }
}
```

45

Java RMI

- RMI/IIOP
  - A partir do Java 2.0, o RMI passou a permitir a utilização do protocolo IIOP (Internet Inter-ORB Protocol) do CORBA
  - IIOP também usa TCP/IP, mas converte os dados para um formato padrão (seralização ou *marshalling*) diferente do Java RMI
  - Com RMI/IIOP, objetos Java podem se comunicar com objetos CORBA, que podem ser escritos em outras linguagens

46

CORBA

- CORBA
  - *Common Object Request Broker Architecture*
  - Padrão definido pela OMG (*Object Management Group*), que reúne cerca de 800 empresas interessadas no desenvolvimento de software orientado a objetos
  - Permite a interação entre objetos distribuídos
  - Fornece um suporte completo para desenvolver aplicações distribuídas orientadas a objetos

47

CORBA

- CORBA proporciona total transparência para os Objetos Distribuídos
  - Transparência de Linguagem
    - Objetos implementados em várias linguagens
    - Interfaces de objetos descritas em IDL
  - Transparência de S.O. e Hardware
    - Implementado em várias plataformas: Windows, Linux, Solaris, AIX, HP-UX, etc.
  - Transparência de Localização dos Objetos
    - Referências podem apontar para objetos em outras máquinas

48

CORBA

- IDL (*Interface Definition Language*)
  - Usada para descrever as interfaces de objetos
  - Linguagem puramente declarativa, sem nenhuma estrutura algorítmica
  - Sintaxe e tipos de dados baseados em C/C++
  - Define seus próprios tipos de dados, que são mapeados nos tipos de dados de cada linguagem de programação suportada
  - Mapeada para diversas linguagens
    - C, C++, Java, Delphi, COBOL, Python, ADA, Smalltalk, LISP, ...

49

CORBA

- Compilador IDL
  - Gera todo o código responsável por:
    - Fazer a comunicação entre objetos
    - Fazer o mapeamento dos tipos de dados definidos em IDL para a linguagem usada na implementação
    - Fazer as conversões de dados necessárias na comunicação (serialização/ *marshalling* dos dados)

50

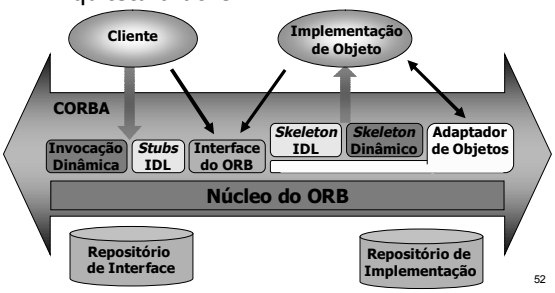
CORBA

- Interação entre objetos no CORBA
  - Segue o modelo Cliente-Servidor
    - Cliente: faz requisições em objs. remotos
    - Implementação de objeto: implementa os serviços descritos na sua interface

51

CORBA

- Arquitetura do ORB



52

CORBA

- Invocação de Operações Remotas
  - Formas de invocação:
    - Estática: através de stubs e skeletons gerados com base na descrição da interface do servidor em IDL; ou
    - Dinâmica: através da interface de invocação dinâmica do CORBA
  - O servidor não percebe o tipo de invocação utilizado na requisição pelo cliente

53

CORBA

- *Stubs* IDL
  - Geradas pelo compilador IDL com base na descrição da interface do objeto
  - Usadas na invocação estática
  - O cliente conhece a interface, o método e os parâmetros em tempo de compilação
- *Skeletons* IDL
  - Geradas pelo compilador IDL
  - Interface estática para os serviços (métodos) remotos executados pelo servidor

54

CORBA

- Interface de Invocação Dinâmica (DII)
  - Permite que o cliente construa uma invocação em tempo de execução
  - Elimina a necessidade das *Stubs* IDL
  - Com a DII, novos tipos de objetos podem ser adicionados ao sistema em tempo de execução
  - O cliente especifica o objeto, o método e os parâmetros com uma seqüência de chamadas
  - O servidor continua recebendo as requisições através de seu skeleton IDL

55

CORBA

- Adaptador de Objetos
  - Interface entre o suporte e os objetos servidores
  - Transforma um objeto escrito em uma linguagem qualquer em um objeto CORBA
  - Usado para geração e interpretação de referências de objetos, invocação dos *Skeletons*, ativação e desativação de implementações de objetos, etc.
  - Existem vários tipos de adaptador de objeto

56

CORBA

- *Portable Object Adapter* (POA)
  - Adaptador padrão: torna o servidor portável entre implementações diferentes
  - Abstrai a identidade do objeto da sua implementação
  - Implementa políticas de gerenciamento de *threads*:
    - uma *thread* por objeto
    - uma *thread* por requisição
    - grupo (*pool*) de *threads*
    - *etc.*

57

CORBA

- Interoperabilidade
  - CORBA garante a interoperabilidade entre objetos que usem diferentes implementações de ORB
  - Solução adotada a partir do CORBA 2.0
    - Padronizar o protocolo de comunicação e o formato das mensagens trocadas
    - Foi definido um protocolo geral, que é especializado para vários ambientes específicos

58

CORBA

- Interoperabilidade (cont.)
  - Protocolo Inter-ORB Geral (GIOP)
    - Especifica um conjunto de mensagens e dados para a comunicação entre ORBs
  - Especializações do GIOP
    - Protocolo Inter-ORB para Internet (IIOP): specifica como mensagens GIOP são transmitidas numa rede TCP/IP
    - Protocolos Inter-ORB para Ambientes Específicos: permitem a interoperabilidade do ORB com outros ambientes (ex.: DCE, ATM nativo, etc.)

59

CORBA

- Interoperabilidade entre CORBA e Java RMI
  - Une as vantagens das duas tecnologias
  - Applets, Servlets e aplicações Java podem ser clientes CORBA usando RMI/IIOP ou ORB Java
  - Mapeamentos: IDL → Java e Java → IDL
- Interoperabilidade entre CORBA e DCOM
  - Permite que objetos DCOM acessem serviços oferecidos por objetos CORBA e vice-versa
  - Bridges convertem mensagens entre os ambientes, integrando o DCOM a plataformas nas quais ele não está disponível

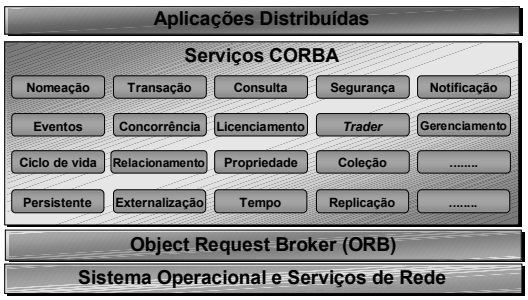
60

CORBA

- Serviços CORBA
  - Coleção de serviços em nível de sistema
  - Oferecem funcionalidades básicas para utilizar e implementar os objetos de aplicações distribuídas
  - Especificam as interfaces e casos de uso, deixando a implementação de lado
  - Estendem ou complementam as funcionalidades do ORB
  - Independentes da aplicação

61

CORBA



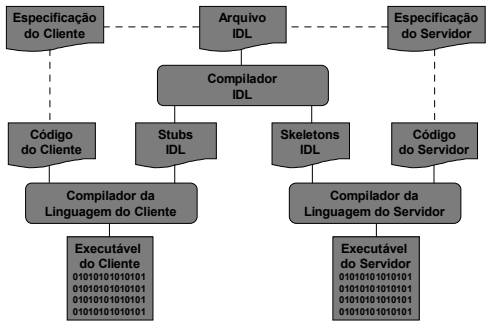
62

CORBA

- Desenvolvimento de Aplicações
  - Passos p/ desenvolver um servidor CORBA
    - Definir a interface IDL do servidor
    - Compilar a IDL (gerar o *skeleton*)
    - Implementar os métodos do servidor
    - Compilar
    - Executar
  - Passos p/ desenvolver um cliente CORBA
    - Obter a IDL do servidor
    - Compilar a IDL do servidor (gerar *stub*)
    - Implementar o código do cliente
    - Compilar
    - Executar

63

CORBA



64

CORBA

- Elementos de uma especificação IDL
  - Módulos
    - Definições de Tipos, Constantes e Exceções
    - Interfaces
      - Definições de Tipos, Constantes e Exceções
      - Atributos
      - Operações (métodos)

65

CORBA

- Módulos
  - Define escopo das declarações em seu interior

```
module ident {
  // declarações de tipos, constantes e exceções
  // declarações de interfaces
};
```

- Interfaces
  - Descrevem a interface de um objeto CORBA

```
interface ident : interfaces_herdadas {
  // declarações de tipos, constantes e exceções
  // declarações de atributos e operações
};
```

66

CORBA

- Exemplo: Banco

```
module Banco {
  // ...
  interface AutoAtendimento {
    // ...
  };
  interface CaixaEletronico: AutoAtendimento {
    // ...
  };
};
```

67

CORBA

- Tipos Básicos IDL
  - boolean: tipo booleano, valor TRUE ou FALSE
  - char: caractere de 8 bits, padrão ISO Latin-1
  - short: inteiro curto com sinal; -2^{15} a $2^{15}-1$
  - long: inteiro longo com sinal; -2^{31} a $2^{31}-1$
  - unsigned short: inteiro curto sem sinal; 0 a $2^{16}-1$
  - unsigned long: inteiro longo sem sinal; 0 a $2^{32}-1$
  - float: real curto, padrão IEEE 754/1985
  - double: real longo, padrão IEEE 754/1985
  - octet: 1 byte, nunca convertido na transmissão
  - any: corresponde a qualquer tipo IDL

68

CORBA

- Tipos Básicos IDL (cont.)
  - Object: corresponde a um objeto CORBA
  - long long: inteiro de 64 bits; -2^{63} a $2^{63}-1$
  - unsigned long long: inteiro de 64 bits sem sinal; 0 a $2^{64}-1$
  - long double: real duplo longo padrão IEEE; base com sinal de 64 bits e 15 bits de expoente
  - wchar: caractere de 2 bytes, para suportar diversos alfabetos
  - fixed<n,d>: real de precisão fixa; n algarismos significativos e d casas decimais

69

CORBA

- Arrays
  - Array de tamanho fixo:
    tipo ident [*tamanho*];
  - Array de tamanho variável sem limite máximo (tamanho efetivo definido durante a execução)
    sequence <*tipo*> *ident* ;
  - Array de tamanho variável c/ tamanho máx.:
    sequence <*tipo*,*tamanho*> *ident* ;
  - Obs.: *tipo* = qualquer tipo IDL
    ident = identificador único da variável
    tamanho = número inteiro

70

CORBA

- Strings
  - Seqüência de caracteres sem limite de tamanho:
    string *ident* ; // seqüência de char's
    wstring *ident* ; // seqüência de wchar's
  - Seqüência de caracteres com tamanho máximo definido:
    string <*tamanho*> *ident* ;
    wstring <*tamanho*> *ident* ;

71

CORBA

- Estrutura de dados (registro)
  - Tipo composto por vários campos

```
struct ident {
  tipo ident ;
  // mais campos ...
};
```

- Lista enumerada
  - Lista com valores de um tipo

```
enum ident { /*lista de valores*/ };
```

72

CORBA

- União discriminada
  - Tipo composto com seleção de campo por cláusula switch/case; o seletor deve ser tipo IDL inteiro, char, boolean ou enum

```
union ident switch (seletor){
  case valor : tipo ident ;
  // mais campos ...
  default: tipo ident ;
};
```

73

CORBA

- Exceções
  - São estruturas de dados retornadas por uma operação para indicar que uma situação anormal ocorreu durante sua execução
  - Cada exceção possui um identificador e uma lista de campos que informam as condições nas quais a exceção ocorreu

```
exception ident {
  // lista de campos
};
```

  - Exceções padrão do CORBA: CONCLUDED_YES, CONCLUDED_NO, CONCLUDED_MAYBE

74

CORBA

- Exemplo: Banco

```
module Banco {
  const string NomeBanco = "UFSC";
  const string Moeda = "R$";
  enum Aplicacao { poupanca, fundoAcoes, rendaFixa };
  struct Transacao {
    long    data;
    string  descricao;
    double  valor;
  };
  sequence <Transacao> Transacoes;
  exception ContaInvalida { long conta; };
  exception SaldoInsuficiente { double saldo; };
  // ...
};
```

75

CORBA

- Atributos
  - São dados de um objeto que podem ter seu valor lido e/ou modificado remotamente
  - Declarados usando a sintaxe:
    attribute *tipo ident* ;
  - Caso a palavra-chave readonly seja utilizada, o valor do atributo pode ser somente lido
    readonly attribute *tipo ident* ;

76

CORBA

- Operações
  - Declaradas em IDL na forma:
    tipo ident (/* lista de parâmetros */)
    [raises (*exceção* [, ...])];
  - Parâmetros são separados por vírgulas e seguem a forma: {in|out|inout} *tipo ident*
    - in: parâmetro de entrada
    - out: parâmetro de saída
    - inout: parâmetro de entrada e saída
  - Apenas operações acessíveis remotamente devem ser declaradas na IDL do servidor

77

CORBA

- Operações Oneway (assíncronas)
  - Declaradas em IDL na forma:
    oneway void ident (/* lista de parâmetros */);
  - Uma operação oneway é assíncrona, ou seja, o cliente não aguarda seu término.
  - Operações oneway não possuem retorno (o tipo retornado é sempre void) e as exceções possíveis são somente as padrão.

78

CORBA

- Exemplo: Banco

```
interface AutoAtendimento {
   readonly attribute string boasVindas;
   double saldo (in long conta) raises (ContaInvalida);
   void extrato (in long conta, out Transacoes t,
      out double saldo) raises (ContaInvalida);
   void transferencia (in long origem, in long destino,
      in double v) raises (ContaInvalida, SaldoInsuficiente);
   void investimento (in long c, in Aplicacao aplic,
      in double v) raises (ContaInvalida, SaldoInsuficiente);
};
interface CaixaEletronico : AutoAtendimento {
   void saque (in long conta, in double valor)
      raises (ContaInvalida, SaldoInsuficiente);
};
```

79

CORBA

- O código pode ser implementado em qualquer linguagem mapeada para IDL

```
public class AutoAtendimentoImpl                    Java
  extends AutoAtendimentoPOA {

  public String boas_vindas() {
    return "Bem-vindo ao Banco";
  }
  ...
};
```

```
class auto_atendimentoImpl:                         C++
  auto_atendimentoPOA { ... };

char* banco_auto_atendimentoImpl::boas_vindas()
  throws (CORBA::SystemException) {
  return CORBA::string_dup("Bem-vindo ao Banco");
}
```

80

CORBA

- Mapeamento IDL para Java
  - Define como são representados em Java os elementos definidos em IDL
- Regras de Mapeamento
  - Módulos são mapeados em packages Java
  - Interfaces, Exceções e Arrays e Strings são idênticos em Java
  - Sequências são mapeadas como Arrays
  - Constantes se tornam atributos estáticos
  - Estruturas de dados, Unions e Enums são mapeadas como classes Java

81

CORBA

Tipo IDL	Equivalente em Java
boolean	boolean
char	char
wchar	char
short	short
long	int
long long	long
unsigned short	short
unsigned long	int
unsigned long long	long
float	float
double	double
long double	(não disponível)
octet	byte
any	CORBA.Any
fixed	Math.BigDecimal
Object	CORBA.Object

82

CORBA

- Mapeamento de Atributos IDL para Java
  - É criado um método com o nome do atributo
  - Se o atributo não for readonly, um método de mesmo nome permite modificar o seu valor
- Mapeamento de Operações IDL para Java
  - São criados métodos na interface correspondente, com os mesmos parâmetros e exceções
  - Contexto inserido no final da lista de parâmetros

83

ORBA

- Implementação do Servidor
  - O servidor deve iniciar o ORB e o POA, e disponibilizar sua referência para os clientes
  - Referências podem ser disponibilizadas através do serviço de nomes, impressas na tela ou escritas em um arquivo acessado pelos clientes usando o sistema de arquivos distribuído, um servidor HTTP ou FTP
  - Feito isso, o servidor deve ficar ouvindo requisições e as executando

84

CORBA

- Exemplo: Implementação do Banco

```
package Banco;
import org.omg.CORBA.*;

public class AutoAtendimentoImpl
    extends AutoAtendimentoPOA {

  public String boasVindas () {
    return "Bem-vindo ao banco "+ NomeBanco.value;
  }

  public Valor saldo (long conta) throws ContaInvalida {
      return BancoDB.getConta(conta).getSaldo();
  }
  // demais métodos...
}
```

85

CORBA

- Exemplo: Servidor do Banco

```
package Banco;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.PortableServer.*;

public class Servidor {
  public static void main(String args[]) {
    try{
      // Inicializa o ORB
      ORB orb = ORB.init(args, null);
      // Instancia o servidor
      AutoAtendimentoImpl aa= new AutoAtendimentoImpl();
      // Localiza e ativa o POA
      POA rootpoa = POAHelper.narrow( orb.
        resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();
```

CORBA

- Exemplo: Servidor do Banco (cont.)

```
      // Gera a referência remota do servidor
      AutoAtendimento ref = AutoAtendimentoHelper.
        narrow(rootpoa.servant_to_reference(aa));
      // Obtém a referência do servidor de nomes
      NamingContextExt ns = NamingContextExtHelper.
        narrow(orb.resolve_initial_references("NameService"));
      // Registra o servidor
      ns.rebind(ns.to_name("Banco"), ref);
      // Aguarda chamadas dos clientes
      orb.run();
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

87

CORBA

- Implementação do Cliente
  - Um cliente deve sempre iniciar o ORB e obter uma referência para o objeto servidor
  - Referências podem ser obtidas através do serviço de nomes, da linha de comando ou lendo um arquivo que contenha a referência
  - De posse da referência, o cliente pode chamar os métodos implementados pelo servidor

88

CORBA

- Implementação do Cliente

```
import Banco.*;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

public class Cliente {
  public static void main(String args[]) {
    try {
      // Inicializa o ORB
      ORB orb = ORB.init(args, null);
      // Obtém a referência do serviço de nomes
      NamingContextExt ns = NamingContextExtHelper.
        narrow(orb.resolve_initial_references("NameService"));
      // Obtém a referência do servidor do banco
      AutoAtendimento server = auto_atendimentoHelper.
        narrow(ns.resolve_str("Banco"));
```

CORBA

- Implementação do Cliente (cont.)

```
      // Imprime mensagem de boas-vindas
      System.out.println(server.boasVindas());
      // Obtém o numero da conta
      System.out.print("Entre o número da sua conta: ");
      String conta = new java.io.BufferedReader(new
        java.io.InputStreamReader(System.in)).readLine();
      // Imprime o saldo atual
      System.out.println("Saldo da conta: R$"
        + server.saldo(Integer.parseInt(conta)));
    } catch (ContaInvalida e) {
      System.out.println("Conta " + e.conta + " não existe.");
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```