Unidade 5

Comunicação entre Processos

- Pipes
- Sockets
- Java RMI

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

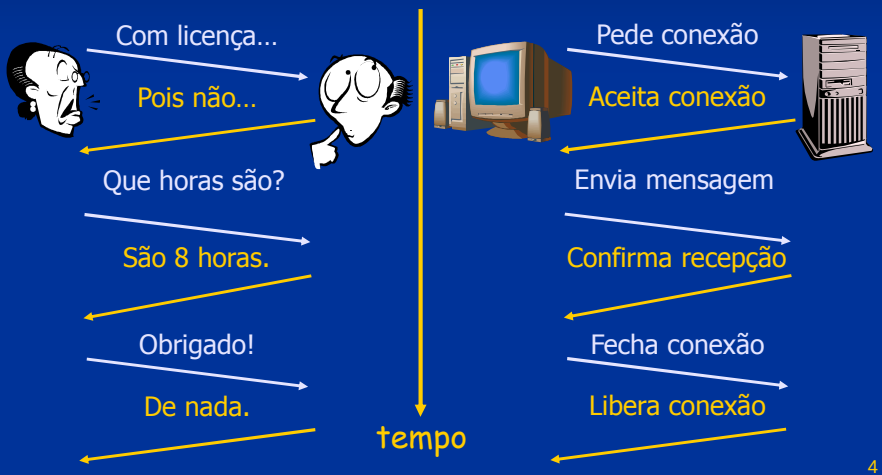
Comunicação entre Processos

- 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

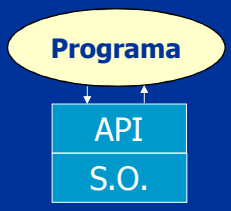
Comunicação entre Processos

- 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

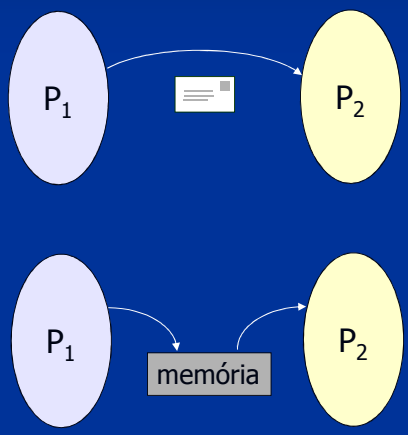
- APIs de comunicação
  - Permitem que aplicações troquem dados
  - Fornecem primitivas de comunicação que podem ser chamadas a partir do código
  - Provêem acesso aos serviços de comunicação, que podem assim ser usados pelas aplicações



7

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



88

Comunicação entre Processos

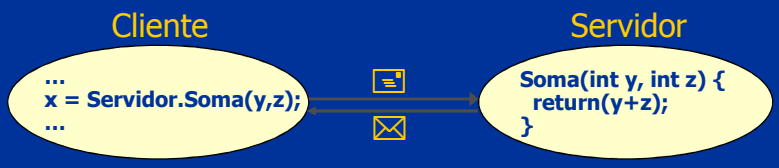
- Exemplos de APIs de comunicação:
  - Pipes: canais de comunicação locais
  - Sockets: portas de comunicação locais ou de rede (versão segura: SSL)
  - Suportes de RPC (*Remote Procedure Call*): permitem chamar procedimentos/métodos remotamente (ex.: RMI, CORBA, COM, …)
  - Canais de eventos: permitem 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



11

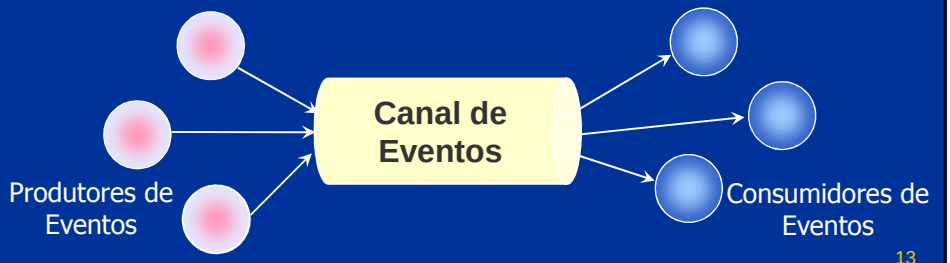
Comunicação entre Processos

- 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

12

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

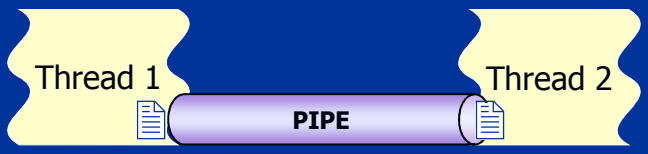
Comunicação entre Processos

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

18

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(out);

       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

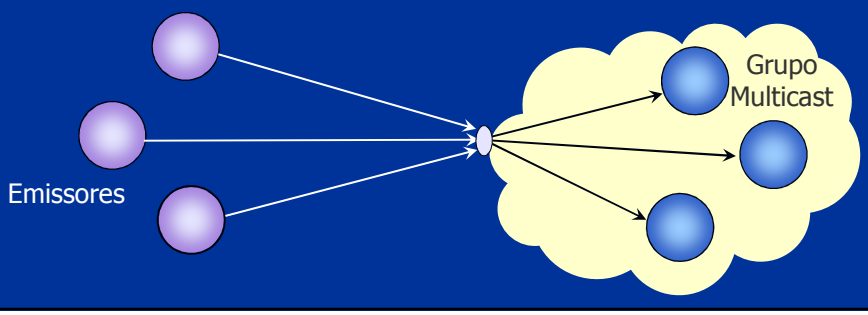
- Sockets Datagrama – Recepção

```
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(
      "230.1.2.3");
   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 output = new ObjectOutputStream(
      c.getOutputStream());
   output.flush();
   ObjectInputStream input = new ObjectInputStream(
      c.getInputStream() );
   String mensagem = (String) input.readObject();
   String resposta = "Olá Cliente";
   output.writeObject(resposta);
   output.flush();
} catch (Exception e) { e.printStackTrace(); }
```

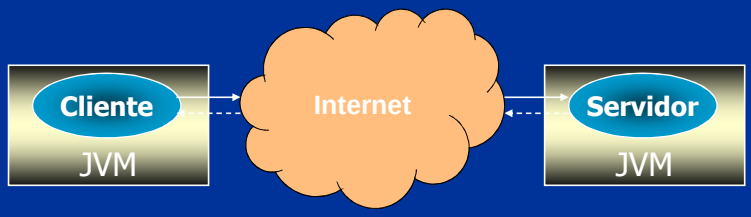

Sockets

- Sockets Stream – Cliente

```
try {
   Socket socket = new Socket(InetAddress.getByName(
      "127.0.0.1"), 1234);
   ObjectOutputStream output = 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(); }
```

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

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(); }
```

36

Java RMI

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

37

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

38

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

39

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

40

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 serializa 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

41

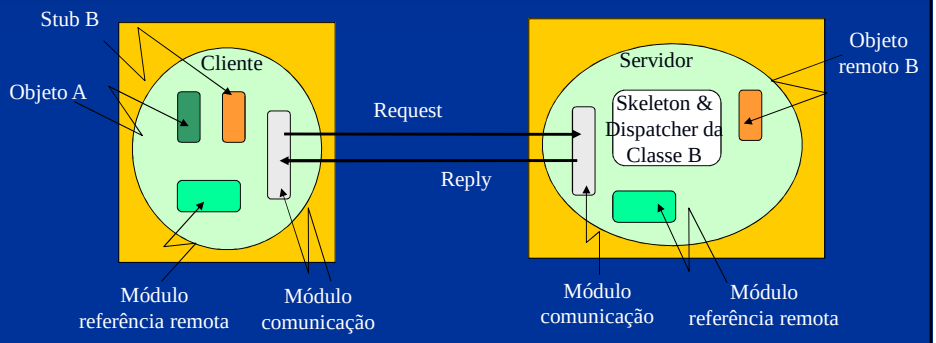
Java RMI

- 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

42

Implementação de RMI

- Vários objetos estão envolvidos com o suporte a RMI.



43

Exemplo: Cliente/Servidor RMI

- Receita de bolo:
  - Definir uma interface remota que descreve como o cliente e o servidor se comunicam um com o outro;
  - Definir a classe servidor que implementa a interface remota e os acessos ao rmiregistry.
  - Definir o aplicativo cliente que utiliza uma referência de interface remota para interagir com a implementação de servidor da interface. Isto é, um objeto da classe que implementa a interface remota;
  - Compilar e executar o servidor e o cliente.

44

Exemplo: Cliente/Servidor 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 automaticamente.

45

Exemplo: Cliente/Servidor RMI

- Passo 1: definindo a interface remota
  - Descrever os métodos remotos que serão oferecidos ao cliente (RMIServidorInt.java);

```
import java.rmi.*;
public interface HelloWorld extends Remote {
    public void mensagem(String str) throws RemoteException;
    public String hello() throws RemoteException;
}
```

- Passo 2: Implementar a interface do servidor declarada no passo 1.

46

HelloServer.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
public class HelloServer implements HelloWorld {
  public HelloServer() {}
  public static void main(String[] args) {
    try {
      HelloServer server = new HelloServer();
      HelloWorld stub =
              (HelloWorld) UnicastRemoteObject.exportObject(server, 0);
      Registry registry = LocateRegistry.getRegistry();
      registry.bind("Hello", stub);
      System.out.println("Servidor pronto");
```

7

HelloServer.java (continuação)

```
    } catch (Exception ex) {
      ex.printStackTrace();
    }
}
  public String hello() throws RemoteException {
    System.out.println("Executando hello()");
    return "Hello!!!";
  }
 public void mensagem(String str) throws RemoteException {
    System.out.println("Client: "+ str);
  }
}
```

48

HelloClient.java

- Passo 3: Implementar o cliente de acordo com a interface do HelloWorld.

```
import java.rmi.registry.*;
public class HelloClient {
  public static void main(String[] args) {
    try {
      String host = args[0];
      Registry registry = LocateRegistry.getRegistry(host);
      HelloWorld stub= (HelloWorld) registry.lookup("Hello");
      String msg = stub.hello();
      System.out.println("Mensagem do Servidor: " + msg);
    } catch (Exception ex) {
       ex.printStackTrace();
} } }
```

49

Exemplo: Cliente/Servidor RMI

- Passo 2 (continuação): gerando o stub (código da comunicação).

  c:\teste\javac *.java, isso vai gerar HelloWorld.class, HelloServer.class, HelloClient.class

Rodar o rmiregistry onde se encontram HelloWorld.class e HelloServer.class

Em seguida, execute o servidor:

> java HelloServer

Finalmente, inicie o cliente com o comando:

> java HelloClient localhost (ou o ip remoto)

50

Outro Exemplo: 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;
}
```

51

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(); }
   }
}
```

52

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(); }
   }
}
```

53

Exemplo: Cliente/Servidor RMI

- Para esse exemplo específico, a stub deve ser gerada a partir da classe que implementa a interface HelloWorld;
- Execute rmic –v1.2 HelloServer para gerar a classe HelloServer_stub.class
- Em seguida, execute o servidor:

> java HelloServer

Finalmente, inicie o cliente com o comando:

> java HelloClient localhost

54

RMIRegitry

```
void rebind (String name, Remote obj)
```
//Este método é usado por um servidor registrar o identificador de um objeto remoto
//pelo nome

```
void bind (String name, Remote obj)
```
//Este método pode de maneira alternativa ser usado por um servidor registrar o
//identificador de um objeto remoto pelo nome.
//Se o nome já é ligado a uma referência remota uma exceção é levantada.

```
void unbind (String name, Remote obj)
```
//Este método remove um *binding*.

```
Remote lookup(String name)
```
//Este método é usado por um cliente para obter uma referência de objeto remoto.

```
String[] list()
```
//Este método retorna um *array* de *Strings* contendo nomes ligados no registro.

Interface do *RMIRegistry*

55

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

56

Exercício 1

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
public interface Calculator extends Remote {
    public long add(long a, long b)
        throws RemoteException;

    public long sub(long a, long b)
        throws RemoteException;

    public long mul(long a, long b)
        throws RemoteException;

    public long div(long a, long b)
        throws RemoteException;
}
```

57

Exercício 2

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
public interface Banco extends Remote {
    public String add(Conta correntista, short val)
        throws RemoteException;

    public String rem(Conta correntista, short val)
        throws RemoteException;

    public short saldo(Conta correntista)
        throws RemoteException;

    public Conta cadastrar(String nome, short CPF)
        throws RemoteException;
}
```

58