Sistemas Distribuídos
RPC –Remote Procedure Call

Universidade Federal do ABC

Turma:
Ciência da Computação

Prof. Dr. Francisco Isidro Massetto

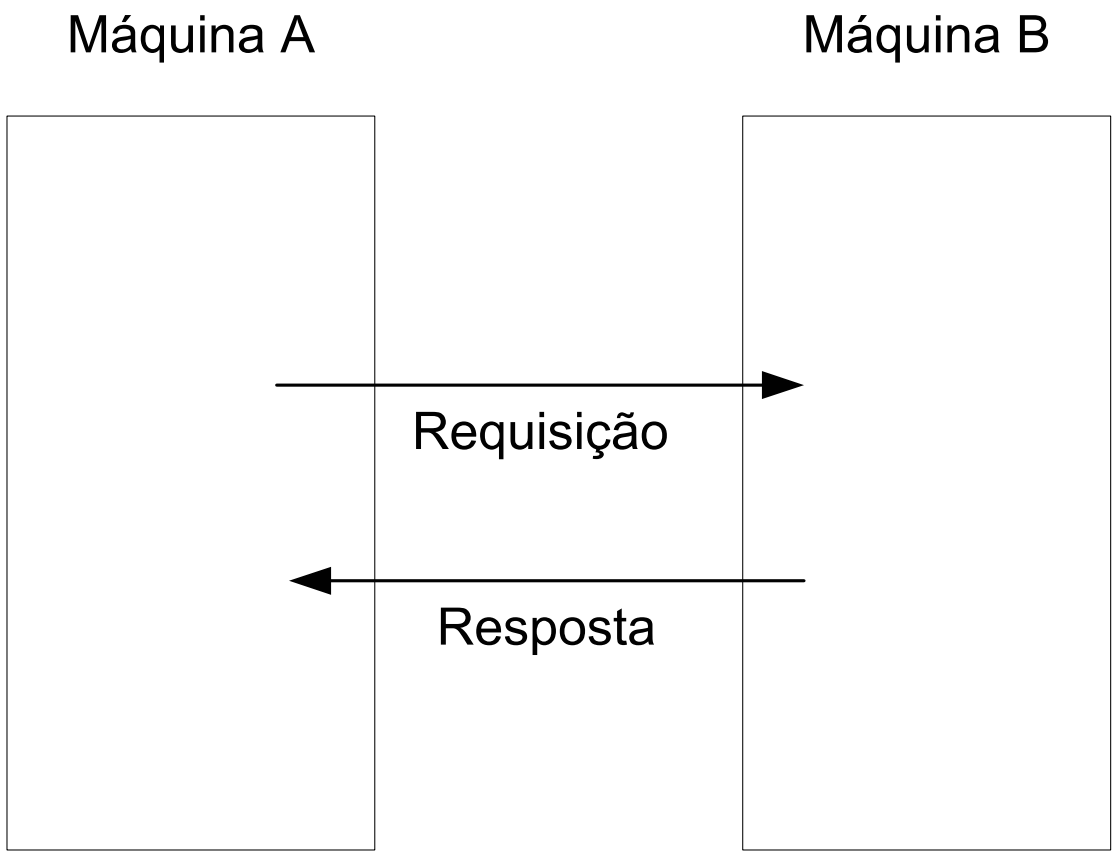
Cliente/Servidor

- Quais os problemas?
  - Baseado em E/S
  - Erro propagado em Sistemas Distribuídos
  - Difícil implementação
  - Não são amigáveis – foge da programação convencional (estruturado, OO, etc.)
  - Diferente da proposta de Sistemas Distribuídos
    - Parecer um Sistema Centralizado

Conceito

- Transparência
  - Chamada de procedimentos que estão em outras máquinas
  - Transferência de parâmetros para execução remota
  - Retorno das informações para o processo “chamador”
  - Conhecido como **chamada remota de procedimento** ou RPC

Conceito

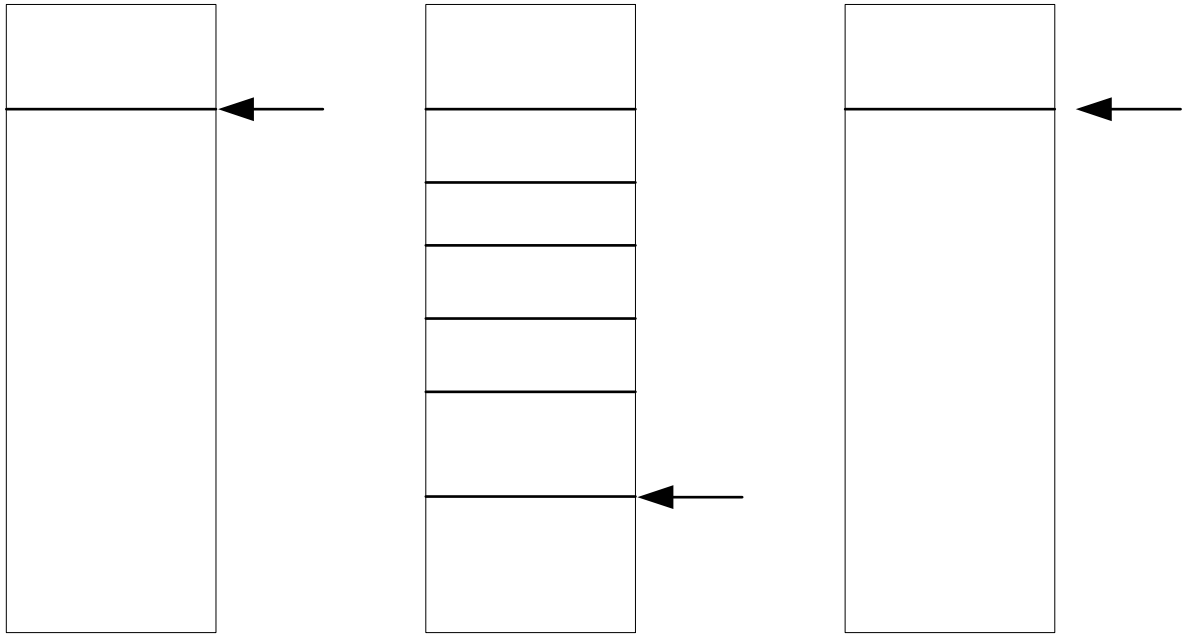


Problemas

- Espaços de endereçamento diferentes (máquinas diferentes)
- Arquitetura diferentes, interferindo nos tipos de dados
- Problemas no servidor
- Problemas no cliente

Características

- Parâmetros



Características

- Parâmetros
  - Pilha de Parâmetros
  - Passagem por valor
    - *fd*, *bytes*
  - Passagem por referência
    - *buf*

Características

- Funcionamento
  - Chamada ao *stub do cliente*
    - Versão no cliente responsável por iniciar a RPC
    - Esta versão recebe os parâmetros do procedimento e empacota para envio ao servidor, usando *send*
    - Após o envio, fica bloqueado (em *receive*) esperando a resposta do servidor

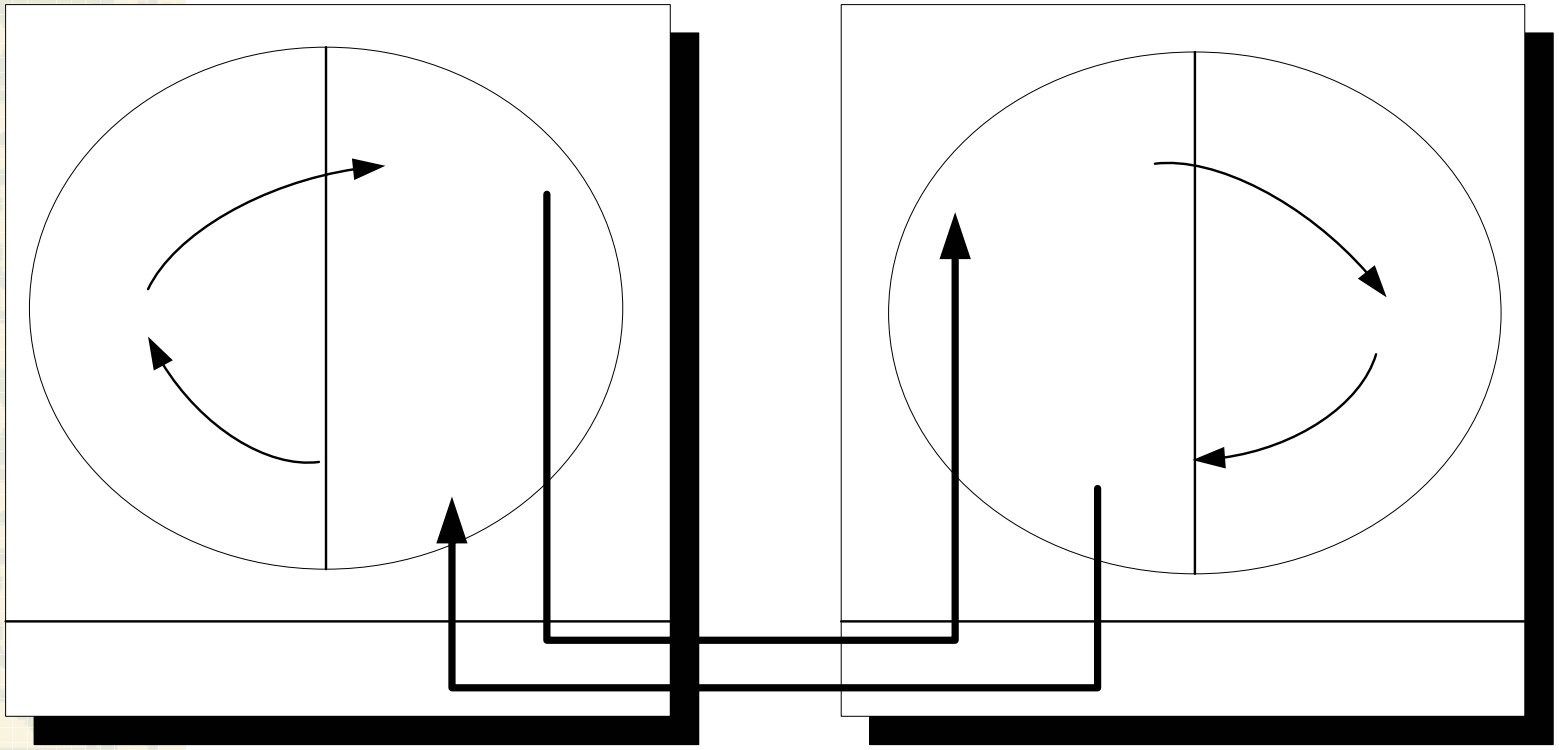
Características

- Funcionamento
  - No servidor, existe o *stub do servidor*
    - Serviço de SO responsável por receber um pacote de informações remotas
    - Verifica para qual serviço, e executa de acordo com os parâmetros recebidos
    - Envia de volta ao cliente, através de *send*

Características

- Funcionamento
  - Para o SO (kernel), o envio e recebimento de mensagens é transparente, ou seja, ele não sabe que é uma RPC
  - Portanto, deve existir um processo específico para o tratamento de RPCs

Características



Características

- 1. procedimento chama o *stub de cliente* (forma normal)
- 2. stub constrói a mensagem e faz um *send* (trap ao kernel)
- 3. o kernel envia a mensagem para a máquina remota
- 4. o kernel remoto entrega a mensagem ao *stub do servidor*
- 5. o stub desempacota a mensagem e chama o “servidor”
- 6. o servidor realiza a tarefa e retorna ao stub
- 7. o stub do servidor empacota o retorno e faz um *send* para o cliente
- 8. o kernel remoto envia a mensagem ao kernel do cliente
- 9. o kernel do cliente entrega a mensagem ao stub do cliente
- 10. o stub desempacota a mensagem e entrega ao cliente

Parâmetros

- Passagem por valor
  - Cópia do valor do parâmetro, mantendo o original
- Passagem por referência
  - Uso de endereço de memória
- Passagem por cópia/restaura
  - Copia o valor, e após sua modificação, sobrescreve o valor original

Parâmetros

- Problemas com Arquitetura
  - EBCDIC vs. ASCII
    - Mainframes e PCs
  - Little Endian vs. Big Endian
    - I386 e SPARC-Sun
  - Complemento de 2 vs. Complemento de 1

Parâmetros

0	3	0	2	0	1	5	0	Intel 386
L	7	L	6	I	4	J	5	

5	0	0	1	0	2	0	3	Sparc
J	4	I	5	L	6	L	7	

0	0	0	1	0	2	5	3	Mensagem Invertida
L	4	L	5	I	6	J	7	

Parâmetros

- Soluções
  - O conhecimento do procedimento, por ambos os lados – cliente/servidor – permite resolver o impasse
  - Definição de um padrão de comum acordo
    - Desempenho?
  - Utilização de 1 bit para identificação do formato do padrão

Parâmetros

- Ponteiros
  - Cópia do endereço?
  - Utilização de técnicas cópia/restaura
    - Envio do buffer da mensagem, a partir do tamanho
  - Aumento do desempenho
    - Diminuição de um envio de mensagem, baseado no tipo do procedimento
- Não existe variáveis “globais”

Binding Dinâmico

- Ligação dinâmica entre Cliente/Servidor
- Especificação formal de um Servidor
  - IN: valores de ENTRADA no servidor
  - OUT: valores de SAÍDA do servidor
  - IN OUT: entrada e saída

Binding Dinâmico

```
#include <header.h>
specification of file_server, version
  3.1:
   long read (in  char name[MAX_PATH], out
   char buf[BUF_SIZE], in long bytes, in
   long position);
   long write (in char name[MAX_PATH], in
   char buf[BUF_SIZE],in long bytes, in
   long position);
   int create (in char name[MAX_PATH], in
   int mode);
   int delete (in char name[MAX_PATH]);
end;
```

Binding Dinâmico

- Essa especificação é utilizada para o processo de linkedição do compilador
  - O arquivo-objeto será linkado ao programa cliente ou servidor
  - Stub do cliente e do servidor
- É necessário construir o código referente a esses procedimentos

Binding Dinâmico

- Servidor
  - Ao ser executado, avisa ao programa de registro (*binder*) que está ativo: registro do serviço
- Cliente
  - Quando um procedimento é chamado, este se liga ao servidor, utilizando o *binder* como intermediário

Binding Dinâmico

- Vantagens
  - Diversos servidores com a mesma “interface”
  - Servidores que falham são automaticamente “desregistrados”
  - Autenticação de usuários/clientes
- Desvantagens
  - Overhead de consulta ao binder
  - Diversos binders necessita de atualização entre eles

Falhas

- A concepção do RPC é deixar a programação transparente, mas
  - Cliente não acha o servidor
  - A mensagem do cliente para o servidor foi perdida
  - A mensagem do servidor para o cliente foi perdida
  - O servidor sai do ar após receber uma solicitação
  - O cliente sai do ar após ter enviado uma solicitação

Falhas

- Cliente não acha o servidor
  - O servidor está fora do ar
  - Versões diferentes de stubs
  - Retornos de variáveis “inválidas”: e.g. -1
  - Criação de exceções
  - Perda da transparência

Falhas

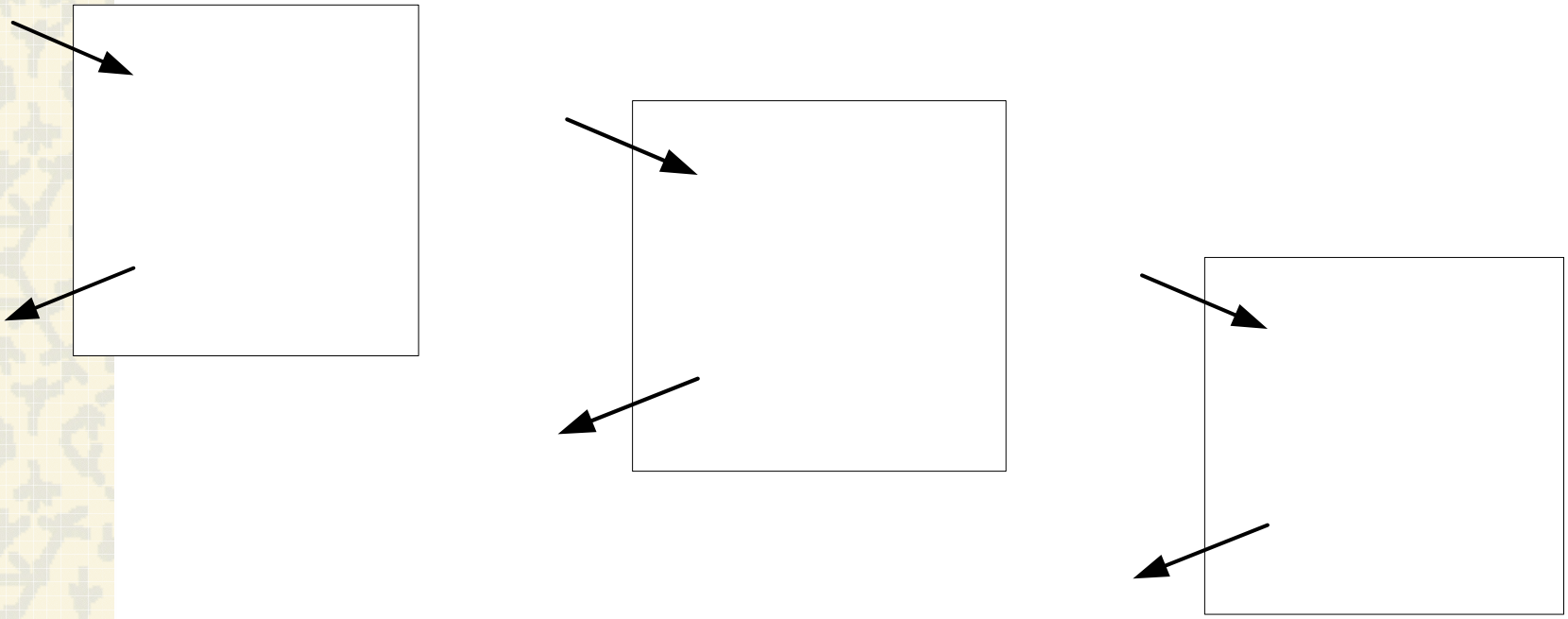
- A mensagem do cliente para o servidor foi perdida
  - Limite de tempo de espera (timeout)
  - Reenvio em kernel
  - Retorno de erro, após diversas tentativas

Falhas

- A mensagem do servidor para o cliente foi perdida
  - Reenvio da solicitação (?)
    - Idempotência de execução
  - O kernel do cliente mantém uma seqüência de números de envio
  - A cada mensagem, o servidor verifica se o número deste kernel já não foi executado
  - Bit de sinalização: original ou retransmissão

Falhas

- O servidor sai do ar após receber uma solicitação



Falhas

- O servidor sai do ar após receber uma solicitação
  - Espera e reenvia / ache novo servidor
  - Desiste e comunique falha
  - Nada é garantido (implementação)
- As falhas até o momento não são distinguíveis para um Cliente

Falhas

- O cliente sai do ar após ter enviado uma solicitação
  - Processamento órfão
  - Gasto de tempo do servidor
  - Soluções
    - Extermínio
    - Reencarnação
    - Reencarnação branda
    - Expiração (quantum *T*)

Falhas

- Extermínio
  - Eliminar todos os órfãos
  - Falha na rede??
  - Problema: encadeamento de falhas (servidor pode ter sido cliente em uma RPC)
- Reencarnação
  - Dividir o tempo em “épocas”
  - Nova chamada → nova época
  - Falha na rede → impossível encontrar o órfão → facilmente detectado depois (necessita nova época)

Falhas

- Reencarnação Branda
  - Verificar se há processamento de épocas anteriores antes de eliminar órfão
  - Servidor tenta encontrar cliente
    - Se cliente não existir, elimina o órfão
- Expiração
  - Tempo máximo para servidor executar serviço
  - Problema: mensurar o *quantum*

CORBA

- Common Object Request Broker Architecture
  - Padrão aberto, definido pela OMG
  - Aplicável a Orientação a Objetos
  - Independente de linguagem (cliente/servidor)
  - Chamada remota através de ORBs (Object Request Broker) locais: de stub-cliente para skeleton-servidor
  - Definição da interface de chamada através da IDL – Interface Definition Language

DCOM

- Distributed Component Object Model
  - Interação distribuída de componentes
  - Definido pela Microsoft para OO e Componentes
  - As interfaces do objetos podem ser definidas em várias linguagens de programação e em diversas plataformas
  - Stub do cliente (proxy) se comunica com o stub do servidor, que requisita o objeto DCOM

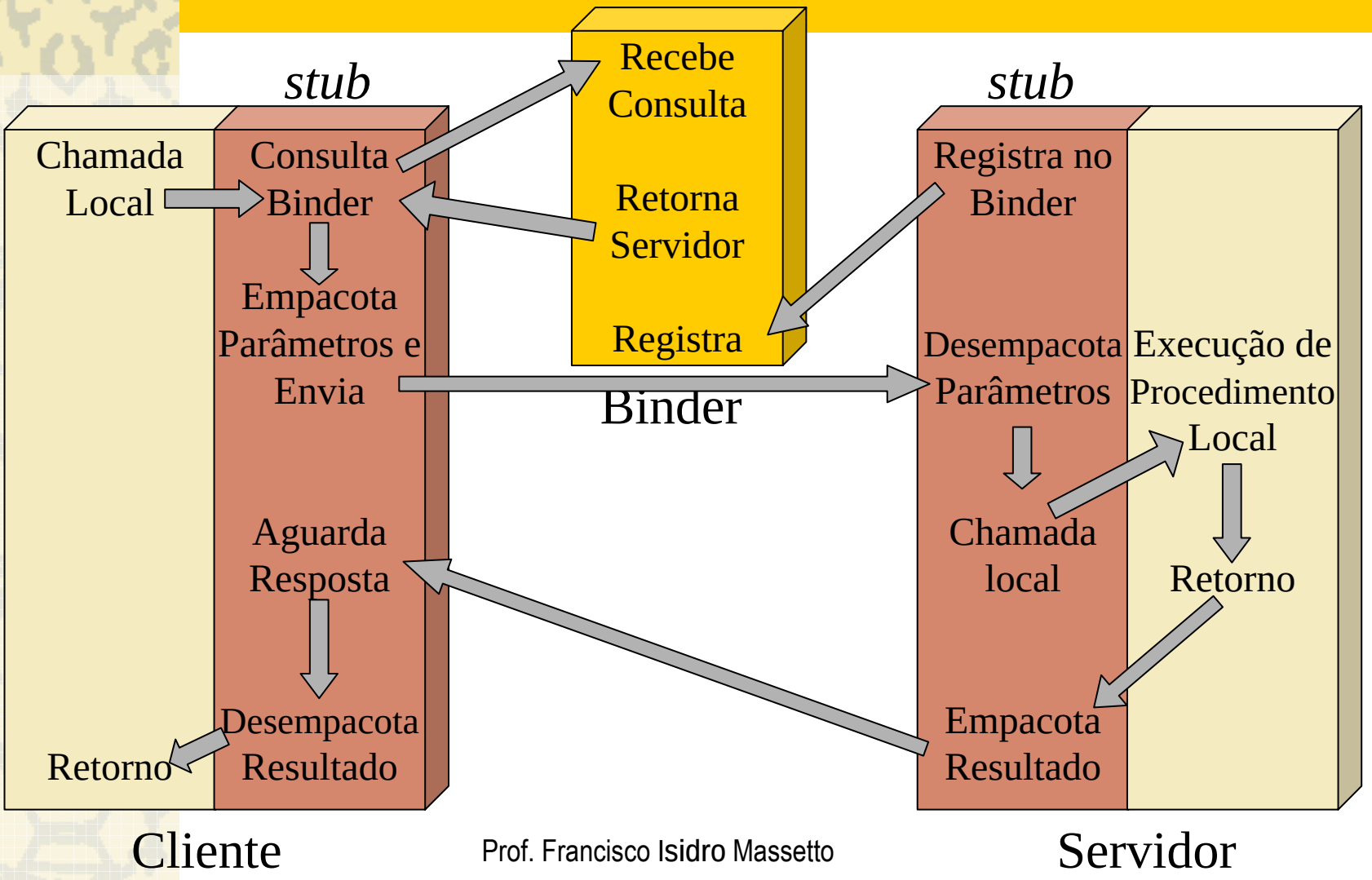
RMI

- Remote Method Invocation
  - RPC da linguagem Java
  - Orientação a Objetos
  - 3 camadas
    - Stub/Skeleton: é invocado primeiro, recebendo objetos e serializando-os (bytes)
    - RRL – Remote Reference Layer: cada lado possui a sua, responsável por montar a mensagem de envio/resposta entre cliente/servidor
    - Transporte: envio das informações via rede

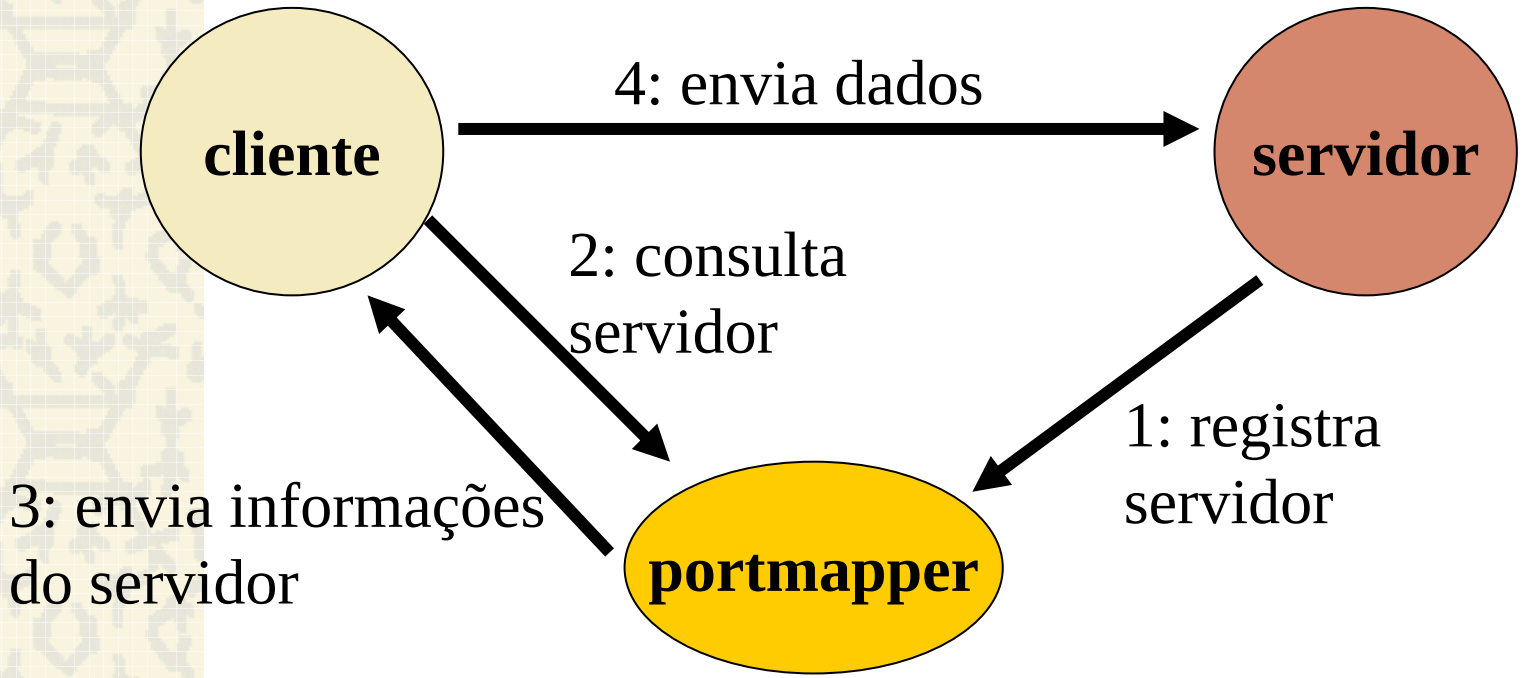
Tutorial Sun-RPC

- Utilização da ferramenta *rpcgen* (RPC Generator)
  - Compilador para a linguagem RPCL (RPC Language)
- Ganho de produtividade: geração a partir da definição
  - Stub do cliente e do servidor
  - Templates para programas do cliente e servidor

Tutorial Sun-RPC



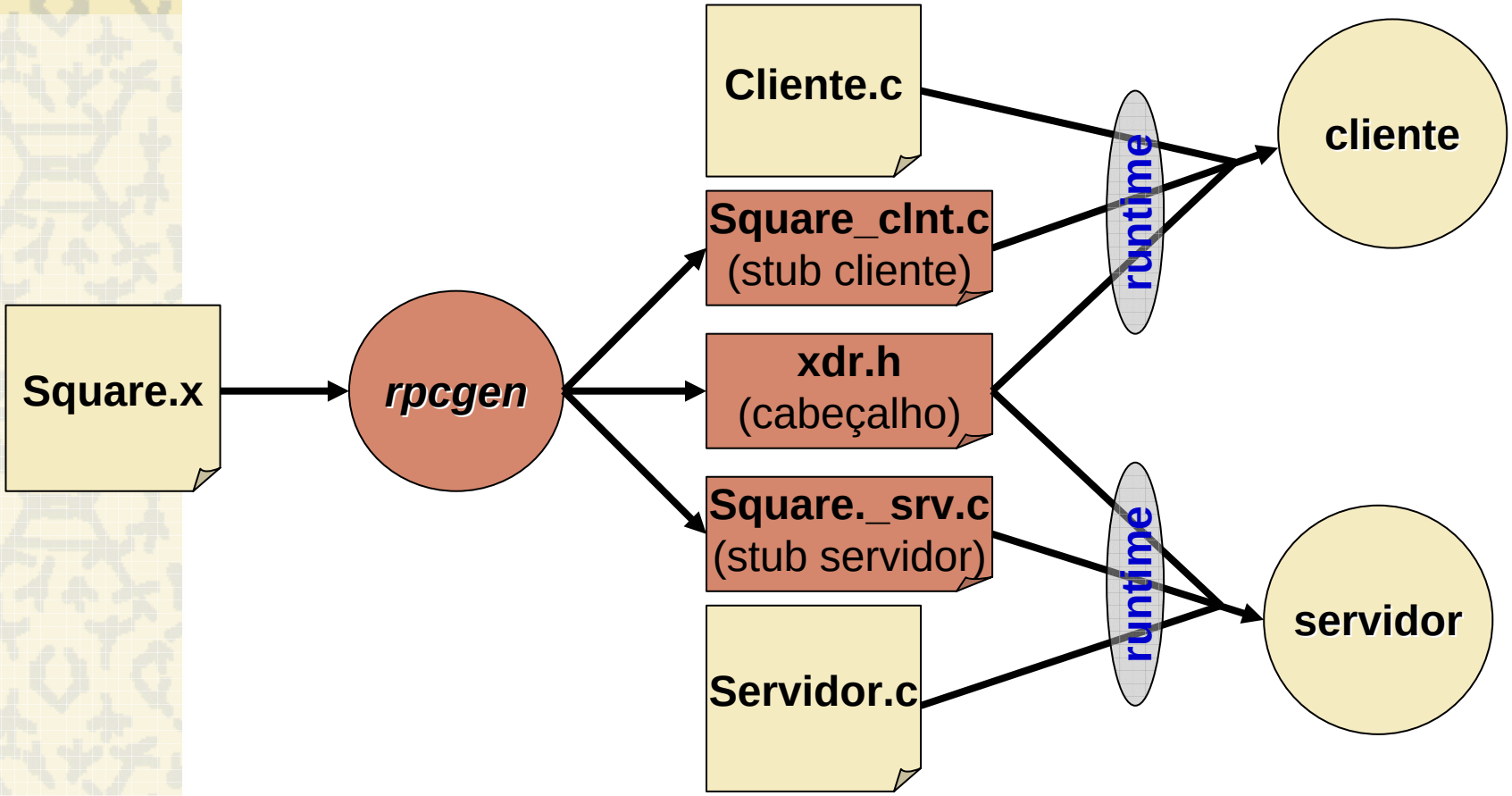
Tutorial Sun-RPC



Tutorial Sun-RPC

- Exemplo: square.x
  - Cria um procedimento que calcula o quadrado de um número
  - Cria um servidor e registra o servidor usando RPC
  - A definição está em *square.x*
  - A partir da definição, gera-se os stubs
  - O servidor de registra no **portmapper**
  - O cliente consulta o **portmapper** e chama o servidor

Tutorial Sun-RPC



Tutorial Sun-RPC

- square.x

```
struct square_in {  /* parâmetro de entrada*/
  long arg;
};
struct square_out {  /* parâmetro de saída */
  long res;
};
program SQUARE_PROG {
    version SQUARE_VERS {
        /* procedimento SQUAREPROC que tem
  square_in como entrada e square_out como
  saída*/
        square_out SQUAREPROC(square_in) = 1;
    } = 1;  /* versao numero 1 */
} = 0x31230000; /* id do programa*/
```

Tutorial Sun-RPC

- Compilando definição...

  $ rpcgen -C square.x

  $

- Resultado da geração
  - square.h → protótipos
  - xdr.c → arquivo de conversão de dados
  - square_clnt.c → stub do cliente
  - square_svc.c → stub do servidor

Tutorial Sun-RPC

- Compilando biblioteca comum

```
$ gcc -c square_xdr.c -o square_xdr.o
$
```

- Resultado da geração
  - square_xdr.o → biblioteca de conversão de tipos

Tutorial Sun-RPC

- servidor.c

```
#include "square.h" /* gerado pelo RPCgen */

/* servidor sempre possui a seguinte forma:
 NOME_VERSAO_SVC e um dos parâmetros é do tipo
 svc_req. Além disso, retorna sempre um
  ponteiro*/
square_out * squareproc_1_svc (square_in *in,
  struct svc_req *rqstp) {

        static square_out out;
        out.res = in->arg * in->arg;
        return(&out);
}
```

Tutorial Sun-RPC

- Compilando o Servidor...
  - Compilando apenas o servidor

$ gcc -c servidor.c -o servidor.o

  - Compilando apenas o stub do servidor

$ gcc -c square_svc.c -o square_svc.o

  - Compilando e linkando a aplicação do servidor

$ gcc -o servidor servidor.o square_xdr.o square_svc.o

Tutorial Sun-RPC

- cliente.c

```
#include "square.h" /*  cabeçalho gerado pelo RPCgen  */
int main(int argc, char **argv) {
        CLIENT  *cl; /* definido em rpc.h */
        square_in in;
        square_out  *outp;

        if (argc != 3) {
          printf("uso: cliente <host> <valor>\n");
          return 1;
        }

        cl = clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");
        /* crio um novo cliente passando o host, qual é o programa,
           sua versão e qual o protocolo de transporte */

        /* pego o argumento da linha de comando */
        in.arg = atol(argv[2]);

        /* chamo o procedimento, seguido de sua versão */
        if ((outp = squareproc_1(&in, cl)) == NULL) {
          perror(clnt_sperror(cl, argv[1])); exit(1); }
        printf("Resultado: %d\n", outp->res);
        return 0;
}
```

Tutorial Sun-RPC

- Compilando o Cliente...
  - Compilando apenas o cliente

$ gcc -c cliente.c -o cliente.o

  - Compilando apenas o stub do cliente

$ gcc -c square_clnt.c -o square_clnt.o

  - Compilando e linkando a aplicação do cliente

$gcc -o cliente cliente.o square_xdr.o square_clnt.o

Tutorial Sun-RPC

- Executando o Servidor

  $ ./servidor &

  [1] 3567

  $

- Executando o Cliente

  $ ./cliente

  uso: cliente <host> <valor>

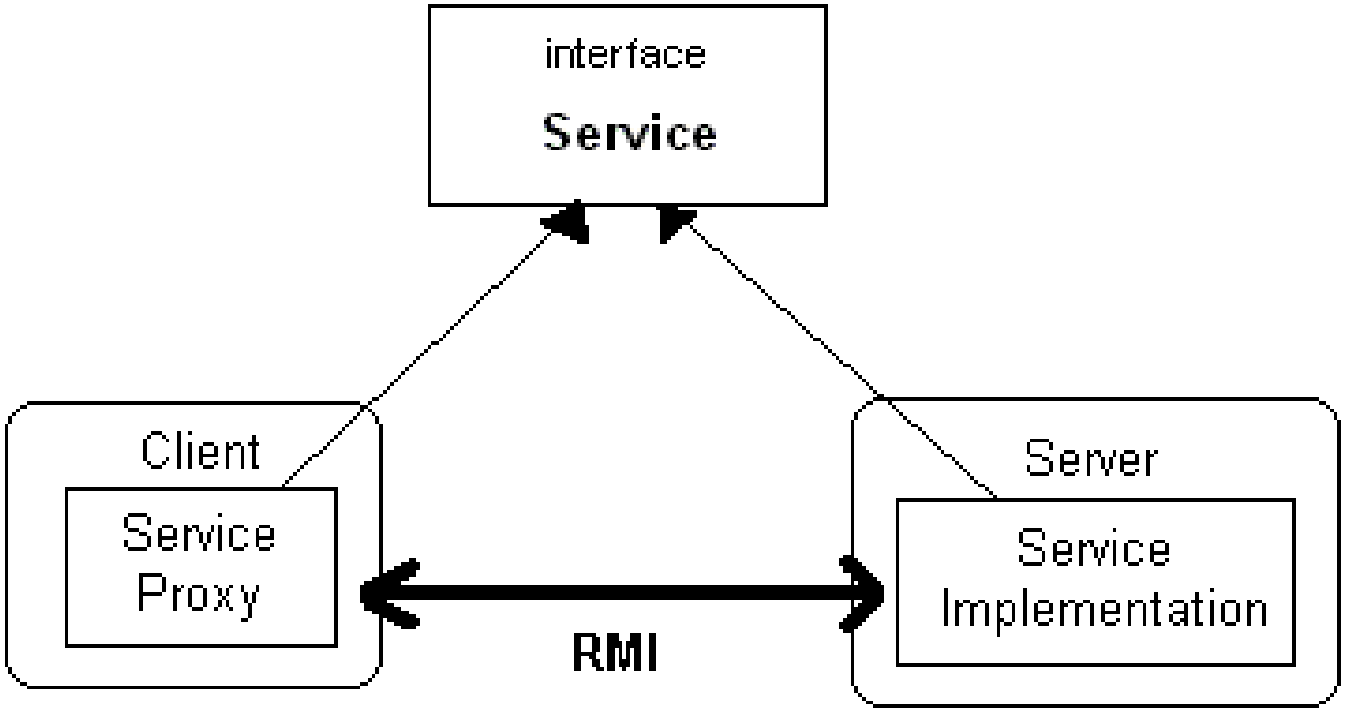
  $ ./cliente localhost 11

  Resultado: 121

Tutorial Java-RMI

- Utilização de Objetos Java de forma "transparente"
- Uso de classe tipificada para a definição dos métodos: *interface*
- A classe do cliente atua como *proxy*
- A implementação da classe fica no servidor

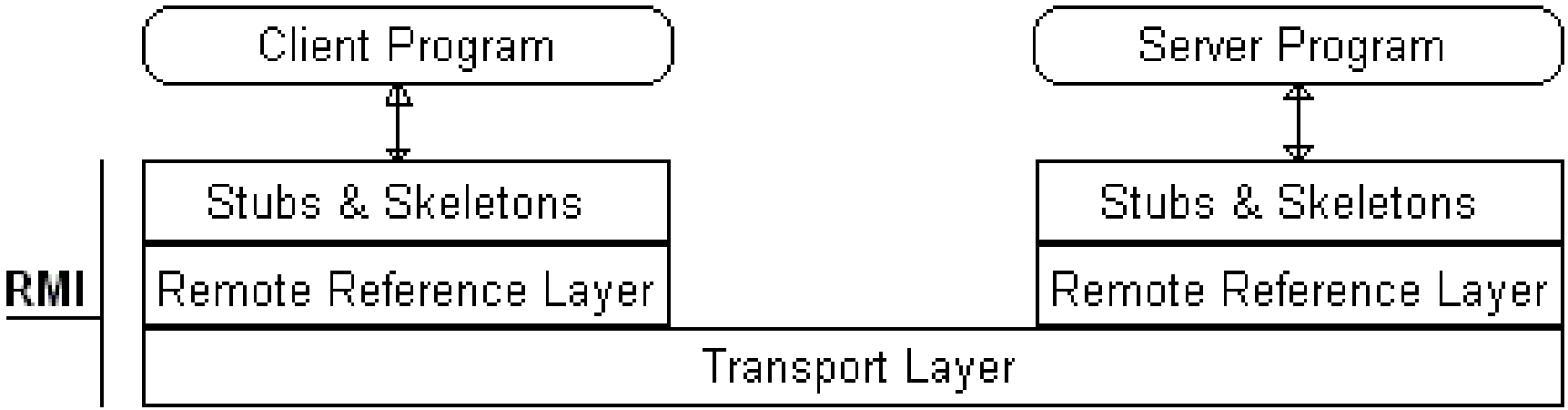
Tutorial Java-RMI



Tutorial Java-RMI

- Camadas
  - Stub/Skeleton: intercepta as chamadas de métodos feitas pelo cliente
  - RRL – Remote Reference Layer: interpreta e gerencia referências feitas dos clientes para os objetos do serviço remoto. A conexão do cliente ao servidor é Unicast (uma-para-um).
  - Transporte: conexões TCP/IP entre as maquinas

Tutorial Java-RMI



Tutorial Java-RMI

- Interface do Servidor

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Servidor
                    extends Remote {
  public String montaMensagem(String o)
                    throws RemoteException;
}
```

Tutorial Java-RMI

- Implementação do serviço do Servidor definido na Interface

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ServidorImpl extends UnicastRemoteObject
                          implements Servidor {
   public ServidorImpl() throws RemoteException {
        super();
   }
   public String montaMensagem(String o)
                    throws RemoteException {
        return "Montagem: "+o.toString();
   }
}
```

Tutorial Java-RMI

- Aplicação Servidor

```
import java.rmi.Naming;

public class ServidorPrincipal {
   public ServidorPrincipal() {
        try {
            Servidor m = new ServidorImpl();
            Naming.rebind("rmi://localhost:1099/Servidor", m);
        } catch( Exception e ) {
            System.out.println( "Trouble: " + e );
        }
    }
    public static void main(String[] args) {
        new ServidorPrincipal();
    }
}
```

Tutorial Java-RMI

- Compilando Servidor....
  - Compilando apenas o serviço do Servidor

  $ javac ServidorImpl.java
  - Criando Stubs do Servidor *ServidorImpl_Stub.class*

  $ rmic ServidorImpl
  - Compilando a aplicação do Servidor

  $ javac ServidorPrincipal.java

Tutorial Java-RMI

- Servidor
  - Registro do servidor através da URL de RMI:
    `rmi://<host>[:port]/<service>`
  - Utilização de herança:
    `Servidor m = new ServidorImpl();`
  - Registro do servido através do método
    `Naming.rebind()`

Tutorial Java-RMI

- Cliente

```
import java.rmi.*;

public class Cliente {
  public static void main(String args[]) {
    try {
      Servidor m = (Servidor)
   Naming.lookup("rmi://localhost/Servidor");
      System.out.println(m.montaMensagem("Ola!"));
    } catch (Exception e) {
      System.out.println("GenericException: " + e.toString());
    }
  }
}
```

Tutorial Java-RMI

- Compilando Cliente...
  - Compilando apenas a aplicação Cliente

  $ javac Cliente.java

Tutorial Java-RMI

- Cliente
  - Procura do servidor através da URL de RMI:
    `rmi://<host>[:port]/<service>`
  - Utilização do método de procura:
    `Naming.lookup()`
  - Chamada convencional de método:
    `m.montaMensagem()`
  - Possibilidades de diversos erros (não codificado)

Tutorial Java-RMI

- Execução do Binder
  - O diretório deve conter apenas:
    - Interface de definição
    - Stub do servidor gerado a partir do *rmic*

  $ rmiregistry

Tutorial Java-RMI

- Execução do Servidor
  - O diretório deve conter apenas:
    - Interface de definição
    - Classe de implementação do serviço
    - Classe de execução da aplicação do servidor

  $ java Servidor

Tutorial Java-RMI

- Execução do Cliente
  - O diretório deve conter apenas:
    - Interface de definição
    - Classe de implementação do cliente

```
$ java Cliente
Montagem: Ola!
$
```