

SISTEMAS DISTRIBUÍDOS I

Capítulo 4

Invocação de procedimentos e de métodos remotos

NOTA PRÉVIA

A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 5th Edition, 2009

Para saber mais:

RMI/RPCs - capítulo 5.

Representação de dados e protocolos - capítulo 4.3.

Web services – capítulo 9

Corba – capítulo 8.3.

PROBLEMA / MOTIVAÇÃO

É possível estruturar uma aplicação distribuída usando como base as interacções através da troca de mensagens entre processos. Os problemas são:

- é complicado e cheio de detalhes não relevantes
- os programas ficam estruturados em função dos protocolos (trocas de mensagens)
- os servidores ficam estruturados em função da lista de mensagens que sabem processar

Muitas linhas de código são repetitivas, não contêm nenhum significado aplicacional específico e referem-se ao processamento das comunicações:

- criação de communication end points e sua associação aos processos
- criação, preenchimento e interpretação das mensagens
- selecção do código a executar consoante o tipo da mensagem recebida
- gestão de temporizadores/tratamento das falhas

Não se poderão automatizar as interacções cliente/servidor?

INVOCAÇÃO REMOTA DE PROCEDIMENTOS

Num programa definido numa linguagem imperativa, definem-se funções/procedimentos para executar uma dada operação

Uma extensão natural num ambiente distribuído consiste em permitir a execução de procedimentos noutra máquina

Ex. ONC/RPC, DCE

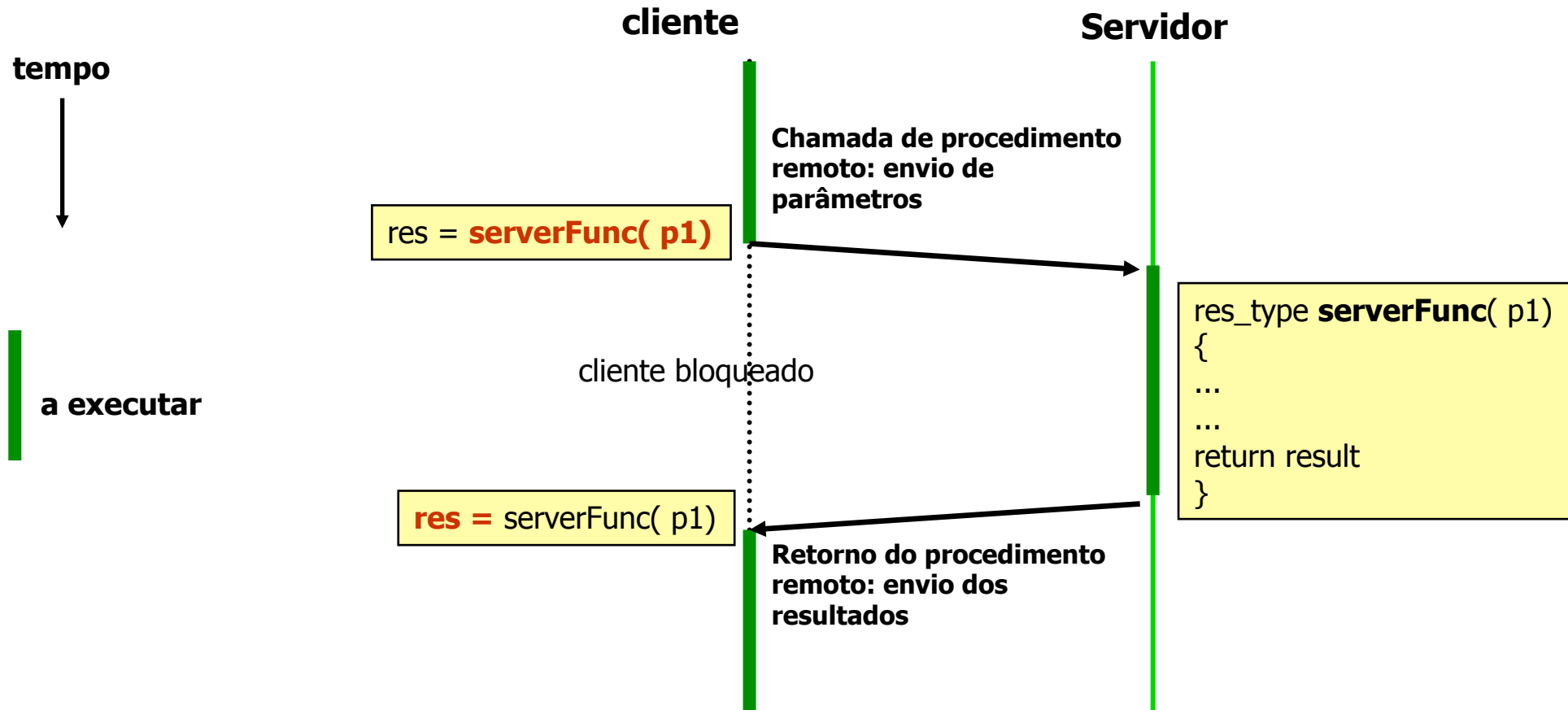
Se os procedimentos estiverem definidos no âmbito de um objecto, chama-se invocação remota de métodos

Ex.: Java RMI, CORBA e .NET

Web services: permitem efectuar invocação remota de procedimentos usando mensagens (geralmente) representadas em XML e enviadas usando HTTP

Web services permitem interacções mais complexas que simples pedido/resposta

INVOCAÇÃO DE PROCEDIMENTOS REMOTOS (RPCs)



Modelo

Servidor exporta interface com operações que sabe executar

Cliente invoca operações que são executadas remotamente e (normalmente) aguarda pelo resultado

INVOCAÇÃO DE PROCEDIMENTOS REMOTOS (CONT.)

Propriedades

- Extensão natural do paradigma imperativo/procedimental a um ambiente distribuído

 - Chamada síncrona de funções – modelo de comunicação ?

- Esconde detalhes de comunicação (e tarefas repetitivas)

 - Construção, envio, recepção e tratamento das mensagens

 - Tratamento básico de erros (devem ser tratados ao nível da aplicação)

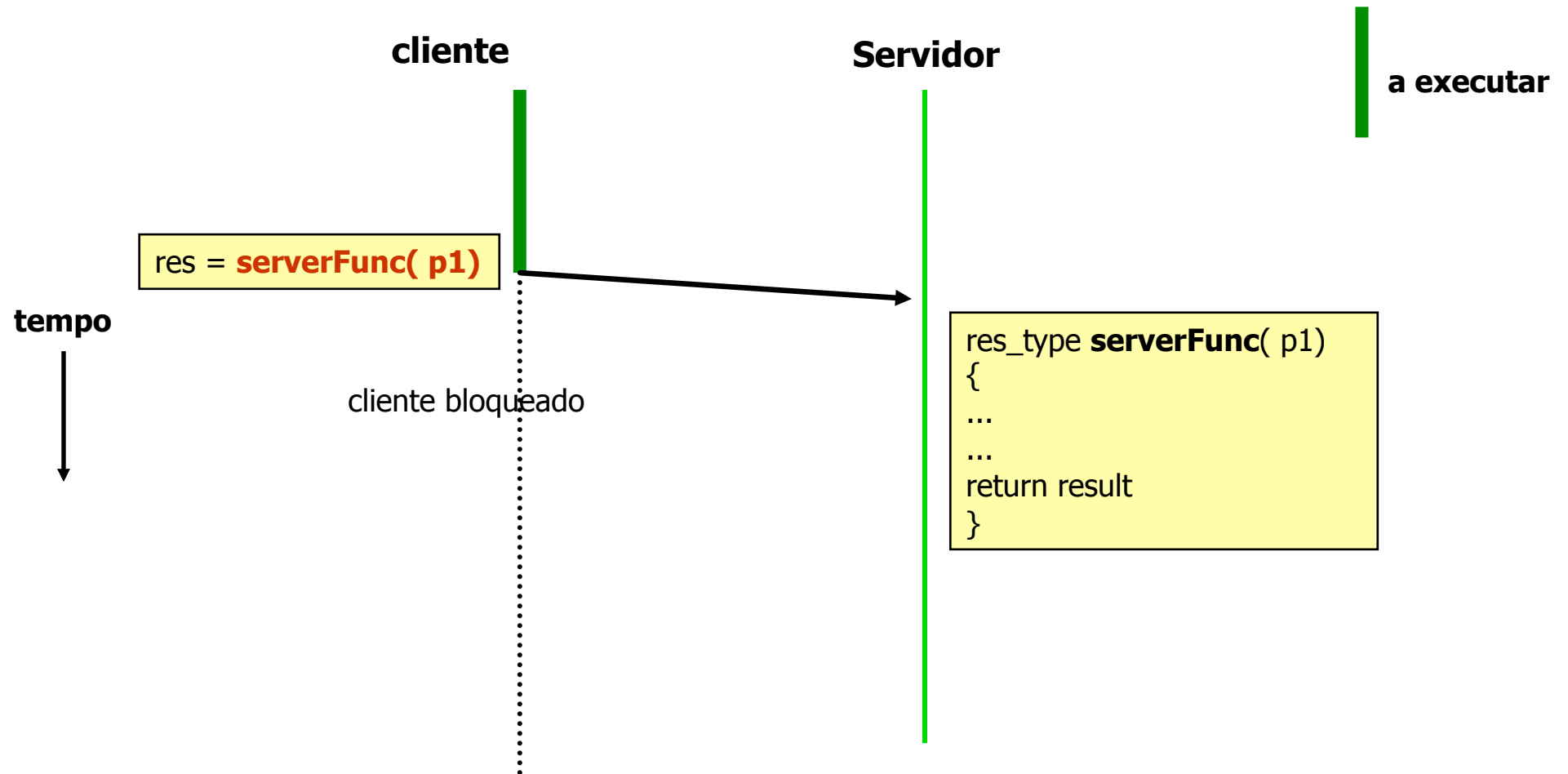
 - Heterogeneidade da representação dos dados

- Simplifica disponibilização de serviços

 - Interface bem definida, facilmente documentável e independente dos protocolos de transporte

 - Sistema de registo e procura de serviços

RPCs: COMO IMPLEMENTAR



COMO ESCONDER OS DETALHES?

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

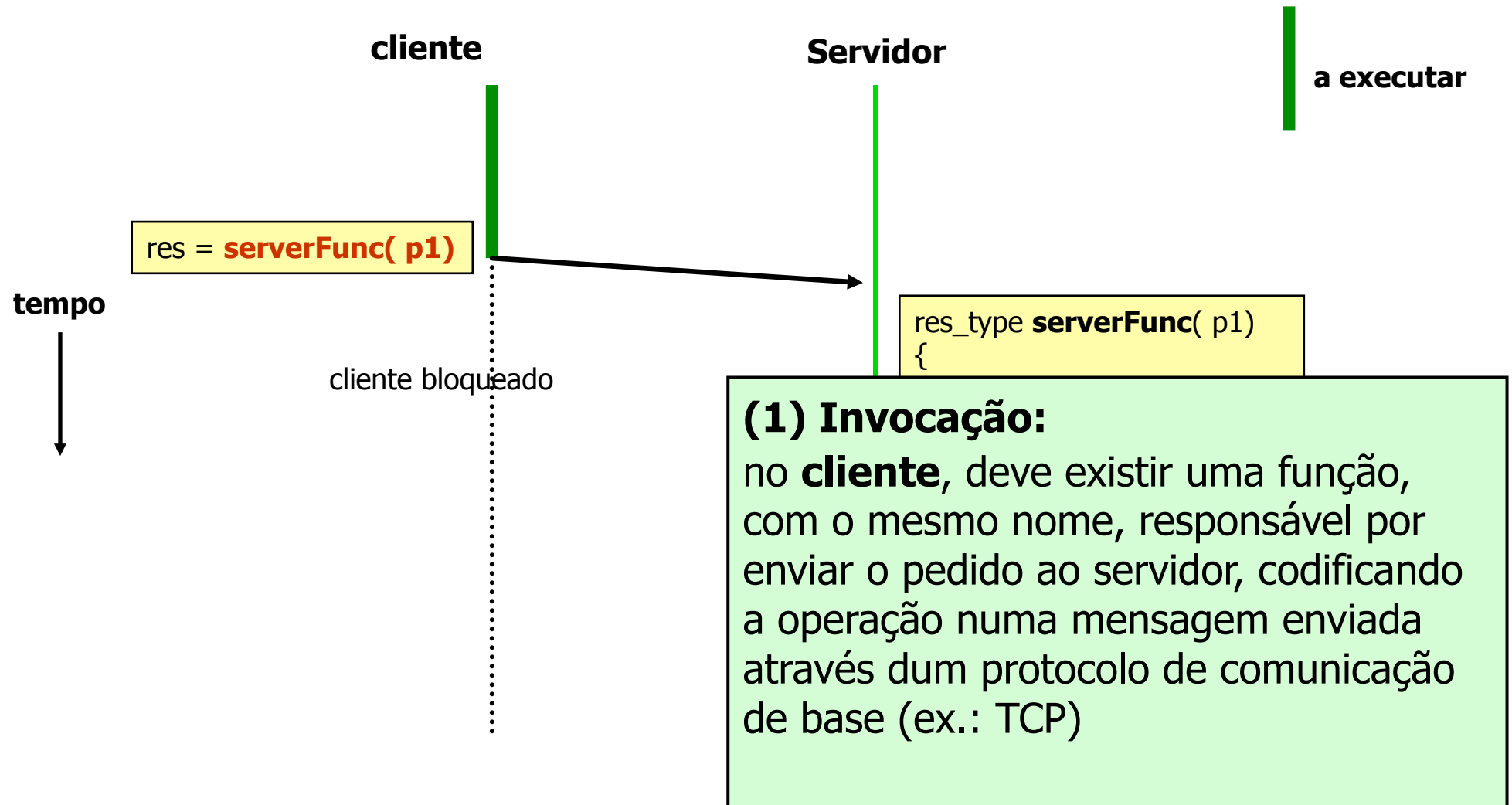
Cliente

```
res = serverFunc( p1)
```

Servidor

```
res_type serverFunc( T1 p1) {  
    ...  
    return result  
}
```

RPCs: COMO IMPLEMENTAR



COMO ESCONDER OS DETALHES

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

Cliente

```
res = serverFunc( p1)
```

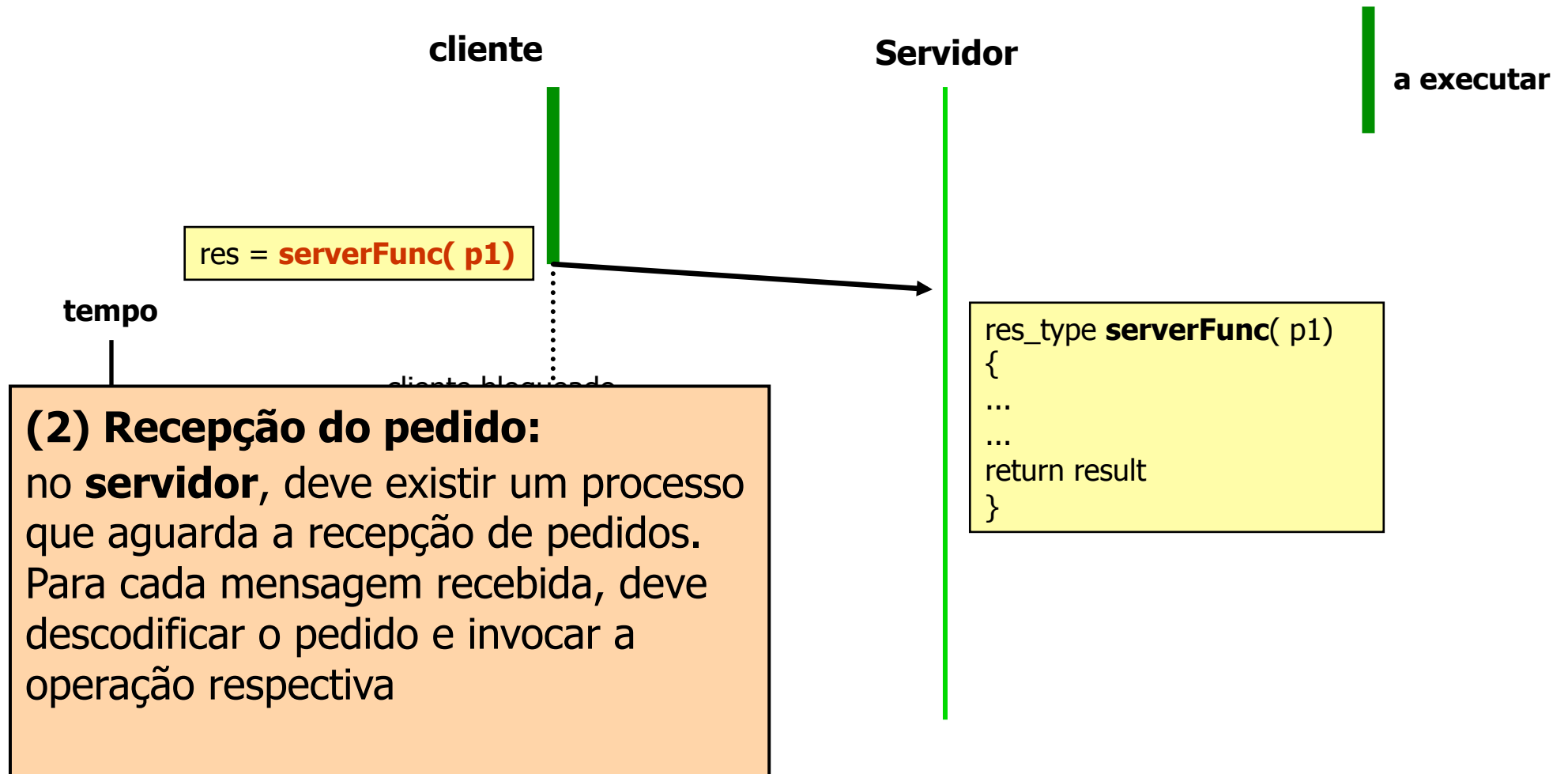
```
res_type serverFunc( T1 p1)  
  s = new Socket( host, port)  
  s.send( msg( "serverFunc",[p1]))
```

(1) Invocação:

no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

```
}
```

RPCs: COMO IMPLEMENTAR



COMO ESCONDER OS DETALHES?

(2) Recepção do pedido:

no **servidor**, deve existir um processo que aguarda a recepção de pedidos. Para cada mensagem recebida, deve decodificar o pedido e invocar a operação respectiva

```
res_type serverFunc( T1 p1)
  s = new Socket( host, port)
  s.send( msg( "serverFunc",[p1]))
```

(GUAGEM)

Servidor

```
res_type serverFunc( T1 p1) {
    ...
    return result
}
```

```
s = new ServerSocket
forever
  Socket c = s.accept();
  c.receive( msg( op, params))
  if( op = "serverFunc")
    res = serverFunc( params[0]);
  else if( op = ...)
    ...
```

RPCs: COMO IMPLEMENTAR

(3) Envio da resposta:

no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

tempo



cliente bloqueado

Servidor

a executar

```
res_type serverFunc( p1)
{
...
...
return result
}
```

COMO ESCONDER OS DETALHES?

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

(3) Envio da resposta:

no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

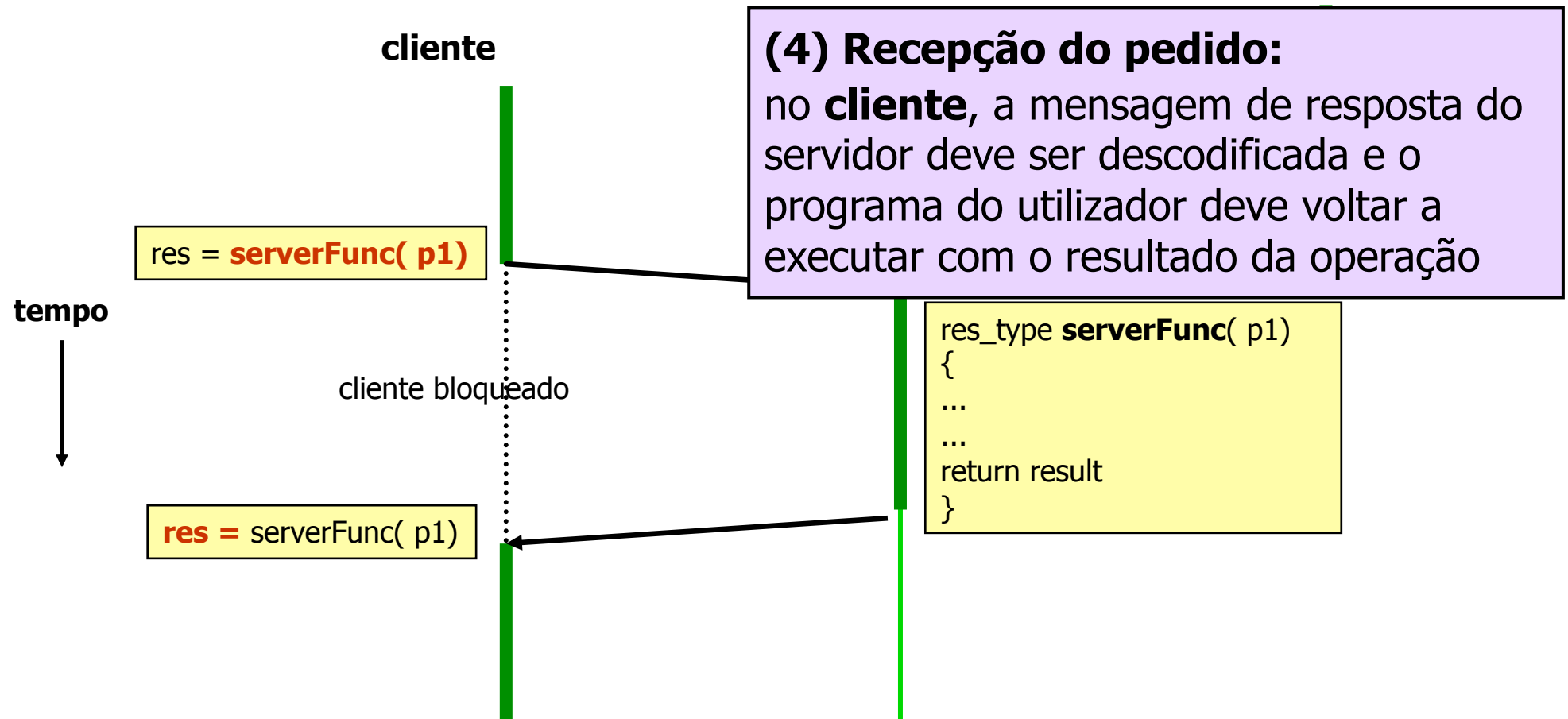
```
res_type serverFunc( T1 p1)
  s = new Socket( host, port)
  s.send( msg( "serverFunc",[p1]))
```

Servidor

```
res_type serverFunc( T1 p1) {
    ...
    return result
}
```

```
s = new ServerSocket
forever
  Socket c = s.accept();
  c.receive( msg( op, params))
  if( op = "serverFunc")
    res = serverFunc( params[0]);
  else if( op = ...)
    ...
  c.send( msg(res))
  c.close
```

RPCs: COMO IMPLEMENTAR



COMO ESCONDER OS DETALHES?

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

Cliente

```
res = serverFunc( p1)
```

```
res_type serverFunc( T1 p1)  
  s = new Socket( host, port)  
  s.send( msg( "serverFunc",[p1]))  
  s.receive( msg( result))  
  s.close  
  return result
```

(4) Recepção do pedido:

no **cliente**, a mensagem de resposta do servidor deve ser decodificada e o programa do utilizador deve voltar a executar com o resultado da operação

```
s = new ServerSocket  
forever  
  Socket c = s.accept();  
  c.receive( msg( op, params))  
  if( op = "serverFunc")  
    res = serverFunc( params[0]);  
  else if( op = ...)  
    ...  
  c.send( msg(res))  
  c.close
```

COMO ESCONDER OS DETALHES?

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

Cliente

```
res = serverFunc( p1)
```

```
res_type serverFunc( T1 p1)  
  s = new Socket( host, port)  
  s.send( msg( "serverFunc",[p1]))  
  s.receive( msg( result))  
  s.close  
  return result
```

Na prática, sucessivas
invocações podem partilhar
o mesmo socket...

Stub do cliente ou *proxy* do servidor

(4) Recepção do pedido:

no **cliente**, a mensagem de resposta do servidor deve ser decodificada e o programa do utilizador deve voltar a executar com o resultado da operação

(1) Invocação:

no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

COMO ESCONDER OS DETALHES?

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

Stub ou skeleton do servidor

(3) Envio da resposta:

no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

(2) Recepção do pedido:

no **servidor**, deve existir um processo que aguarda a recepção de pedidos. Para cada mensagem recebida, deve decodificar o pedido e invocar a operação respectiva

Servidor

```
res_type serverFunc( T1 p1) {  
    ...  
    return result  
}
```

```
s = new ServerSocket  
forever  
    Socket c = s.accept();  
    c.receive( msg( op, params))  
    if( op = "serverFunc")  
        res = serverFunc( params[0]);  
    else if( op = ...)  
        ...  
    c.send( msg(res))  
    c.close
```

AUTOMATIZAÇÃO DO PROCESSO - *STUB* OU *PROXY* *COMPILERS*

Nos sistemas de RPC/RMI, o código da comunicação é transparente para a aplicação

Stub do cliente inclui funções do cliente efectuem a comunicação com o servidor para executar o método no servidor

Stub do servidor inclui código de comunicação para esperar invocações e executá-las, devolvendo o resultado

No Java RMI, o stub do cliente é gerado automaticamente

O .NET remoting inclui suporte nativo para invocação remota, pelo que as comunicações são tratadas pelo *runtime* de suporte

AGENDA

Invocação remota de procedimentos/objectos

Motivação

Modelo

Definição de interfaces e método de passagem de parâmetros

Codificação dos dados

Mecanismos de ligação (binding)

Protocolos de comunicação

Concorrência no servidor

Sistemas de objetos distribuídos

INTERFACE DEFINITION LANGUAGES (IDL)

IDLs são linguagens que permitem definir interfaces de servidores/objectos remotos, especificando:

- Tipos e constantes

- Interface do serviço - assinatura das funções/procedimentos

Os IDLs são usados apenas para definir as interfaces, não o código das operações

- Por vezes, esta distinção é difícil de fazer porque os IDLs estão integrados com linguagem

- Em certos sistemas (e.g. .NET remoting), a interface pode não ser definida autonomamente

IDLs – APROXIMAÇÕES POSSÍVEIS

Usar sub-conjunto de uma linguagem já existente

Ex.: Java RMI

Definir linguagem específica para especificar interfaces dos servidores/objectos remotos

Ex.: WSDL

Geralmente baseado numa linguagem existente

Necessidade de mapear o IDL e as linguagens de desenvolvimento dos clientes/servidores

INTERFACE REMOTA EM JAVA RMI

```
public interface ContaBancaria
```

```
    extends Remote
```

```
{
```

```
    public void depositar ( float quantia )
```

```
        throws RemoteException;
```

```
    public void levantar ( float quantia)
```

```
        throws SaldoDescoberto, RemoteException;
```

```
    public float saldoActual ( )
```

```
        throws RemoteException;
```

```
}
```

Interfaces remotos

estendem **Remote**

Interfaces definidos em Java
standard

Métodos devem lançar
RemoteException para tratar
erros de comunicação

INTERFACE DEFINIDA EM C# PARA .NET REMOTING

```
using System;
namespace IRemoting
{
    public interface ContaBancaria
    {
        double SaldoActual
        {
            get;
        }
        void depositar ( float quantia );
        void levantar (float quantia);
    }
}
```

Permite definir atributos acessíveis
por operações associadas (get/set)

Interface definida em C#
comum

INTERFACE DEFINIDA EM C# PARA .NET REMOTING

```
using System;
namespace IRemoting
{
    public interface ContaBa
    {
        double SaldoActu
        {
            get;
        }
        void depositar ( float
        void levantar (float qu
    }
}
```

```
public class ServiceClass :
    System.MarshalByRefObject
{
    public void AddMessage (String msg) {
        Console.WriteLine (msg);
    }
}
```

No .NET Remoting não é necessário definir qual a interface remota – esta pode ser inferida a partir da definição do servidor

Um objecto remoto deve estender MarshalByRefObject

WSDL – IDL PARA *WEB SERVICES*

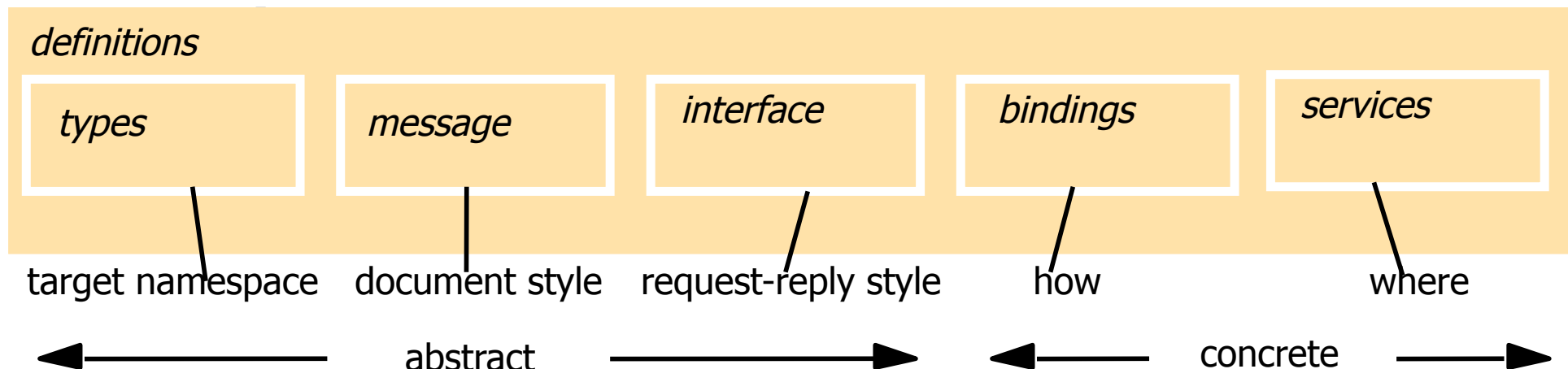
Definição da interface em XML

WSDL permite definir a interface do serviço, indicando quais as mensagens trocadas na interacção

WSDL permite também definir a forma de representação dos dados e a forma de aceder ao serviço

Especificação WSDL bastante verbosa – normalmente criada a partir de interface ou código do servidor

Ex. JAX-WS tem ferramentas para criar especificação a partir de interfaces



WSDL - EXEMPLO

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
  targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="SayHelloRequest">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="SayHelloResponse">
    <part name="greeting" type="xsd:string"/>
  </message>

  <portType name="Hello_PortType">
    <operation name="sayHello">
      <input message="tns:SayHelloRequest"/>
      <output message="tns:SayHelloResponse"/>
    </operation>
  </portType>
</definitions>
```

(exemplo do livro Web Services Essentials, O'Reilly, 2002.)

WSDL - EXEMPLO

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
  targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
  <message name="SayHelloRequest">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="SayHelloResponse">
    <part name="greeting" type="xsd:string"/>
  </message>
```

```
  <portType name="Hello_PortType">
    <operation name="sayHello">
      <input message="tns:SayHelloRequest"/>
      <output message="tns:SayHelloResponse"/>
    </operation>
  </portType>
```

<definitions>: The HelloService

<message>:

- 1) sayHelloRequest: firstName parameter
- 2) sayHelloResponse: greeting return value

<portType>: sayHello operation that consists of a request/response service

<binding>: Direction to use the SOAP HTTP transport protocol.

<service>: Service available at: http://localhost:8080/soap/servlet/rpcrouter

WSDL - EXEMPLO

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
  targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="SayHelloRequest">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="SayHelloResponse">
    <part name="greeting" type="xsd:string"/>
  </message>

  <portType name="Hello_PortType">
    <operation name="sayHello">
      <input message="tns:SayHelloRequest"/>
      <output message="tns:SayHelloResponse"/>
    </operation>
  </portType>
```

<definitions>: The HelloService

<message>:

- 1) sayHelloRequest: firstName parameter
- 2) sayHelloResponse: greeting return value

<portType>: sayHello operation that consists of a request/response service

<binding>: Direction to use the SOAP HTTP transport protocol.

<service>: Service available at: http://localhost:8080/soap/servlet/rpcrouter

WSDL - EXEMPLO

```
<binding name="Hello_Binding" type="tns:Hello_PortType">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sayHello">
    <soap:operation soapAction="sayHello"/>
    <input>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:examples:helloservice"
        use="encoded"/>
    </input>
    <output>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:examples:helloservice"
        use="encoded"/>
    </output>
  </operation>
</binding>
```

```
<service name="Hello Service">
  <documentation>WSDL File for Hello Service</documentation>
  <port binding="tns:Hello_Binding">
    <soap:address
      location="http://localhost:8080/soap/servlet/rpcrouter"/>
  </port>
</service>
</definitions>
```

<definitions>: The HelloService

<message>:

- 1) sayHelloRequest: firstName parameter
- 2) sayHelloResponse: greeting return value

<portType>: sayHello operation that consists of a request/response service

<binding>: Direction to use the SOAP HTTP transport protocol.

<service>: Service available at: http://localhost:8080/soap/servlet/rpcrouter

WSDL - EXEMPLO

```
<binding name="Hello_Binding" type="tns:HelloRequestResponse" style="rpc"
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"
  <operation name="sayHello">
    <soap:operation soapAction="sayHello"
    <input>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:examples:HelloService"
        use="encoded" />
    </input>
    <output>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:examples:HelloService"
        use="encoded" />
    </output>
  </operation>
</binding>
```

```
<service name="Hello_Service">
  <documentation>WSDL File for HelloService</documentation>
  <port binding="tns:Hello_Binding" name="Hello_Port">
    <soap:address
      location="http://localhost:8080/soap/servlet/rpcrouter" />
  </port>
</service>
</definitions>
```

<definitions>: The HelloService

<message>:

- 1) sayHelloRequest: firstName parameter
- 2) sayHelloResponse: greeting return value

<portType>: sayHello operation that consists of a request/response service

<binding>: Direction to use the SOAP HTTP transport protocol.

<service>: Service available at: http://localhost:8080/soap/servlet/rpcrouter

MÉTODOS DE PASSAGEM DE PARÂMETROS

Independentemente dos tipos dos parâmetros, os mesmos podem ser:

- parâmetros de entrada (in) : cópia no pedido

- parâmetros de saída/resultado (out) : cópia na resposta

- parâmetros de entrada/saída (in/out) : cópia no pedido e na resposta

MÉTODOS DE PASSAGEM DE PARÂMETROS (CONT.)

Aproximação comum nos sistemas de RPC/RMI:

Passagem por valor para tipos básicos, arrays, estruturas e objectos não remotos

Apontadores/referências para arrays, objectos, etc. são seguidas

Estado dos objectos é copiado (ex: Java RMI)

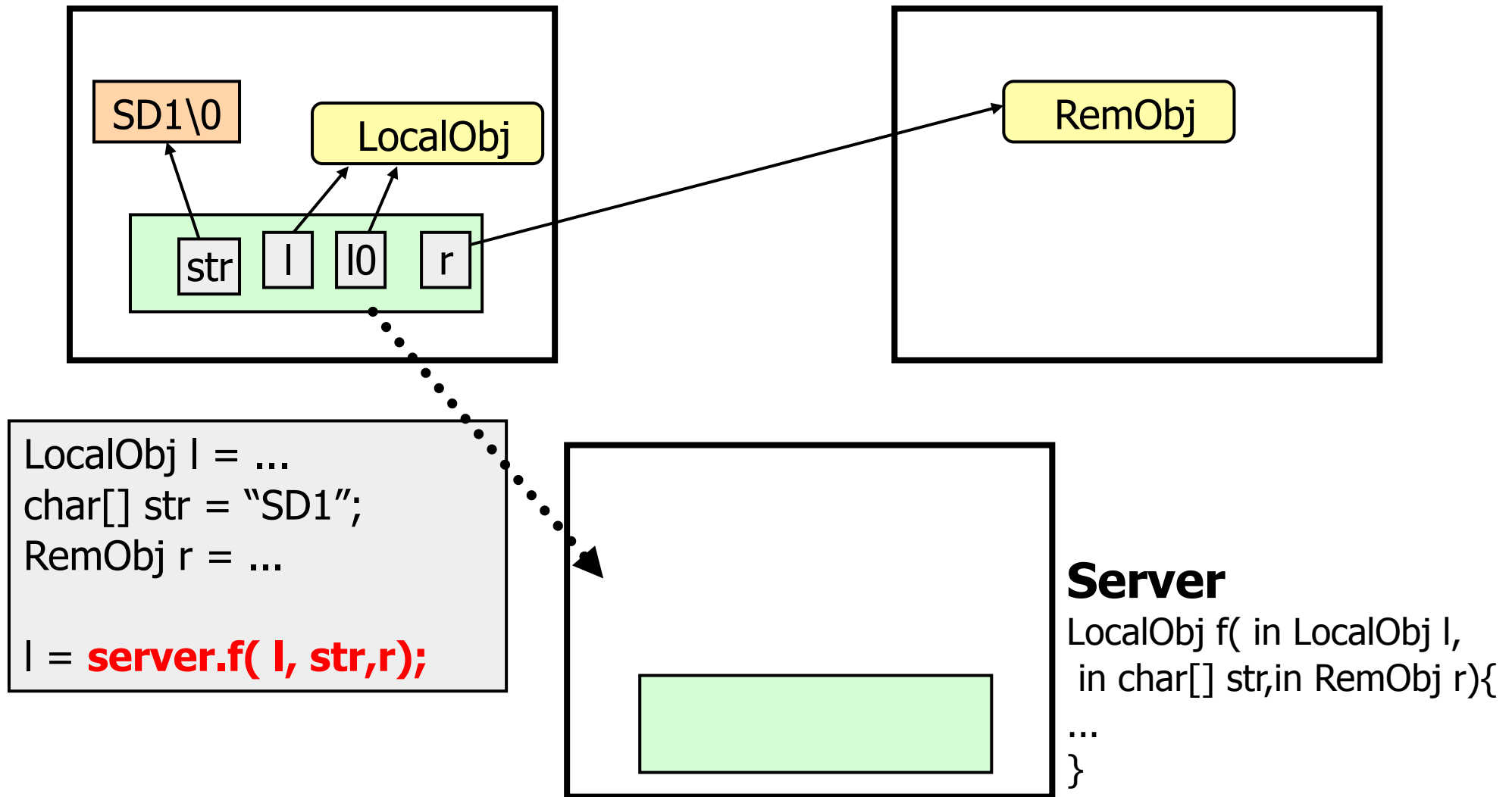
Porque não passar tipos básicos por referência?

Passagem por referência para objectos remotos

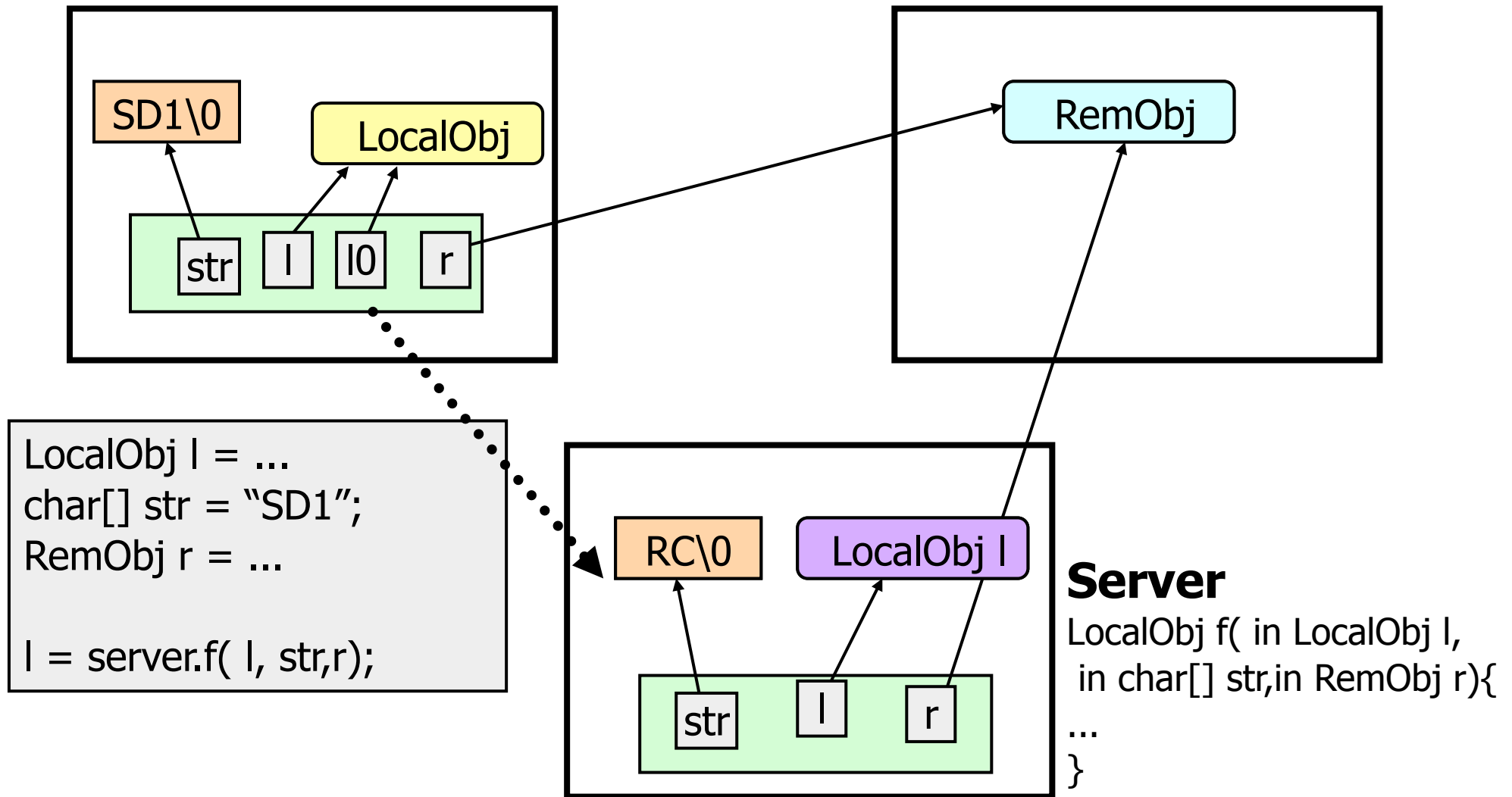
quando o tipo de um parâmetro é um objecto remoto, uma referência para o objecto é transferida

Porque não passar objectos remotos por valor?

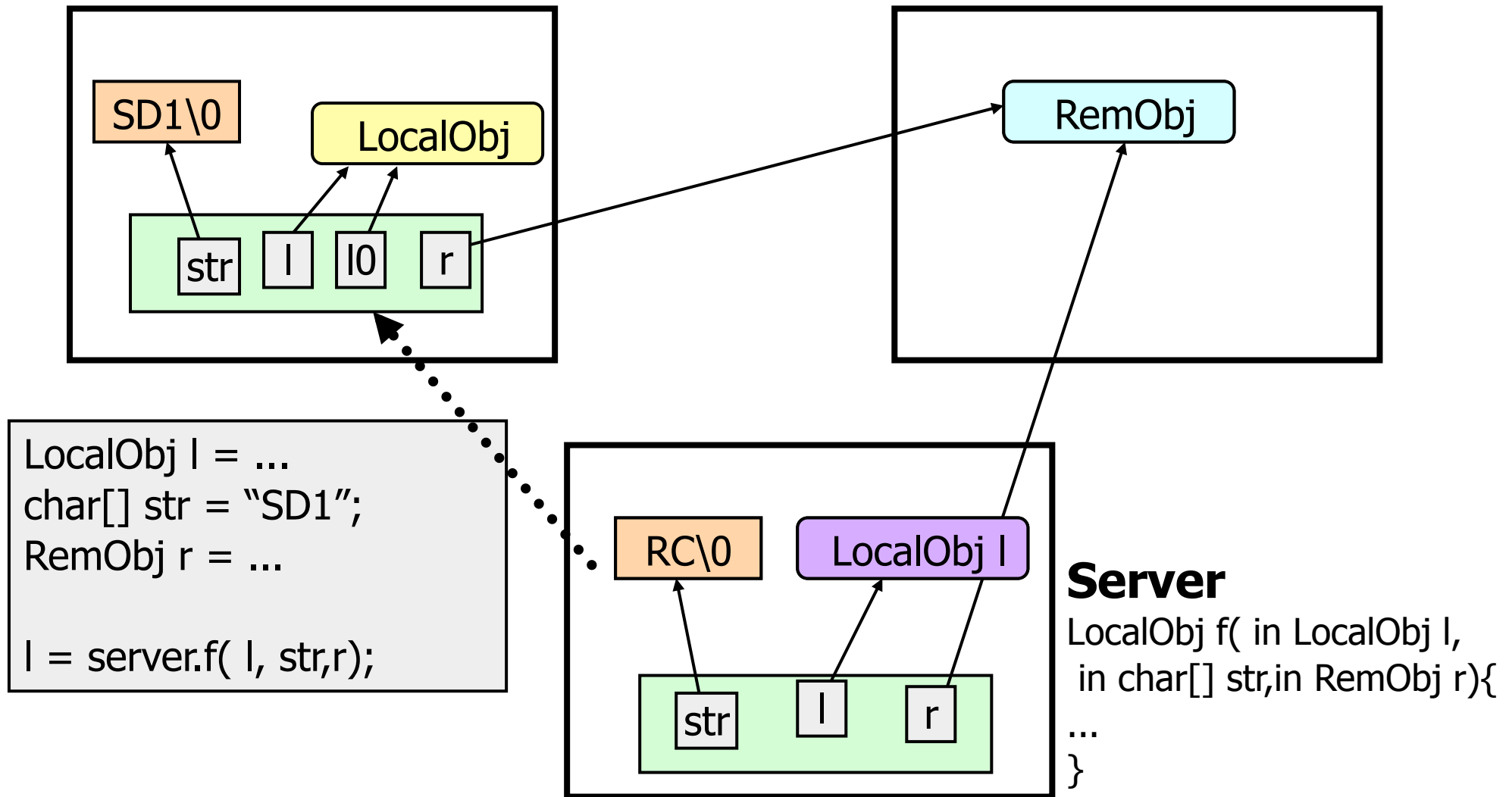
MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



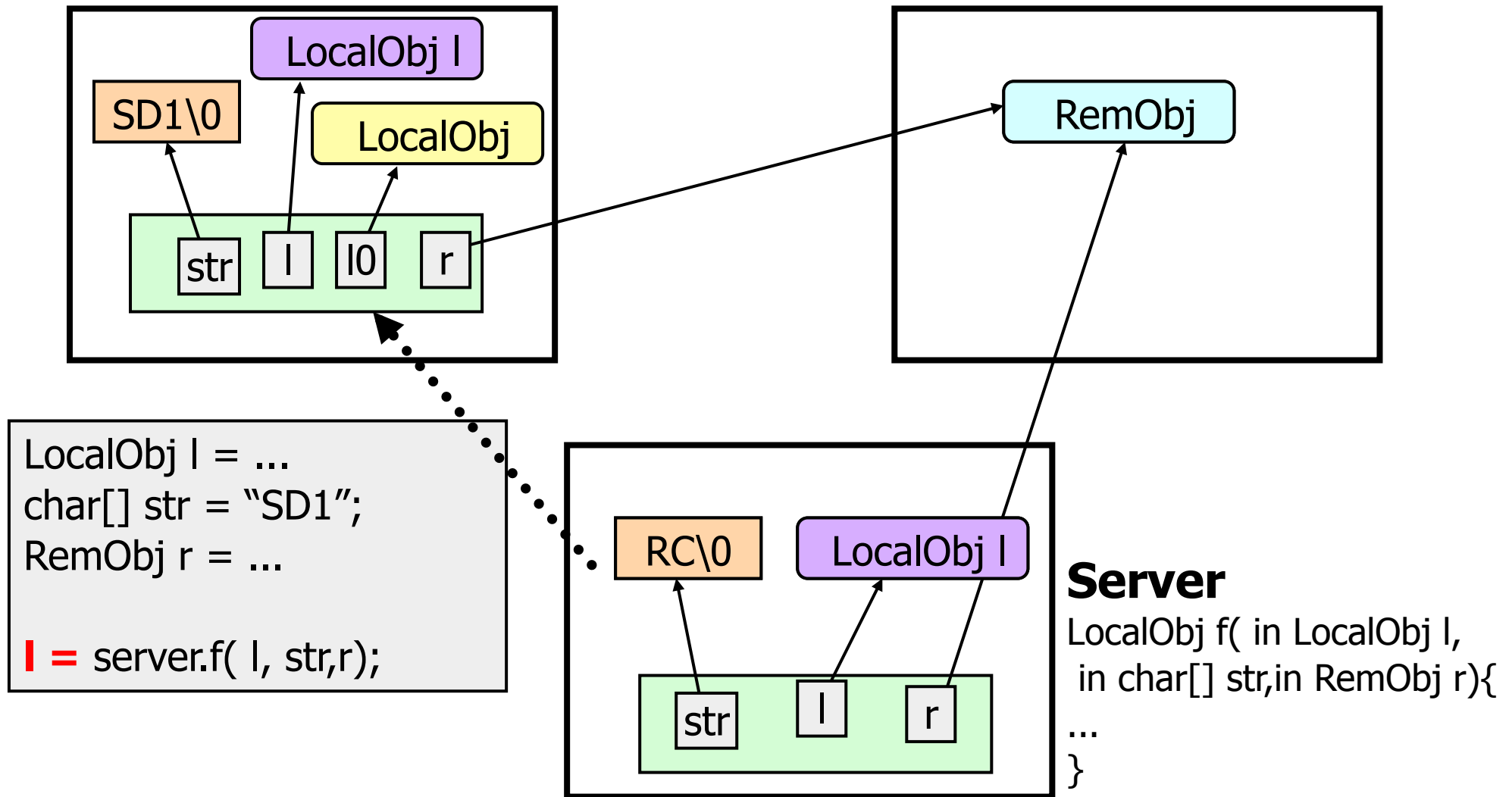
MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



AGENDA

Invocação remota de procedimentos/objectos

Motivação

Modelo

Definição de interfaces e método de passagem de parâmetros

Codificação dos dados

Mecanismos de ligação (binding)

Protocolos de comunicação

Concorrência no servidor

Sistemas de objetos distribuídos

CODIFICAÇÃO DOS DADOS - PROBLEMA

Como representar dados trocados entre os clientes e os servidores?

CODIFICAÇÃO DOS DADOS - PROBLEMA

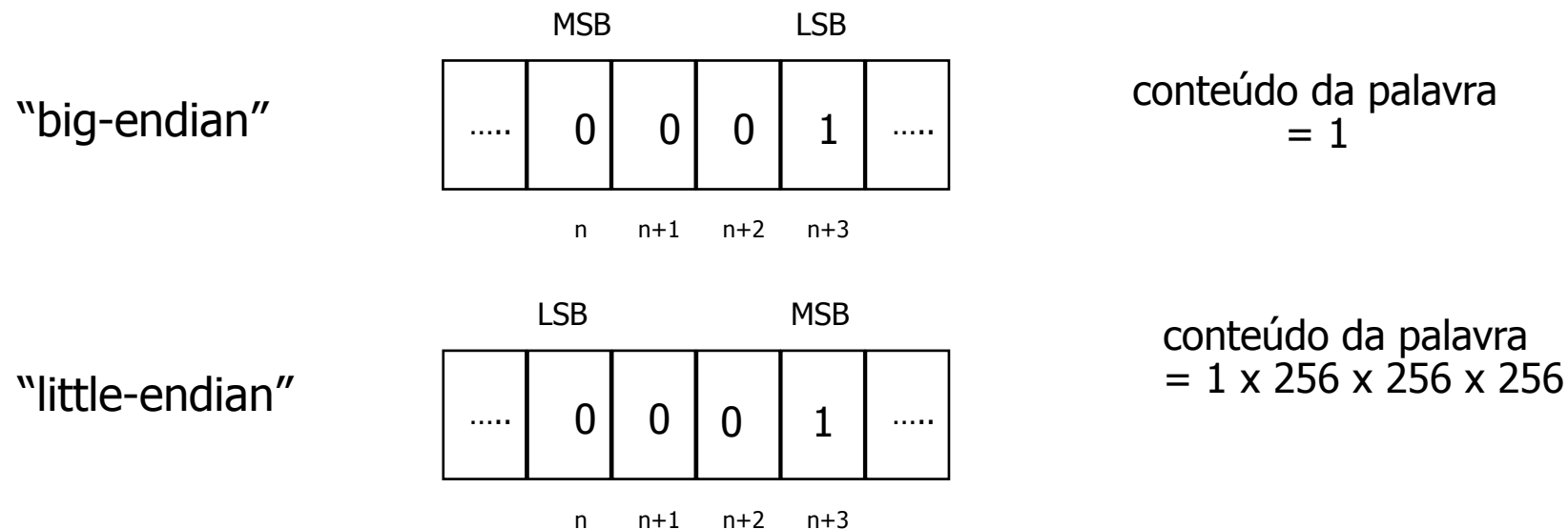
Diferentes sistemas representam os tipos primitivos de formas diferentes

Inteiros armazenados por ordem diferente em memória

Diferentes representações para números reais

Caracteres com diferentes codificações

Simple transmissão dos valores armazenados pode levar a resultados errados



REPRESENTAÇÕES DOS DADOS – TIPOS COMPLEXOS

Aplicações manipulam estruturas de dados complexas

Ex.: representadas por grafos de objectos

Mensagens são sequências de bytes

O que é necessário fazer para propagar estrutura de dados complexa?

É necessário convertê-la numa sequência de bytes

Por exemplo, para um objecto é necessário:

Converter as variáveis *internas*, incluindo outros objectos

Necessário lidar com ciclos nas referências

Marshalling – processo de codificar do formato interno para o formato rede

Unmarshalling – processo de decodificar do formato rede para o formato interno

APROXIMAÇÕES À CODIFICAÇÃO DOS DADOS

Utilização de formato intermédio independente (network standard representation)

Emissor converte da representação nativa para a representação da rede

O receptor converte da representação da rede para a representação standard

Utilização do formato do emissor (receiver makes it right)

Emissor envia usando a sua representação interna e indicando qual ela é

Receptor, ao receber, faz a conversão para a sua representação

Utilização do formato do receptor (sender makes it right)

Propriedades:

Desempenho ?

rep. intermédia tem pior desempenho - exige duas transformações

Complexidade (número de transformações a definir) ?

rep. intermédia exige apenas que em cada plataforma se saiba converter de/para formato intermédio

CDR - CORBA COMMON DATA REPRESENTATION

CDR primitive or simple data types

Tipo	Wire size (bytes)	Descrição
short	2	16 bits signed binary integer
unsigned short	2	16 bits unsigned binary integer
long	4	32 bits signed binary integer
unsigned long	4	32 bits unsigned binary integer
float	4	single precision floating point number
double	8	double precision floating point number
Boolean	4	boolean value (0 or 1)
char	1	ASCII char
octet	1	any 8 bits
any		any type

Inteiros são *big-endian* ou *little-endian* **de acordo com a ordem do emissor** (enviada em cada mensagem).

Floats são IEEE, também *big-endian* ou *little-endian*

CORBA CDR WIRE MESSAGE

<i>index in sequence of bytes</i> ← 4 bytes →		<i>notes on representation</i>	
0–3	5	<i>length of string</i>	<pre>struct Person { string name; string place; long year; }; {'Smith', 'London', 1934}</pre>
4–7	"Smit"	<i>'Smith'</i>	
8–11	"h _ _ _"		
12–15	6	<i>length of string</i>	
16–19	"Lond"	<i>'London'</i>	
20–23	"on _ _"		
24–27	1934	<i>unsigned long</i>	

Dados com dimensão variável são alinhados preenchendo com 0s posições não usadas

Não é enviada informação sobre os tipos. Como funciona?

Assume-se que o emissor e o receptor sabem os tipos que esperam

São geradas funções para fazer o *marshalling* e o *unmarshalling*

JAVA SERIALIZATION

Serialized values

Person	8-byte version number		h0
3	int year	java.lang.String name	java.lang.String place
1934	5 Smith	6 London	h1

Explanation

class name, version number

number, type and name of instance variables

values of instance variables

The true serialized form contains additional type markers; h0 and h1 are handles

```
public class Person
    implements Serializable
{
    private String name;
    private String place;
    private int year;
    ...
}
```

Assume-se que o processo de *deserialization* não tem informação sobre os objectos serializados

Forma serializada inclui informação dos tipos

Serialização grava estado de um grafo de objectos

A cada objecto é atribuído um *handle*. Permite escrever apenas uma vez cada objecto, mesmo quando existem várias referências para o mesmo no grafo de objectos.

SERIALIZAÇÃO DE OBJECTOS

Permite codificar/descodificar grafos de objectos

Detecta e preserva ciclos pois incorpora a identidade dos objectos no grafo

Adaptável em cada classe (os métodos responsáveis podem ser redefinidos)

Os objectos devem ser serializáveis

por omissão não são – porquê?

poderia abrir problemas de segurança. Exemplo?

Permitia acesso a campos private, por exemplo.

Os campos static e transient não são serializados

Usa *reflection* – permite obter informação sobre os tipos em runtime

Assim, não necessita de funções especiais de marshalling e unmarshalling

EXTENSIBLE MARKUP LANGUAGE (XML)

XML permite descrever estruturas de dados complexas

Tags usadas para descrever a estrutura dos dados

Permite associar pares atributo/valor com a estrutura lógica

XML é extensível

Novas tags definidas quando necessário

Num documento XML toda a informação é textual

Podem-se codificar valores binários, por exemplo, em base64

No contexto dos sistemas de RPC/RMI, o XML pode ser usado para:

Codificar parâmetros em sistemas de RPC

Codificar invocações (SOAP)

Etc.

```
<person id="123456789">  
  <name>Smith</name>  
  <place>London</place>  
  <year>1934</year>  
  <!-- a comment -->  
</person >
```

```
<?xml version="1.0"?>  
<methodCall>  
  <methodName>inc</methodName>  
  <params>  
    <param>  
      <value><i4>41</i4></value>  
    </param>  
  </params>  
</methodCall>
```

EXTENSIBLE MARKUP LANGUAGE (XML)

XML permite descrever estruturas de dados complexas

Tags usadas para descrever a estrutura dos dados

```
<person id="123456789">  
  <name>Smith</name>  
  <place>London</place>
```

```
<?xml version='1.0' encoding='UTF-8'?>  
<soap:Envelope xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'  
  xmlns:xsd='http://www.w3.org/2001/XMLSchema' xmlns:soap='http://  
schemas.xmlsoap.org/soap/envelope/' xmlns:soapenc='http://  
schemas.xmlsoap.org/soap/encoding/' soap:encodingStyle='http://  
schemas.xmlsoap.org/soap/encoding/'>  
  <soap:Body>  
    <n:sayHello xmlns:n='urn:examples:helloservice'>  
      <firstName xsi:type='xsd:string'>World</firstName>  
    </n:sayHello>  
  </soap:Body>  
</soap:Envelope>
```

Etc.

```
</param>  
</params>  
</methodCall>
```

XML SCHEMA / XML NAMESPACES

Um XML namespace permite criar espaço de nomes para os nomes dos elementos e atributos usados nos documentos XML

Um XML schema define os elementos e atributos que podem aparecer num documento XML

```
<xsd:schema xmlns:xsd = URL of XML schema definitions >
  <xsd:element name= "person" type = "personType" />
  <xsd:complexType name="personType">
    <xsd:sequence>
      <xsd:element name="name" type="xs:string"/>
      <xsd:element name="place" type="xs:string"/>
      <xsd:element name="year" type="xs:positiveInteger"/>
    </xsd:sequence>
  <xsd:attribute name= "id" type = "xs:positiveInteger"/>
</xsd:complexType>
</xsd:schema>
```

XML SCHEMA / XML NAMESPACES

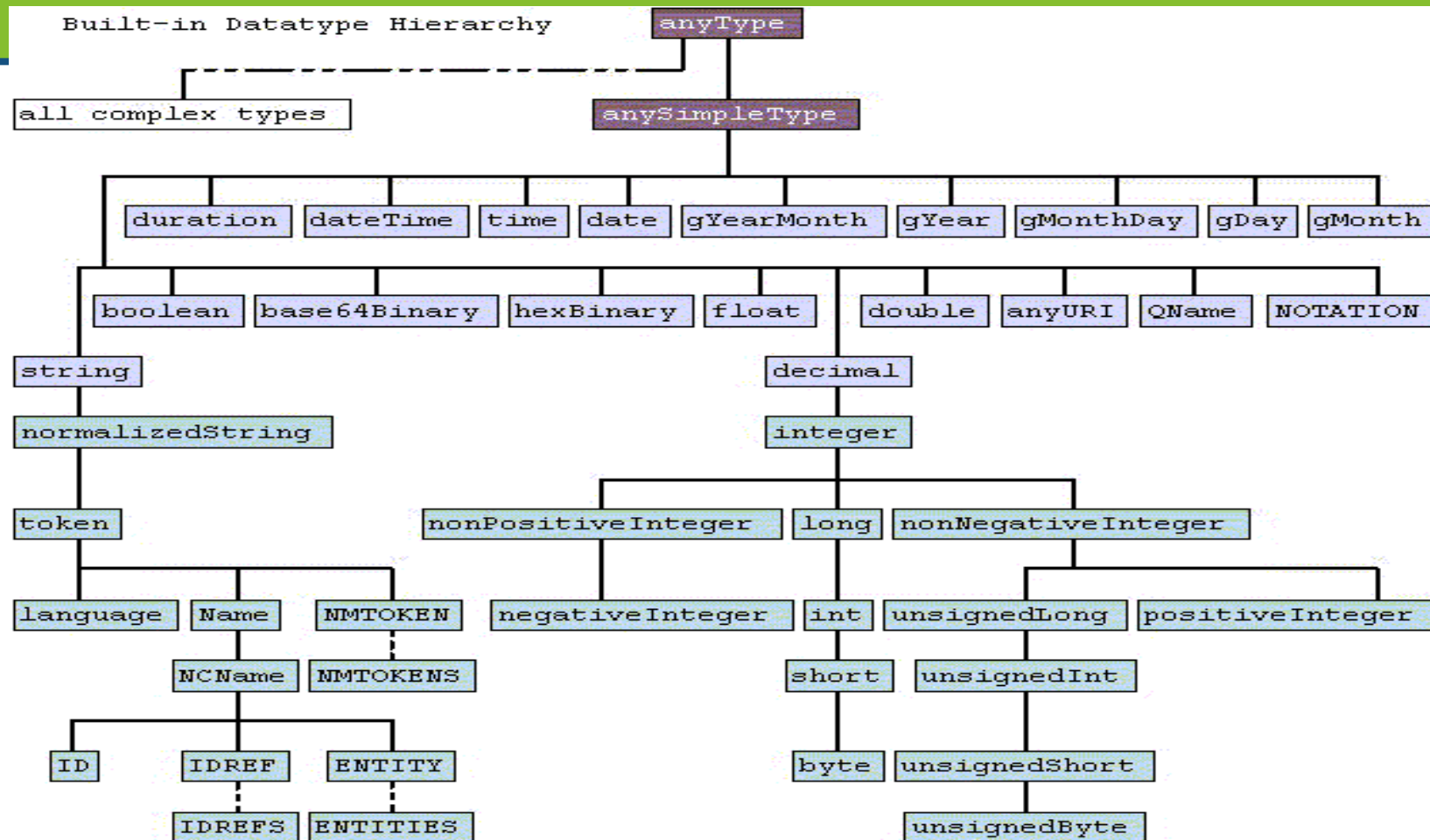
Um XML namespace permite criar espaço de nomes para os nomes dos elementos e atributos usados

Um XML schema define os elementos aparecer num documento XML

```
<person id="123456789">  
  <name>Smith</name>  
  <place>London</place>  
  <year>1934</year>  
</person >
```

```
<xsd:schema xmlns:xsd = URL of XML schema definitions >  
  <xsd:element name= "person" type ="personType" />  
  <xsd:complexType name="personType">  
    <xsd:sequence>  
      <xsd:element name="name" type="xs:string"/>  
      <xsd:element name="place" type="xs:string"/>  
      <xsd:element name="year" type="xs:positiveInteger"/>  
    </xsd:sequence>  
    <xsd:attribute name= "id" type = "xs:positiveInteger"/>  
  </xsd:complexType>  
</xsd:schema>
```

TIPOS XML



ur types



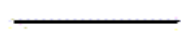
built-in primitive types



built-in derived types



complex types



derived by restriction



derived by list



derived by extension or
restriction

REPRESENTAÇÕES DOS DADOS: CLASSIFICAÇÃO

Conteúdo da representação

- Formato binário – Corba, Java

- Formato de texto – XML

Integração com linguagem

- Independente – Corba, XML

- Integrado – Java

Informação de tipos

- Incluída – Java, XML

- Não incluída – Corba

PASSAGEM DE OBJECTOS REMOTOS EM PARÂMETRO

Nos sistemas de RMI é, em geral, possível passar (referências para) objectos remotos em parâmetro (ou como resultado de uma operação)

Em Java RMI pode-se enviar uma referência para um objecto remoto:

- Passando como parâmetro/resultado uma referência remota – neste caso, uma cópia da referência remota é enviada

- Passando como parâmetro/resultado o objecto servidor – neste caso, uma referência para o objecto remoto é enviada (e não o próprio objecto) – passagem por referência

Uma referência remota inclui, pelo menos, a seguinte informação:

- Endereço/porta do servidor

- Tipo do servidor

- Identificador único

PASSAGEM DE OBJECTOS REMOTOS EM PARÂMETRO

Nos sistemas de RMI é, em geral, possível passar (referências para) objectos remotos em parâmetro (ou como resultado de uma operação)

Em Java RMI pode-se enviar uma referência para um objecto remoto

Passando como parâmetro/resultado
cópia da referência remota é enviada

Passando como parâmetro/resultado
referência para o objecto remoto é enviada (e não o próprio objecto) –
passagem por referência

Com esta representação seria fácil
mudar a localização do objecto?

Não. Para tal, a referência remota
não deve incluir directamente a
localização do objecto.

uma

Uma referência remota inclui, pelo menos, a seguinte informação:

Endereço/porta do servidor

Tipo do servidor

Identificador único

AGENDA

Invocação remota de procedimentos/objectos

Motivação

Modelo

Definição de interfaces e método de passagem de parâmetros

Codificação dos dados

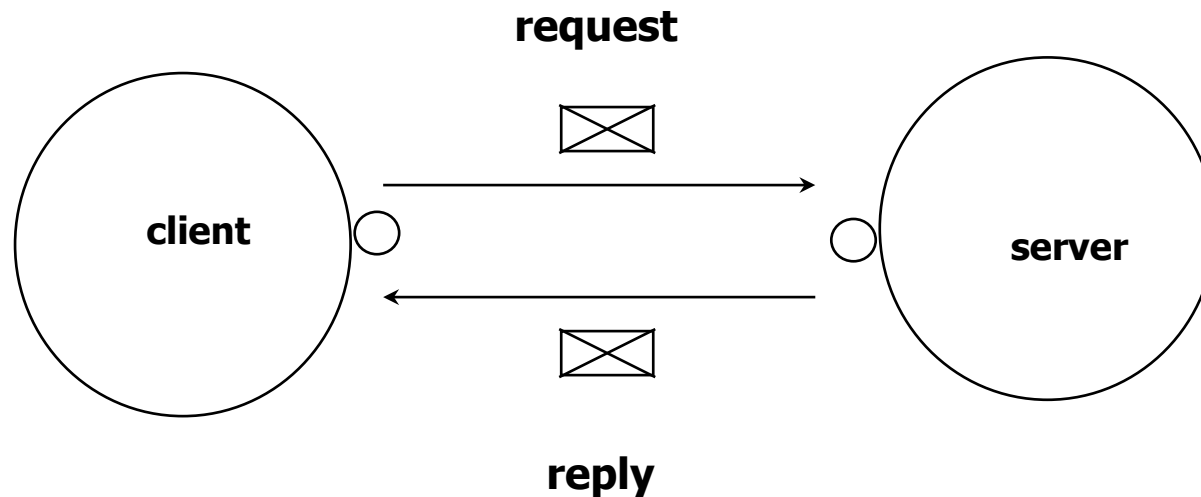
Mecanismos de ligação (binding)

Protocolos de comunicação

Concorrência no servidor

Sistemas de objetos distribuídos

LIGAÇÃO DO CLIENTE AO SERVIDOR (*BINDING*)



Para poder invocar o servidor, o cliente tem de obter uma referência para o servidor

Nos sistemas de RPC, referência corresponde ao communication end-point do servidor – endereço IP + porta + ...

Nos sistemas de RMI, a referência remota corresponde geralmente a um proxy com a mesma interface do servidor (que internamente inclui informação de localização do servidor)

Como obter essa referência?

COMO OBTER REFERÊNCIA PARA O SERVIDOR?

Por configuração directa

Ex.: Web services, .NET remoting

Servidor de nomes regista associação entre nome e referência remota

Ex.: Java RMI, Corba

Servidor de nomes e directório regista informação sobre servidores

Ex. Universal Directory and Discovery Service – UDDI (web services)

Além de permitir obter servidor dado o nome, permite procurar servidor pelos seus atributos

Cliente procura servidor usando multicast/broadcast

Alguns sistemas de objectos distribuídos usavam esta aproximação

POR CONFIGURAÇÃO DIRECTA

No código do cliente, para obter uma referência para o servidor indica-se explicitamente a sua localização

.NET remoting

```
ChannelServices.RegisterChannel(new TcpChannel());  
HelloServer obj = (HelloServer)Activator.GetObject(  
    typeof(Examples.HelloServer),  
    "tcp://localhost:8085/SayHello");
```

Web services

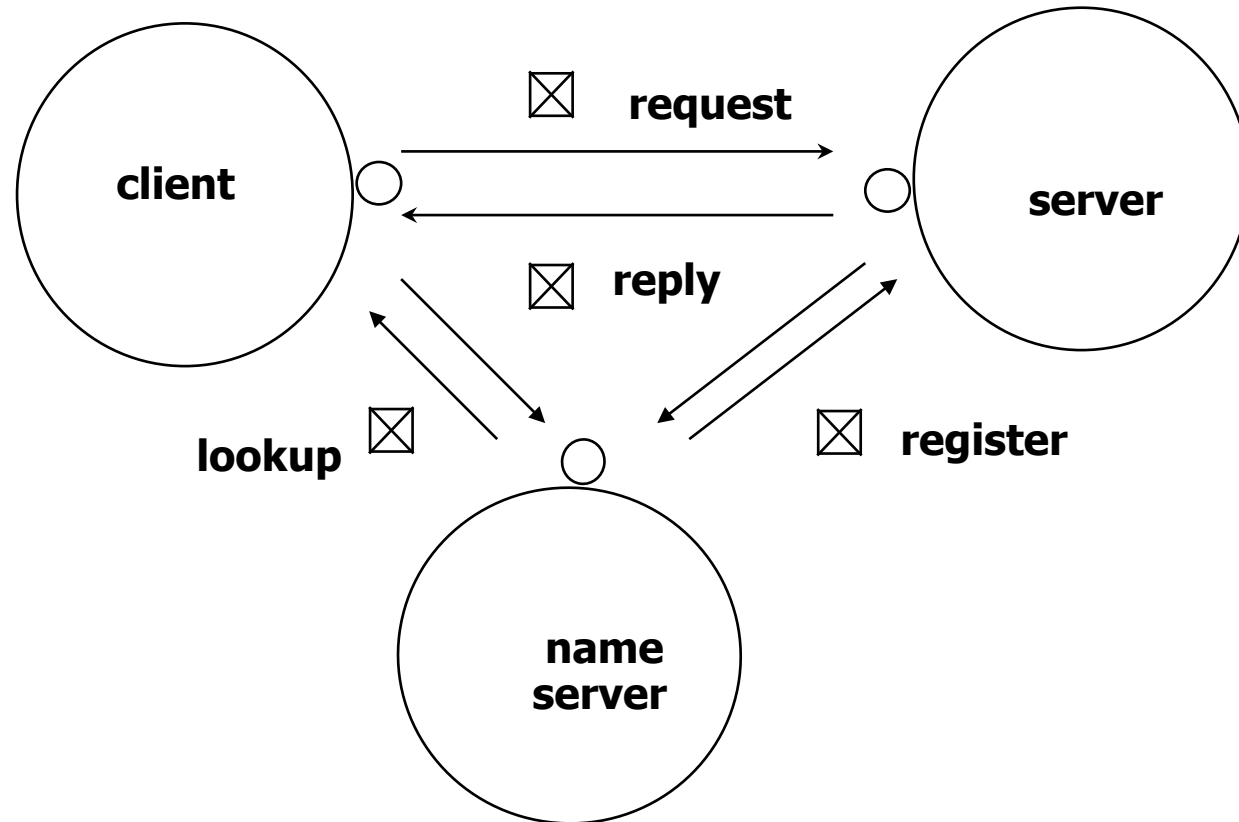
```
NetServerImplWSService service = new NetServerImplWSService(  
    new java.net.URL("http://host/xpto"), ...);
```

Ao criar o código da referência remota a partir da descrição do serviço, a mesma inclui a localização do servidor

Web services

```
NetServerImplWSService service = new NetServerImplWSService();
```

USAR UM SERVIÇO (*BINDING, NAMING OR TRADING*)



EXEMPLO: INTERFACE DA *REGISTRY* JAVA RMI

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name.

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. A remote object reference is returned. The code of the remote reference may be downloaded, if necessary. It encodes the protocol to be used.

String [] list()

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

PROBLEMAS QUANDO SE USA UM SERVIDOR DE NOMES?

Como encontrar o name server ?

Nome do objecto indica máquina em que está o name server

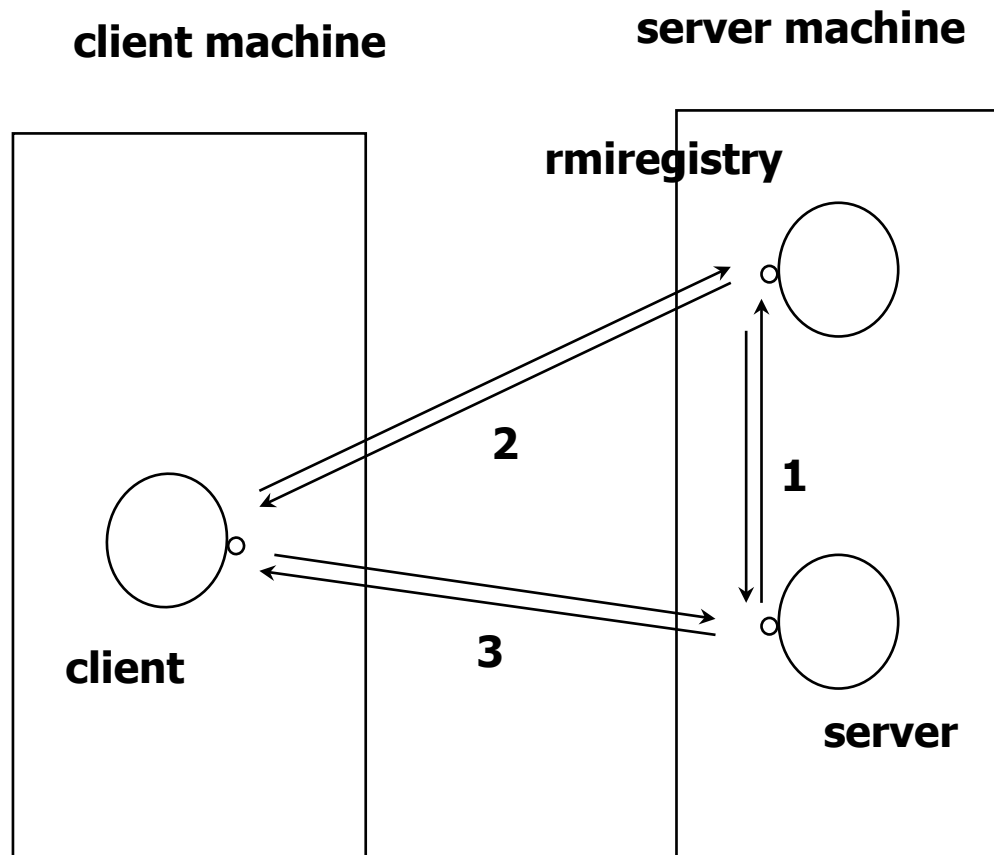
Name server descoberto por multicast/broadcast

O name server pode ser único ?

O sistema CORBA define um serviço de nomes em que o espaço de nomes é único e global. A implementação depende do sistema.

Problemas de segurança – como evitar que haja serviços / objectos impostores ou que atacantes impeçam o acesso ao serviço?

SOLUÇÃO PRAGMÁTICA DO JAVA RMI



Em cada máquina existe um RMI registry. O cliente tem de saber em que máquina está o serviço/objecto remoto em que está interessado.

Em cada máquina, só pode estar associado uma instância de uma interface/classe a cada nome
Pode existir mais do que uma instância da mesma interface/classe associados a nomes diferentes

Porque é que esta solução diminui os problemas de segurança?

AGENDA

Invocação remota de procedimentos/objectos

Motivação

Modelo

Definição de interfaces e método de passagem de parâmetros

Codificação dos dados

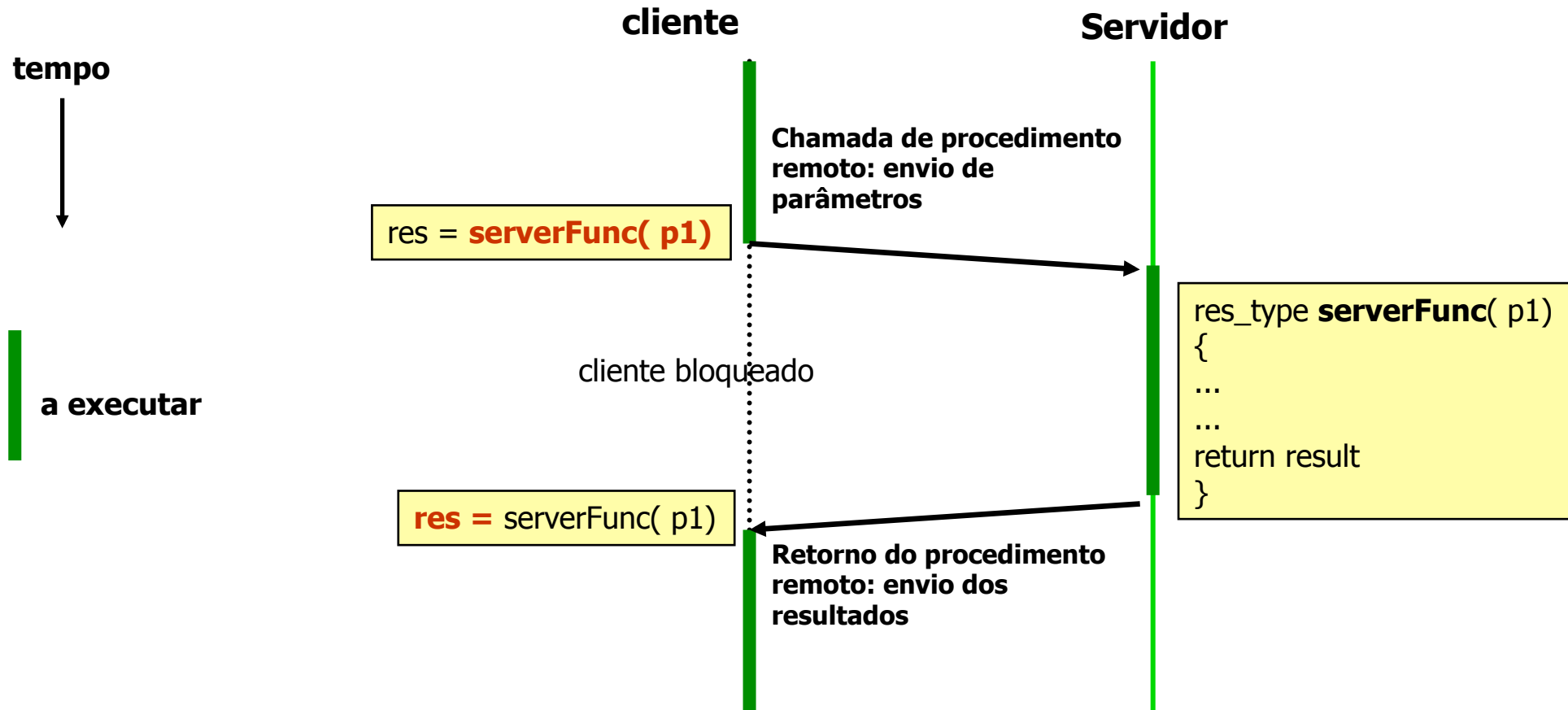
Mecanismos de ligação (binding)

Protocolos de comunicação

Concorrência no servidor

Sistemas de objetos distribuídos

MODELO DE FALHAS



Admitindo que o canal não introduz falhas arbitrárias, podemos ter:

Canal: falhas de omissão e temporização

Cliente e servidor: crash-failures e falhas de temporização

ANOMALIAS POSSÍVEIS DURANTE UMA INVOCACÃO REMOTA

Anomalias a considerar:

- a mensagem com o pedido pode perder-se
- a mensagem com a resposta pode perder-se
- o servidor está muito lento e aparentemente não responde
- o servidor falha e não responde
 - o servidor falha e recupera mais tarde (quando ?)
- o cliente falha e recupera

Algumas destas situações podem ser facilmente detectáveis, outras não.

PROTOCOLO CONNECTION-LESS VS. CONNECTION-ORIENTED

Motivações para o uso do UDP:

- Estabelecer uma conexão tem um peso que se pode revelar demasiado pesado (para pedidos pontuais e de pequenas dimensões)
- Resposta à invocação remota funciona como ACK (não é necessário suportar peso dos mecanismos de fiabilidade incluídos no TCP)

Motivações para o uso de TCP (ou HTTP):

- Dimensão dos parâmetros/resultados não influencia complexidade da implementação
- No caso de usar UDP pode ser necessário enviar um pedido/resposta em mais do que uma mensagem

PROTOCOLO *CONNECTION-ORIENTED*

Hipótese: o cliente usa o protocolo TCP para contactar o servidor e enquanto não receber resposta a um pedido, ou não decidir abandonar um pedido, não avança com outro pedido

Situação semelhante quando se usa protocolo HTTP

1) O cliente fez um pedido e recebeu a resposta:

O cliente tem a certeza que o procedimento executou (uma e uma só vez)

2) O cliente fez o pedido e não recebeu resposta nenhuma até um certo *timeout*.
Decidiu então abandonar o pedido

Que se passou ? Não se sabe se a operação foi executada ou não.

3) O cliente fez o pedido e recebeu uma notificação de que a conexão foi quebrada
(por falha na comunicação ou por *crash* no servidor)

Que se passou ? Não se sabe se a operação foi executada ou não.

PROTOCOLO *CONNECTION-LESS* NÃO FIÁVEL (E.G. UDP)

Hipótese: o cliente usa o protocolo UDP para contactar o servidor e enquanto não receber resposta a um pedido, ou não decidir abandonar um pedido, não avança com outro pedido

1) O cliente enviou uma só vez o pedido e recebeu a resposta

O cliente tem a certeza que o procedimento se executou (uma e uma só vez... assumindo que não existe duplicação de pacotes)

2) O cliente fez o pedido e não recebeu resposta nenhuma até um certo timeout

Que se passou ? Não se sabe se a operação foi executada ou não.

INTRODUÇÃO DE RE-EMISSÕES

Nas situações em que o cliente não recebe resposta até um dado timeout poder-se-ia introduzir um mecanismo de re-emissão do pedido.

No caso *connection-less*, reenviando a mensagem

No caso *connection-oriented*, se a conexão se quebrou, criando uma nova conexão

Quantas vezes se re-emite ?

EM CASO DE ANOMALIA, APÓS RETRANSMISSÃO, O QUE SE PASSOU?

Caso o cliente não receba resposta ao seu pedido até um timeout após, pelo menos, uma retransmissão, e deseje abandonar, podem ter acontecido duas situações:

- A anomalia ocorreu antes de executar a operação em todas as retransmissões

- A anomalia ocorreu após a execução da operação em uma ou mais retransmissões

Sabe-se que: a operação foi executada zero, uma ou mais vezes.

Caso o cliente receba resposta ao seu pedido após, pelo menos, uma retransmissão, podem ter acontecido duas situações:

- Na tentativa de transmissão anterior, a anomalia ocorreu após a execução da operação

- Na tentativa de transmissão anterior, a anomalia ocorreu antes de executar a operação

Sabe-se que: a operação foi executada uma vez ou mais vezes.

FILTRAGEM DE DUPLICADOS

Problema: como evitar que um pedido seja executado mais do que uma vez no servidor?

O cliente numera com um número de sequência único cada pedido que faz. Re-emite enquanto não receber resposta até um número limite de vezes.

O servidor quando recebe um pedido novo, executa a operação e guarda numa “cache” a resposta dada. Caso receba uma re-emissão devolve o resultado previamente calculado (“cached”) pois a resposta pode ter-se perdido.

Problemas?

Para tolerar crashes e recuperações do servidor, a cache com resultados deve ser mantida em memória estável

FILTRAGEM DE DUPLICADOS (CONT.)

Problema: quando é que o servidor pode remover a informação sobre resultados de pedidos antigos?

O servidor pode remover informação quando souber que já não vai ser necessária:

- Cliente pode informar o servidor que recebeu o resultado (ack)

- Cliente pode iniciar um novo pedido (equivalente a já ter recebido resultado do anterior)

Caso não inicie um novo pedido e o ack da resposta se perder, o servidor pode anular a informação ao fim de um tempo alargado.

SEMÂNTICA DA INVOCAÇÃO REMOTA

May be (talvez): método pode ter sido executado uma vez ou nenhuma vez

usa-se quando não se esperam resultados; não tem garantias. O interesse é muito limitado.

Protocolo?

Implementado por protocolo de envio simples de mensagem (sem resposta nem retransmissões)

SEMÂNTICA DA INVOCAÇÃO REMOTA (CONT.)

At least once (uma ou mais vezes ou “pelo menos uma vez”):

1. se cliente recebeu a resposta, o procedimento foi executado uma ou mais vezes
2. se o cliente não recebeu a resposta, o procedimento foi executado zero ou mais vezes

Protocolo ?

Implementado por protocolo com re-emissões e sem filtragem de duplicados

SEMÂNTICA DA INVOCAÇÃO REMOTA (CONT.)

At most once (zero ou uma vez ou “no máximo uma vez”):

1. Se cliente recebeu a resposta, o procedimento foi executado uma só vez
2. Se o cliente não recebeu a resposta, o procedimento foi executado zero ou uma vezes

Protocolo ?

Protocolo sem re-emissões ou protocolo com re-emissões e filtragem de duplicados

SEMÂNTICA DA INVOCÇÃO REMOTA (CONT.)

- **Exactly once (exactamente uma vez):**
 - É necessário registar em memória estável o estado do cliente e servidor para:
 - Quando o cliente recupera de uma falha, deve retomar a execução da invocção
 - Quando o servidor recupera de uma falha, deve recuperar o seu estado (incluindo informação sobre as operações executadas)
 - Nota: a indisponibilidade do serviço durante a falha não é mascarada
 - Protocolo ?
 - Protocolo com re-emissões, filtragem de duplicados e manutenção do estado em memória estável

PROTOCOLOS DE INVOCACÃO REMOTA (RESUMO)

Com transporte connection-oriented (TCP)

Request / Reply protocol (RR): semântica?

Implementa facilmente a política *no máximo uma vez*. Como?

Com transporte connectionless (UDP)

Request protocol (R): semântica?

Implementa a semântica *maybe*

Request / Reply protocol (RR): semântica?

Implementa facilmente semântica *pelo menos uma vez*. Como?

Request / Reply / Acknowledge protocol (RRA): semântica?

usado para melhorar RR no caso de se estar a implementar a semântica no máximo uma vez. Porquê?

OPERAÇÕES IDEMPOTENTES

Em geral, a execução repetida devido às re-emissões conduz a um estado errado do servidor. Nestes casos a semântica “pelo menos uma vez” é inadequada.

Uma operação é **idempotente** se a sua execução repetida não altera o efeito produzido (deixando o servidor no mesmo estado ou num estado aplicacionalmente aceitável como equivalente e produzindo o mesmo resultado).

Exemplos de operações idempotentes:

- em geral todas as operações que não mudam o estado
- reescrever os primeiros 512 bytes de um ficheiro se se ignorar o problema da concorrência de acessos ao ficheiro

Exemplos de operações não idempotentes:

- acrescentar 512 bytes a um ficheiro
- transferir dinheiro entre contas

As operações idempotentes podem ser usadas com um protocolo de invocação remota que faça re-emissões sem filtrar duplicados.

OPERAÇÕES IDEMPO

Nota: Podem existir operações que deixem o servidor no mesmo estado e não sejam idempotentes.

Exemplo ???

Exemplo: uma operação de criação de um ficheiro que devolva como resultado um booleano que indique se o ficheiro foi criado ou não (caso já existisse).

Em geral, a execução repetida de uma operação não altera o estado do servidor. Nestes casos a operação é idempotente.

Uma operação é **idempotente** se a sua execução repetida não altera o efeito produzido (deixando o servidor no mesmo estado ou num estado aplicacionalmente aceitável como equivalente e produzindo o mesmo resultado).

Exemplos de operações idempotentes:

- em geral todas as operações que não mudam o estado

- reescrever os primeiros 512 bytes de um ficheiro se se ignorar o problema da concorrência de acessos ao ficheiro

Exemplos de operações não idempotentes:

- acrescentar 512 bytes a um ficheiro

- transferir dinheiro entre contas

As operações idempotentes podem ser usadas com um protocolo de invocação remota que faça re-emissões sem filtrar duplicados.

SOLUÇÕES GERALMENTE ADOPTADAS

Java RMI

"At most once" sobre TCP

Corba

"At most once" geralmente sobre TCP, "maybe"

.NET Remoting

"At most once" sobre TCP, HTTP, pipes

Web Services

"At most once" sobre HTTP (SMTP, etc.)

WS-Reliability: suporte para "at least once", "exactly-once"

Sistemas *antigos*

DCE RPC permite utilizar as semânticas "at-most once" e "at-least once" a partir de directivas introduzidas na definição da interface. Estão disponíveis implementações sobre UDP e sobre TCP.

SUN/RPC permite utilizar as semânticas "maybe", "at-most once" e "at least once" quer sobre UDP quer sobre TCP. O programador pode também manipular os valores dos timeouts e o número de repetições.

AGENDA

Invocação remota de procedimentos/objectos

Motivação

Modelo

Definição de interfaces e método de passagem de parâmetros

Codificação dos dados

Mecanismos de ligação (binding)

Semântica na presença de falhas

Organização interna do servidor

Sistemas de objetos distribuídos

ORGANIZAÇÃO DOS SERVIDORES

Activação dos servidores

- Servidor a executar continuamente

- Servidor activado quando necessário

Organização interna

- Iterativo vs. concorrente

ACTIVAÇÃO DE OBJECTOS REMOTOS

Motivação: num sistema pode haver um número muito elevado de objectos remotos cujo estado se quer que persista durante tempo ilimitado, mas que não estão em uso durante grande parte do tempo

Solução: activam-se os objectos remotos apenas quando necessário

Quando um método é invocado ou quando uma referência remota é obtida

Activator: servidor responsável por:

- Manter informação sobre os objectos activáveis

- Activar os objectos remotos quando solicitado por um cliente

- Manter informação sobre localização dos objectos activados

Objecto remoto *passivo* (quando não activado)

- Código

- Estado do objecto *marshalled*

Referência remota mantém informação necessária para solicitar a activação do objecto

ORGANIZAÇÃO DOS SERVIDORES: ACTIVAÇÃO

Quando é necessário activar um servidor (objecto remoto),
pode-se criar:

- Um servidor para atender todos os clientes

 - Aproximação mais comum

- Um servidor para atender cada cliente

 - E.g. .NET remoting: servidor *SingleCall*

ORGANIZAÇÃO DOS SERVIDORES: *THREADS*

Servidor iterativo: o servidor executa os pedidos de forma sequencial, executando um de cada vez

Modelo simples e claro

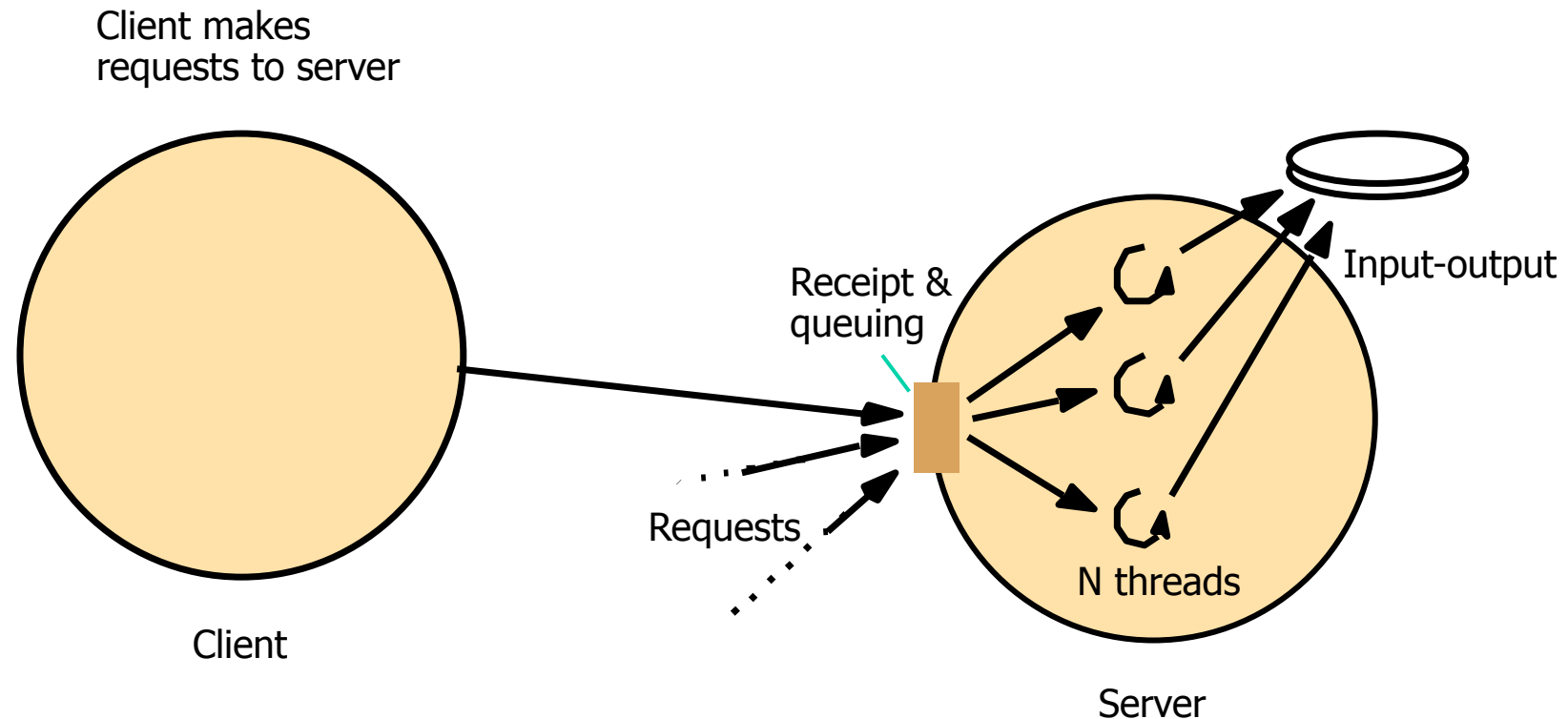
Para alguns tipos de serviços, esta aproximação pode ter um desempenho inadequado

Exemplos: servidores de bases de dados, de ficheiros, etc. Porquê?

Exemplo: serviços que chamam outros serviços em ambos os sentidos ($A \rightarrow B$ e $B \rightarrow A$). Porquê?

Em geral, quando a execução de uma operação remota pode ser longa é interessante introduzir concorrência no servidor.
Porquê?

UTILIZAÇÃO DE THREADS NUM SERVIDOR

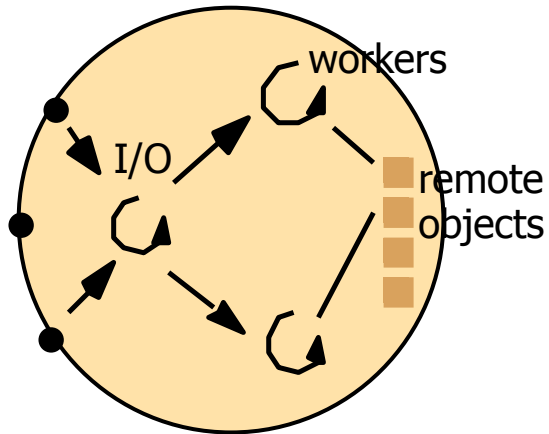


A ter em atenção:

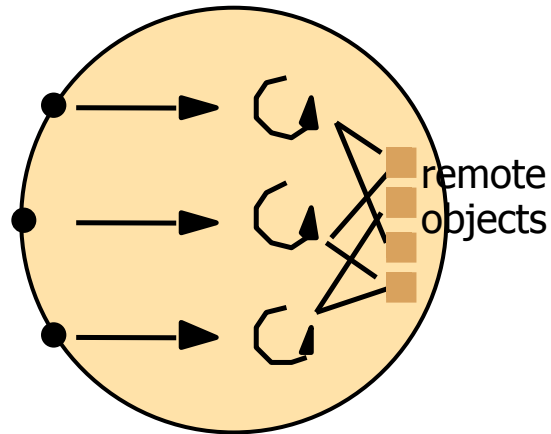
Possíveis problemas de concorrência: necessidade de sincronizar execução dos vários *threads*.

Como é que os threads se organizam e se relacionam com os pedidos?

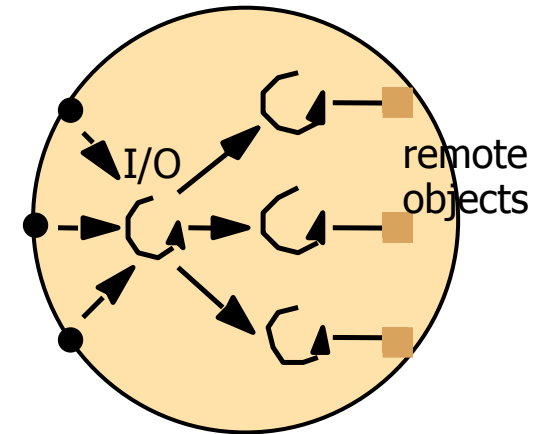
HIPÓTESES DE RELAÇÃO INVOCACÕES / *THREADS*



a. Thread-per-request



b. Thread-per-connection



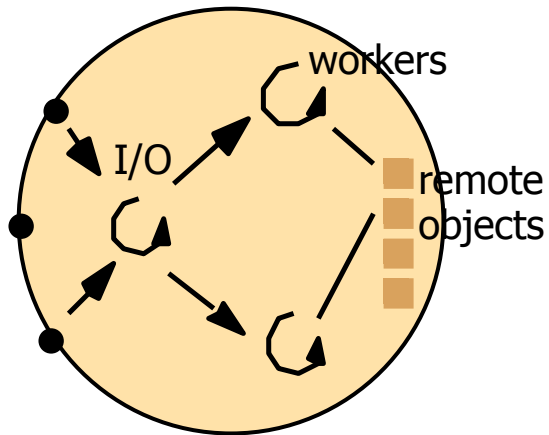
c. Thread-per-object

Thread-per-request: cada invocação é executada por um *thread*

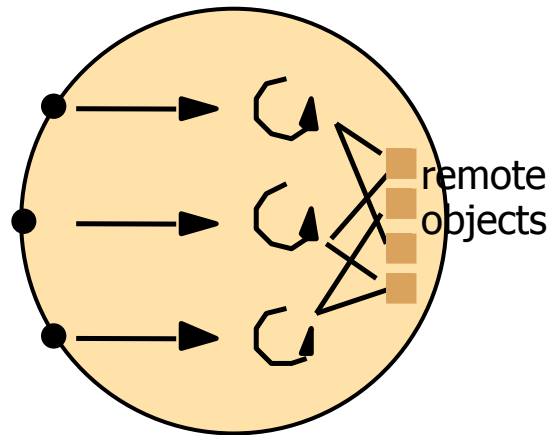
Thread-per-connection: as invocações de uma conexão são executadas todas pelo mesmo *thread*

Thread-per-object: as invocações para um objecto são executadas todas pelo mesmo *thread*

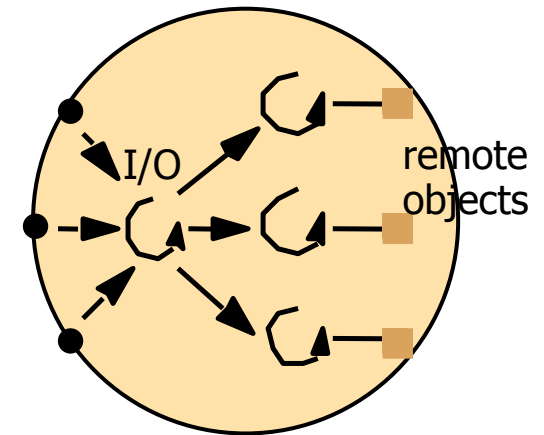
HIPÓTESES DE RELAÇÃO INVOCACÕES / *THREADS*



a. Thread-per-request



b. Thread-per-connection



c. Thread-per-object

Thread-per-request: cada inv

Thread-per-connection: as in
mesmo *thread*

Thread-per-object: as invoca
mesmo *thread*

Quais os problemas de concorrência entre os threads nos vários modelos?

Nos modelos *thread-per-request* e *thread-per-connection* podem existir vários *threads* a aceder aos mesmos dados.

No modelo *thread-per-object* pode existir um problema de deadlock distribuído se a execução de um método puder levar à invocação local de um método noutra objecto

ORGANIZAÇÃO DOS *THREADS*

A aproximação normal é uma solução *thread-per-request*.

Nos sistemas que usam múltiplos *threads* é comum:

Existir um *thread* responsável por distribuir as invocações e existir um conjunto de *threads* responsáveis por executar as invocações, sendo reutilizados em sucessivas invocações

Pools de threads

Em cada momento, o sistema mantém informação sobre os *threads* que não estão a processar nenhuma operação, os quais se encontram a *dormir*

Quando uma nova invocação é recebida, a informação sobre a mesma é passada para um *thread* da *pool*, o qual fica responsável por processar o pedido

No fim de processar o pedido, o *thread* volta à *pool*

AGENDA

Invocação remota de procedimentos/objectos

Motivação

Modelo

Definição de interfaces e método de passagem de parâmetros

Codificação dos dados

Mecanismos de ligação (binding)

Semântica na presença de falhas

Organização interna do servidor

Sistemas de objetos distribuídos

SISTEMAS DE OBJETOS DISTRIBUÍDOS

Extensão do modelo de uma aplicação composta por múltiplos objetos para um ambiente distribuído, em que os objetos executam em diferentes máquinas

Problemas adicionais:

- Garbage collection

- Carregamento dinâmico de código

GARBAGE-COLLECTION: PROBLEMA

Numa aplicação distribuída composta por objetos a executar em diferentes máquina, como saber que um objeto já não é necessário?

Usar abordagem normal de garbage collection: se não existir referência para o objeto, este pode ser removido.

GARBAGE-COLLECTION: JAVA RMI

Entre máquinas virtuais, o sistema Java executa um algoritmo de *garbage-collection* distribuído para remover (apagar) objectos remotos para os quais não existem referências

Cada máquina virtual (VM) contabiliza as referências que existem para cada objecto remoto e informa a VM do objecto

Quando aparece a primeira referência, a VM do objecto remoto é informada

Quando é removida a última referência, a VM do objecto remoto é informada

Um objecto remoto pode ser removido (apagado) quando não existem referências em nenhuma VM

QUESTÃO DUM EXAME

Como sabe, o sistema Java RMI usa um mecanismo de garbage collection distribuído para recolher (remover) os objectos remotos que já não são necessários. Explique o que acontece a um objecto remoto no caso de existir uma falha temporária da rede que impeça a comunicação entre a máquina virtual em que executa um objecto remoto e a máquina virtual do único programa que mantém uma referência para esse objecto remoto.

CARREGAMENTO DE CÓDIGO: PROBLEMA

Numa aplicação com objetos, dado um método

```
void function( T t)
```

é possível invocar o método com qualquer objeto cujo tipo estenda T.

Num sistema distribuído, se considerarmos um método disponível remotamente, seria possível passar um objeto dum tipo cujo código não seria conhecido no servidor.

Solução: impedir esta situação ou carregar o código dinamicamente.

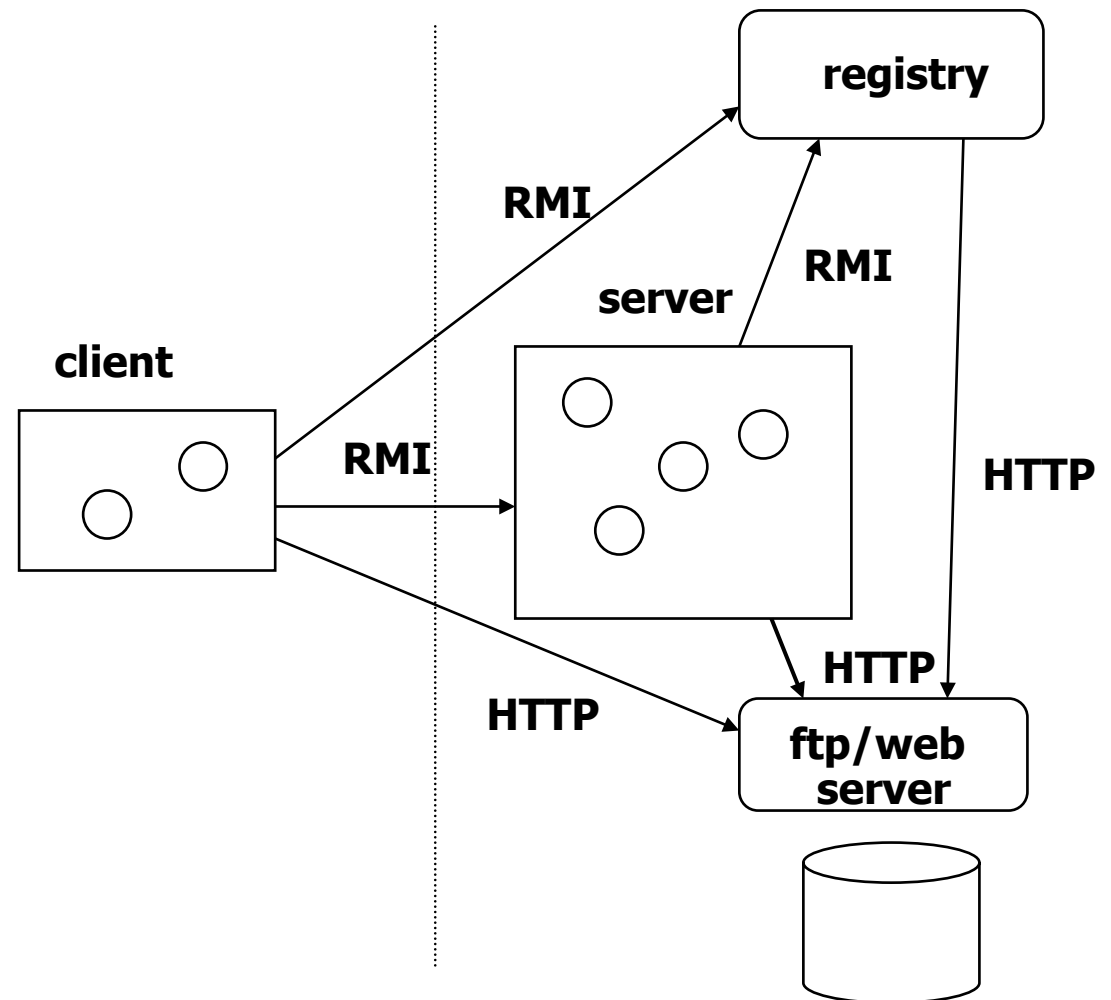
CARREGAMENTO DE CLASSES: SOLUÇÃO JAVA

RMI carrega o código das classes se o mesmo não estiver disponível

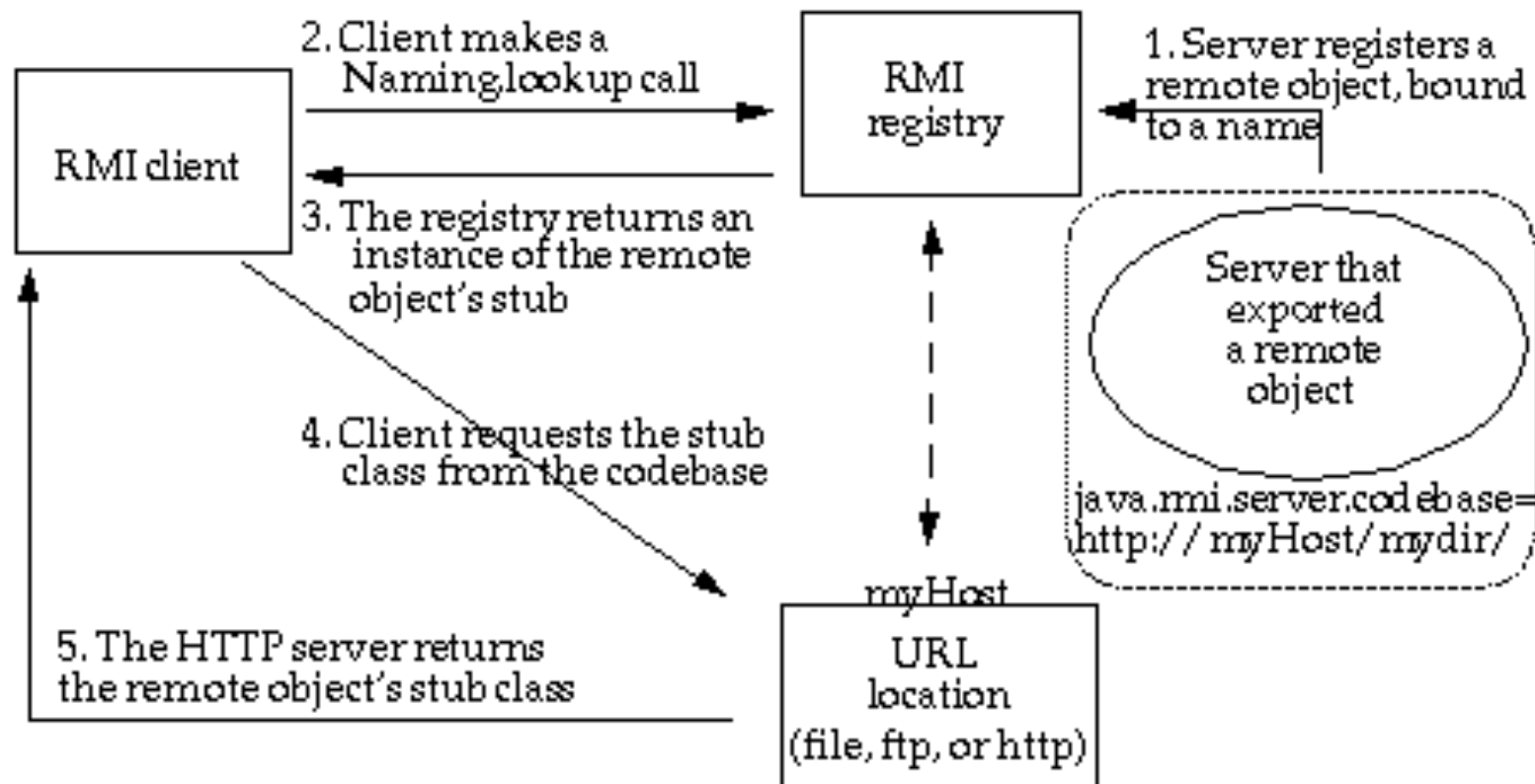
Isto aplica-se às classes do objecto remoto, do *stub*, do esqueleto, dos parâmetros e do valor de retorno dos métodos

O carregamento faz-se remotamente se necessário

Os URLs das classes ficam anotados nas referências remotas para se poderem localizar as mesmas

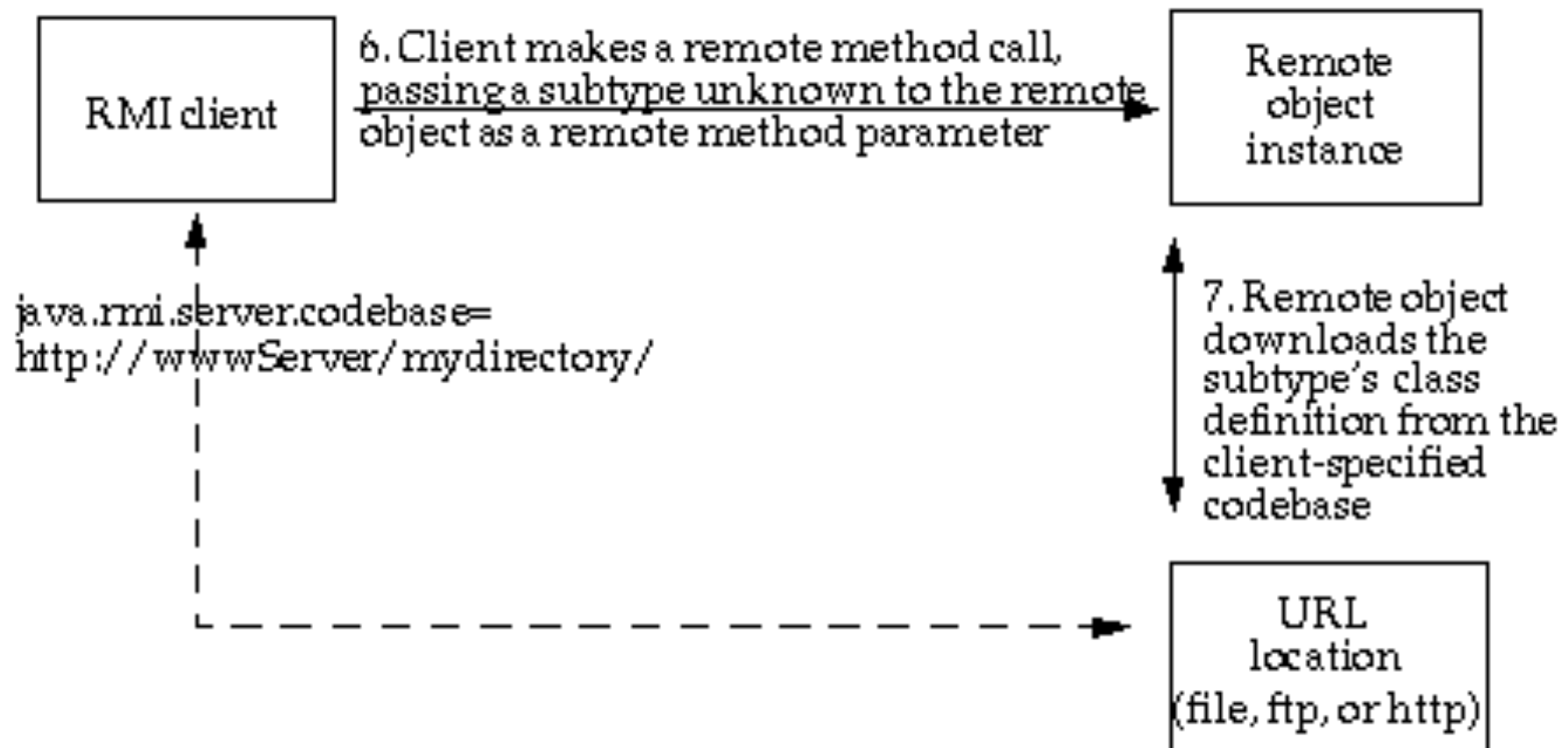


CARREGAMENTO DE CÓDIGO NO CLIENTE



Os objectos anotam o seu *codebase*, o que permite obter o código das classes remotamente sem configuração adicional.

CARREGAMENTO DE CÓDIGO NO SERVIDOR



SEGURANÇA DO CARREGAMENTO DO CÓDIGO

Carregar código de fontes desconhecidas é um potencial problema de segurança

O RMI usa os mecanismos de segurança da linguagem Java

Qualquer código a carregar tem de o ser através de um carregador de classes (*class loader*)

Um gestor de segurança (*security manager*) tem de estar presente para que o carregamento de código seja possível

As aplicações podem fornecer o seu próprio gestor de segurança

PARA SABER MAIS

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 5th Edition, 2009

RMI/RPCs - capítulo 5.

Representação de dados e protocolos - capítulo 4.3.

Web services – capítulo 9

Corba – capítulo 8.3.