

SISTEMAS DISTRIBUÍDOS

Capítulo 4

Invocação remota

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

MOTIVAÇÃO

Estruturar uma aplicação distribuída com base nas mensagens trocadas pelos seus componentes é a abordagem mais óbvia, mas:

- exige atenção a muitos detalhes de baixo nível; a estrutura dos programas espelha os padrões de comunicação, em vez da lógica da aplicação no seu todo.
- em particular, os servidores ficam estruturados em função das mensagens que sabem tratar.

PROBLEMAS ?

De aplicação para aplicação, verifica-se que muitas linhas de código são *repetitivas*, não contêm nenhum significado aplicacional específico e referem-se ao processamento das comunicações.

Em particular, boa parte do código está dedicado a:

- 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

OBJETIVO

- Não será possível automatizar aquilo que é repetitivo?
- Não será possível que o programador apenas especifique o código aplicativo?

INVOCAÇÃO REMOTA

Nas linguagem imperativas definem-se funções / procedimentos / métodos para executar uma dada operação. Num ambiente distribuído, uma extensão natural consiste em permitir que a **execução ocorra noutra máquina**.

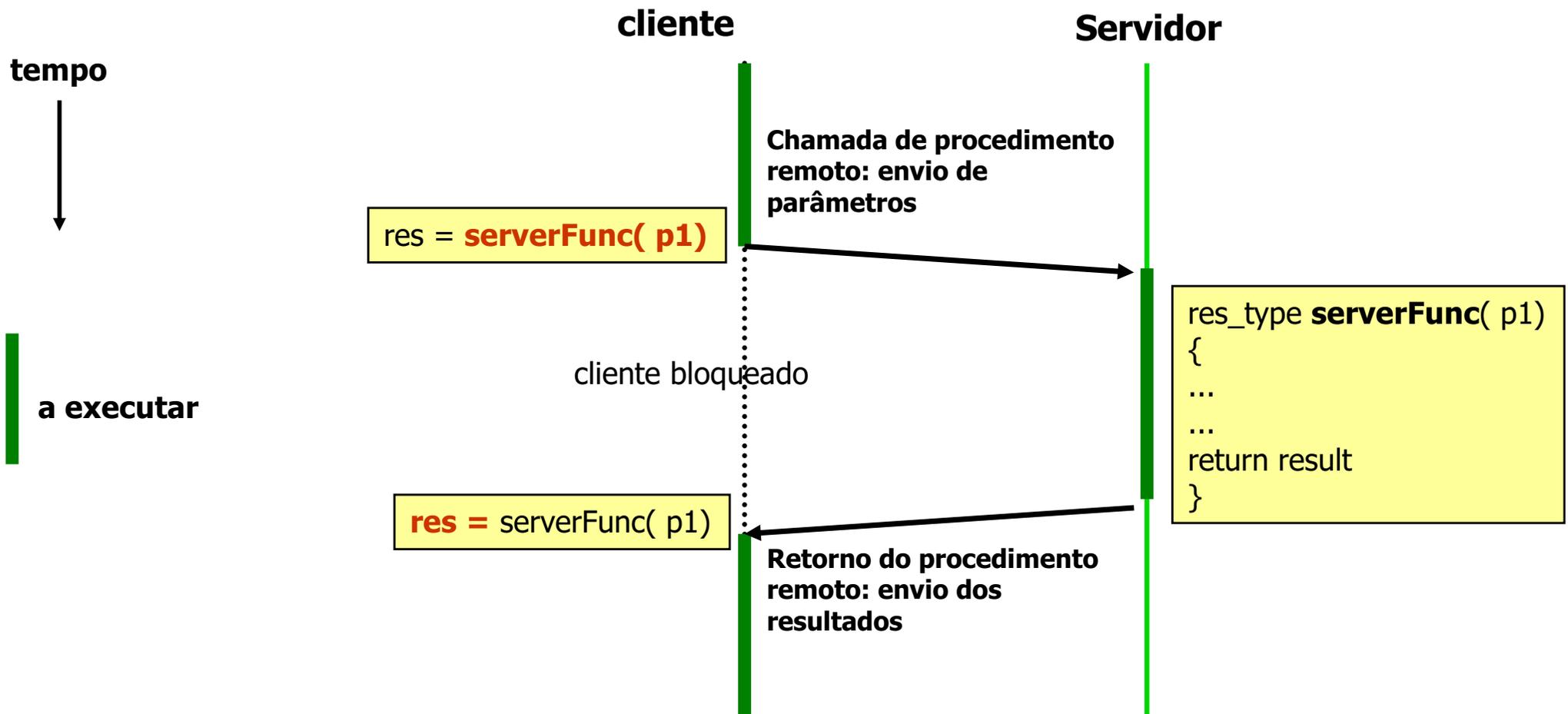
Invocação Remota de Procedimentos (RPCs), quando são executados funções/procedimentos remotamente.

- gRPC (<https://grpc.io/>), ONC/RPC, DCE,

Invocação Remota de Métodos (RMI), quando são executados métodos de objetos remotos.

- JAVA RMI, .NET Remoting, Corba.
- WebServices (Rest e SOAP)

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 REMOTA - PROPRIEDADES

Extensão natural do paradigma imperativo/procedimental a um ambiente distribuído

- Modelo síncrono de comunicação suporta chamadas bloqueantes no cliente

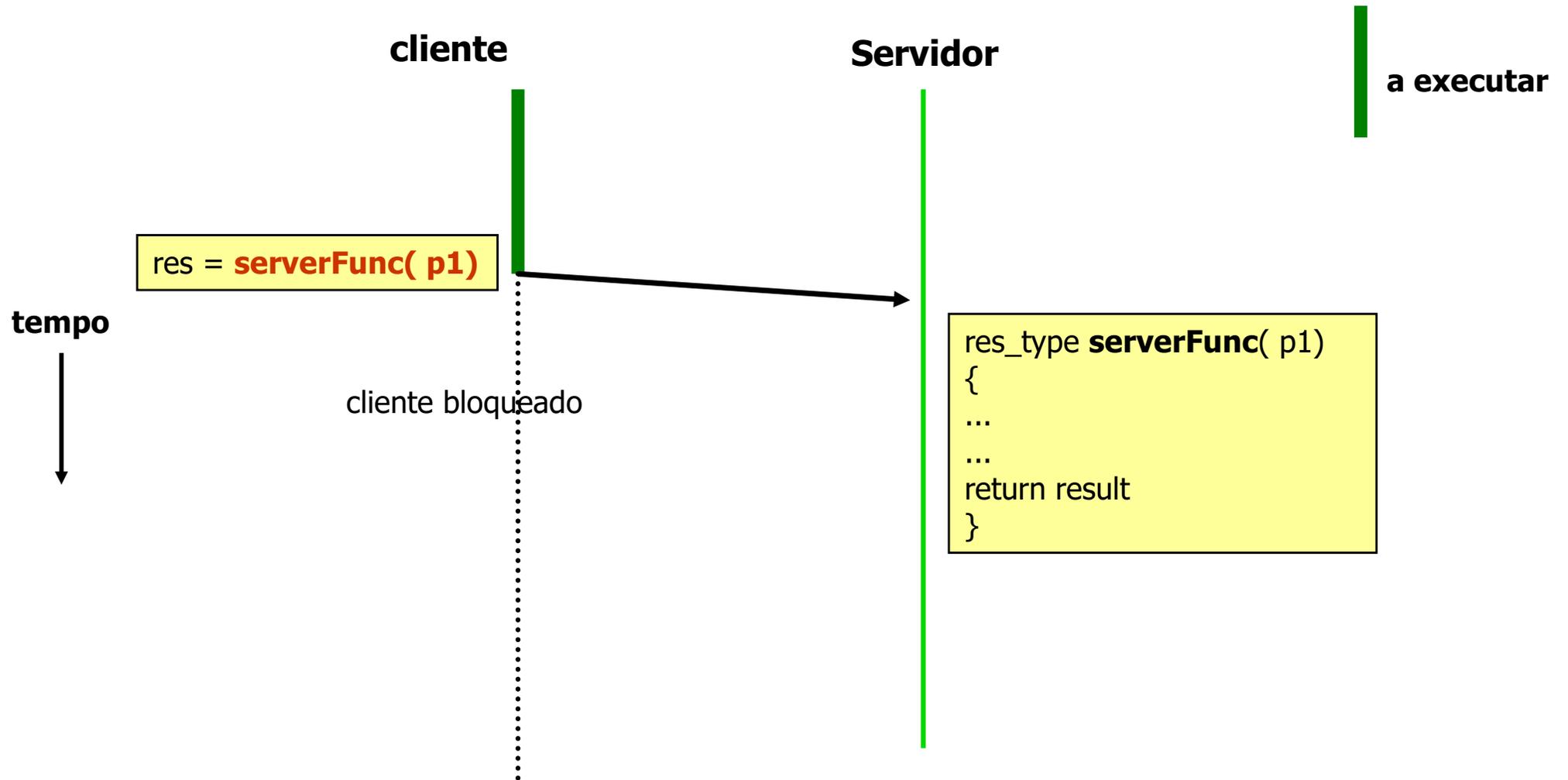
Esconde detalhes de comunicação (e tarefas repetitivas)

- 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)

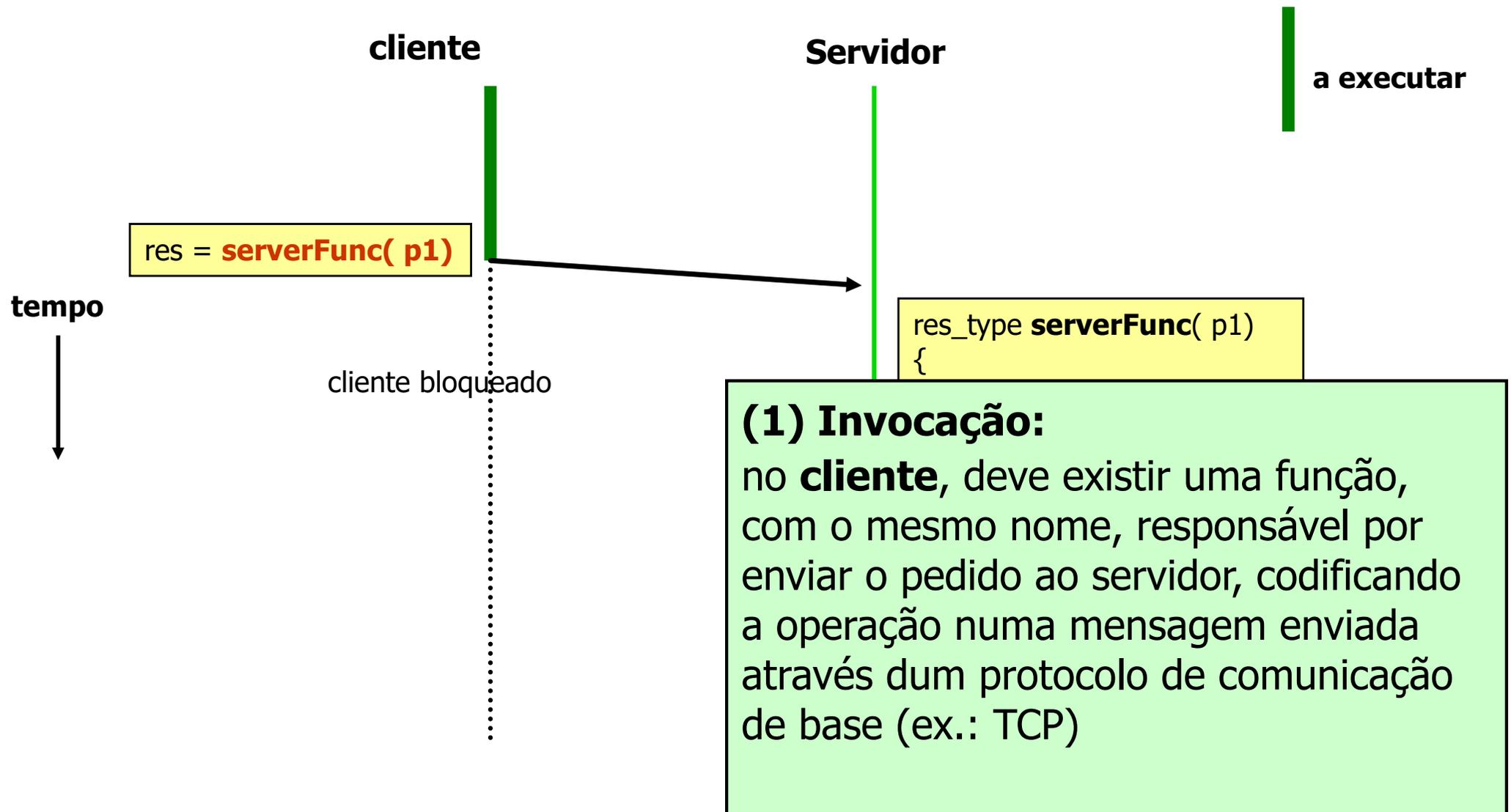
Cliente

```
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)
```

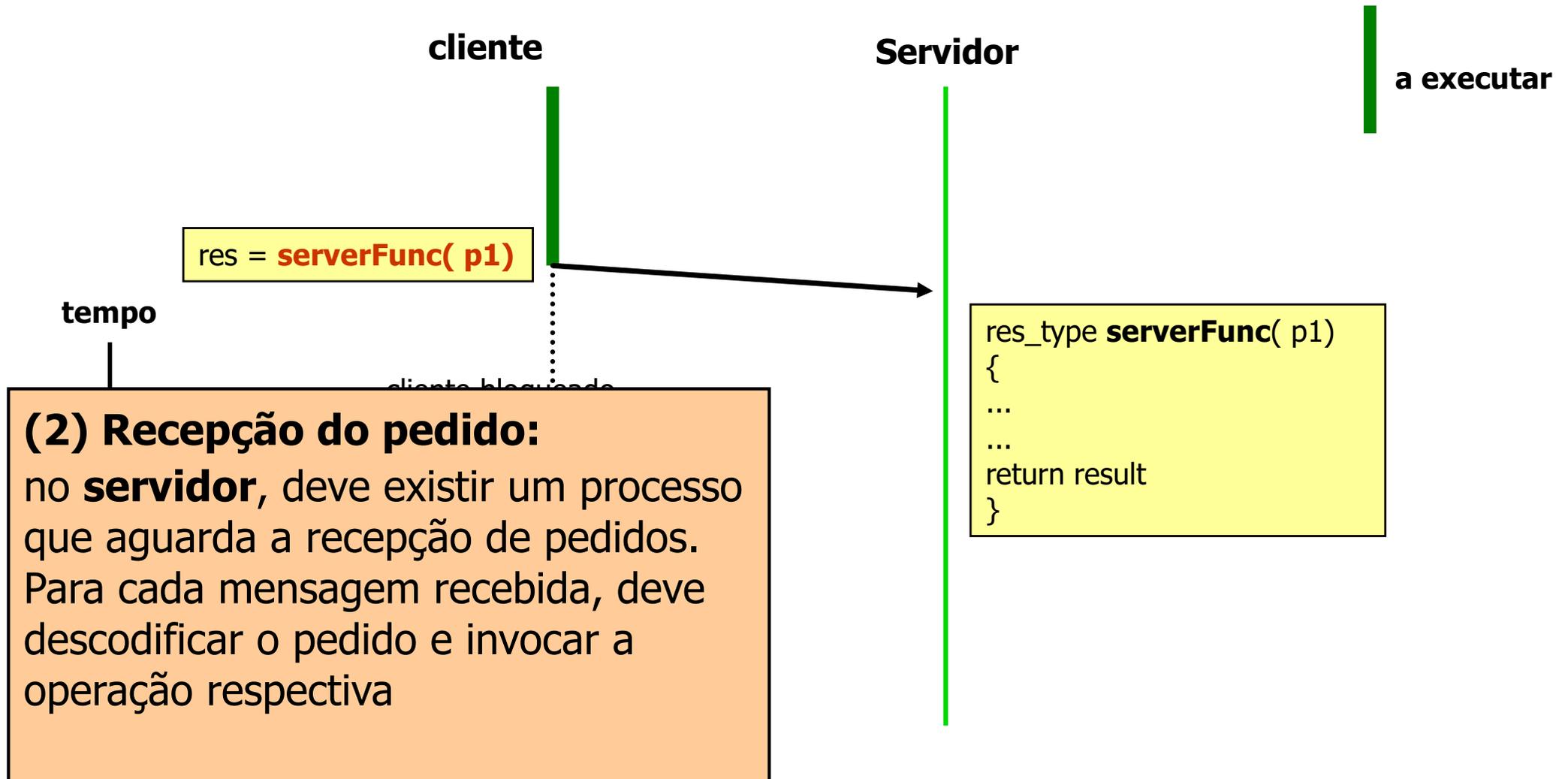
```
res_type serverFunc( T1 p1)  
  s = new Socket( host, port)  
  s.send( msg( "serverFunc",[p1]))
```

(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 ESCONDE 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

GUAGEM)

Servidor

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

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

```
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

Servidor

a executar

tempo

cliente bloqueado

```
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

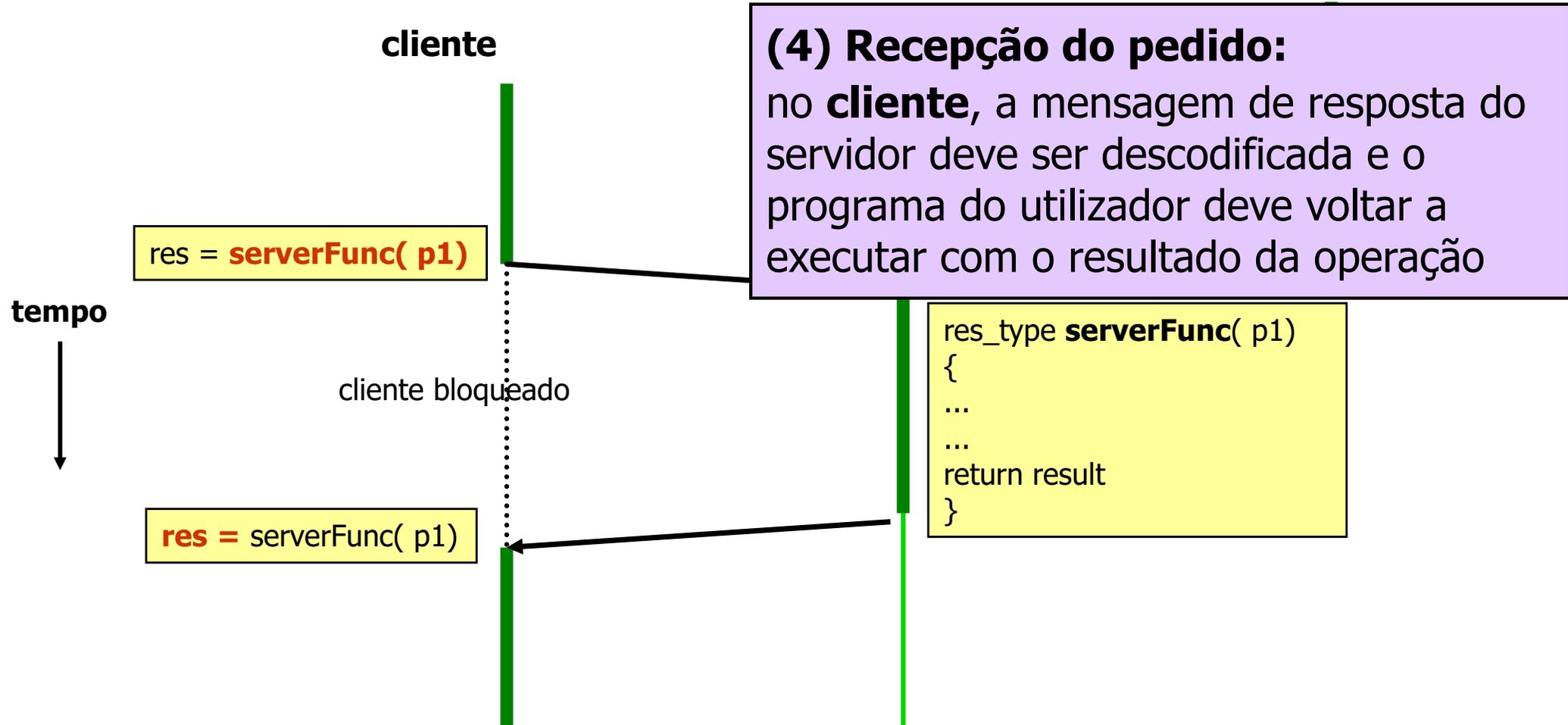
```
res_type serverFunc( T1 p1)
  s = new Socket( host, port)
  s.send( msg( "serverFunc",[p1]))
```

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
```

RPCs – AUTOMATIZAÇÃO (PROXY/STUB COMPILERS)

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

É costume designar-se de **stub do cliente** às funções do cliente que efetuam a comunicação com o servidor para executar o método no servidor;

Do lado do servidor, o **stub ou *skeleton* do servidor** corresponde ao código de comunicação para esperar as invocações e executá-las, devolvendo o resultado;

RPCs – AUTOMATIZAÇÃO (PROXY/STUB COMPILERS)

Em alguns sistemas e ambientes usam-se ferramentas (compiladores) para gerar os stubs; e.g., wsimport para WebServices SOAP;

Noutros sistemas a geração é automática: no servidor, quando este é instanciado; no cliente quando este se liga ao servidor da primeira vez;

- e.g., Java RMI, Servidor JAX-RS(Jersey);

Há ainda casos onde a invocação remota faz parte da própria especificação do ambiente/linguagem e é parte integrante do runtime.

- e.g., .NET Remoting

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
{
    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 ContaBancaria
    {
        double SaldoAtual
        {
            get;
        }
        void depositar ( float q );
        void levantar ( float q );
    }
}
```

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

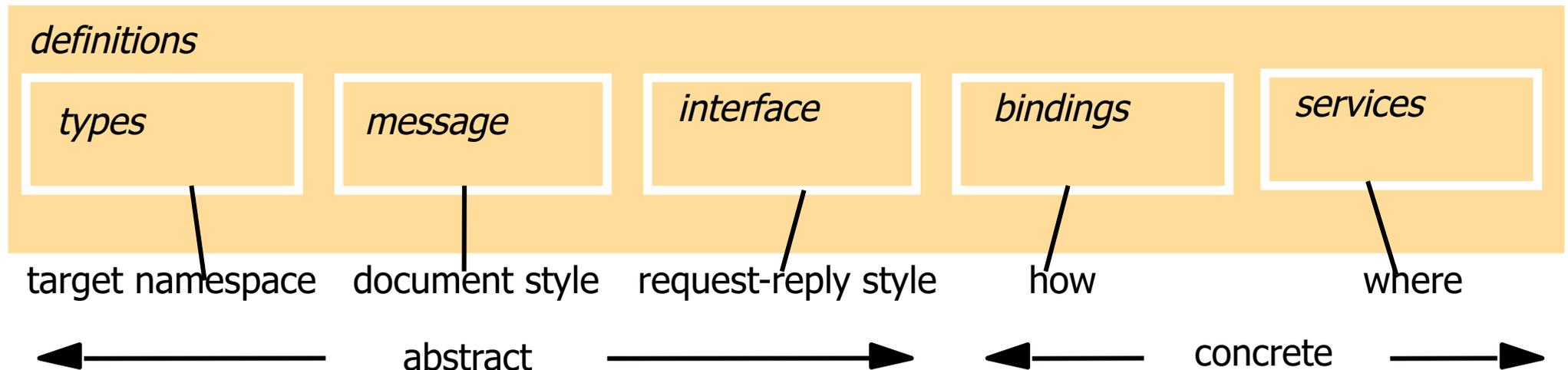
No .NET Remoting não é necessário definir qual a interface remota – esta pode ser inferida a partir da definição do servidor

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



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>
```

(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://sc
      namespace="urn:examples:
      use="encoded"/>
  </output>
</operation>
</binding>

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

<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

```
<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>
```

WSDL A PARTIR DO JAVA (JAX-WS)

@WebService()

public class SimpleWSServer {

...

public SimpleWSServer() {

...

}

@WebMethod()

public String[] list(String path) {

...

}

}

INTERFACE SERVIDOR REST EM JAVA (JAX-RS)

`@Path("/files")`

public interface FileServerREST {

`@GET`

`@Path("/{path}")`

`@Produces(MediaType.APPLICATION_JSON)`

public Response list(`@PathParam("path")` String path);

`@POST`

`@Path("/{path}")`

`@Consumes(MediaType.OCTET_STREAM)`

`@Produces(MediaType.APPLICATION_JSON)`

public Response upload (`@PathParam("path")` String path, byte[] contents);

}

}

SISTEMAS DISTRIBUÍDOS

Capítulo 4 – aula 5

Invocação de procedimentos e de métodos remotos

NA ÚLTIMA AULA

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

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
```

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

NA AULA DE HOJE

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

Várias dimensões do problema

- Diferentes representações de tipos primitivos dependendo do sistema/processador
- Diferentes representações dos tipos complexos em diferentes linguagens

Dados têm se ser enviados como uma sequência/array de bytes

REPRESENTAÇÃO DOS TIPOS PRIMITIVOS

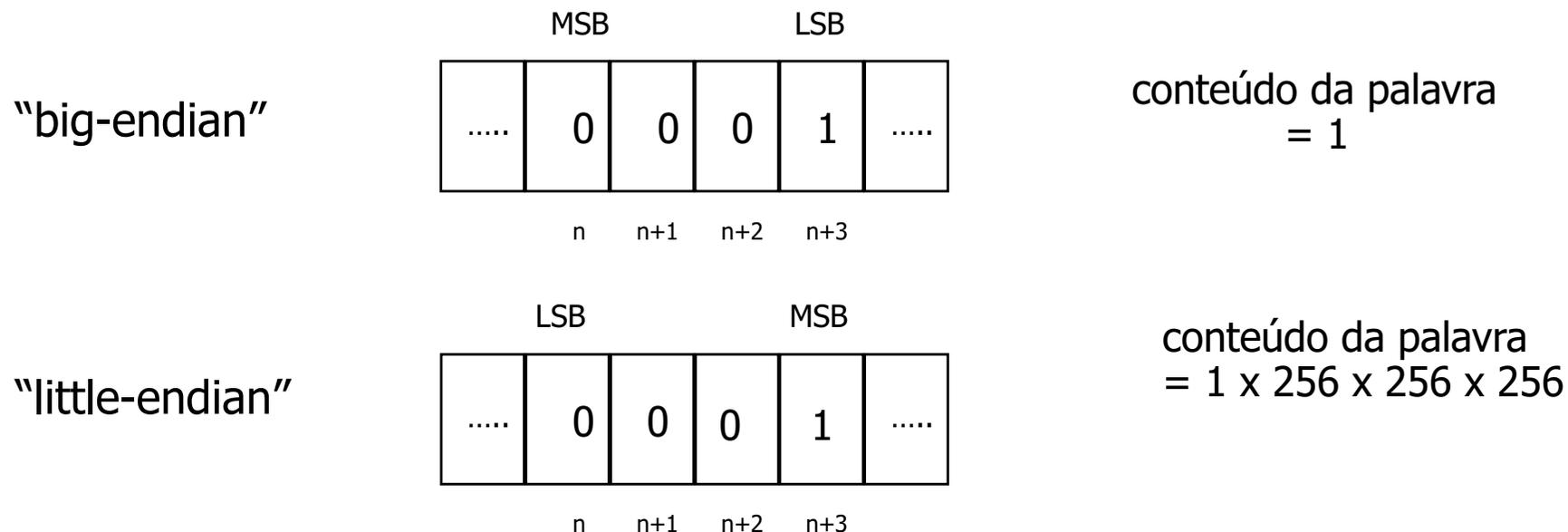
Diferentes sistemas representam os tipos primitivos de formas diferentes

Inteiros armazenados por ordem diferente em memória – big-endian vs. little endian

Diferentes representações para números reais – IEEE 754, decimal32, etc.

Caracteres com diferentes codificações – ASCII, UTF-8 ,UTF-16, etc.

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

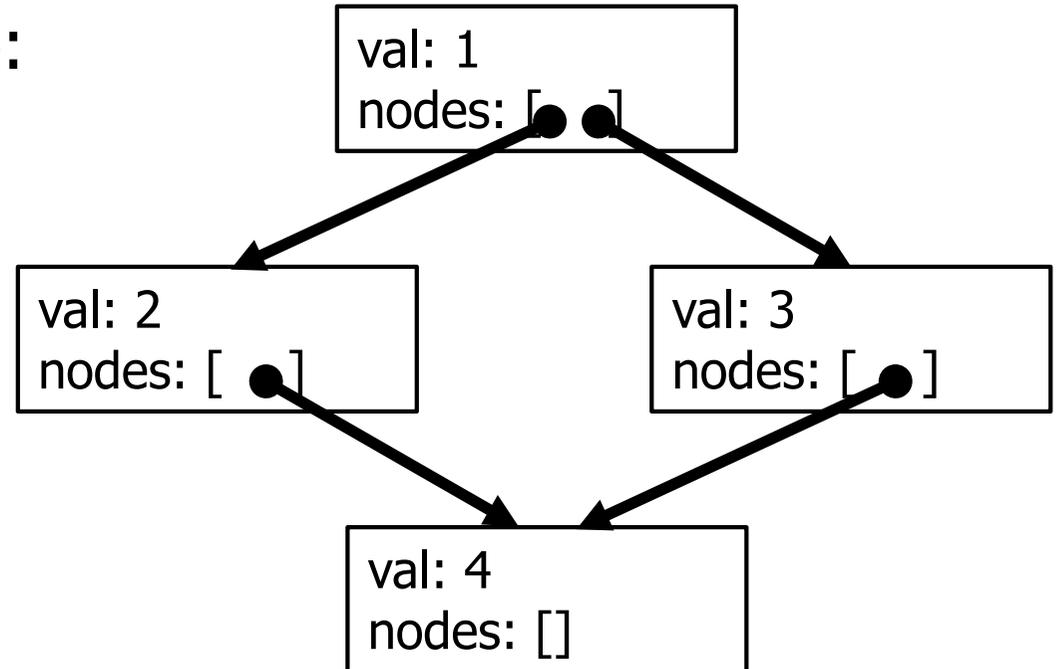
Marshalling – processo de codificar do formato interno para o formato rede

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

PORQUE É QUE O MARSHALLING É COMPLEXO

Considere o seguinte exemplo:

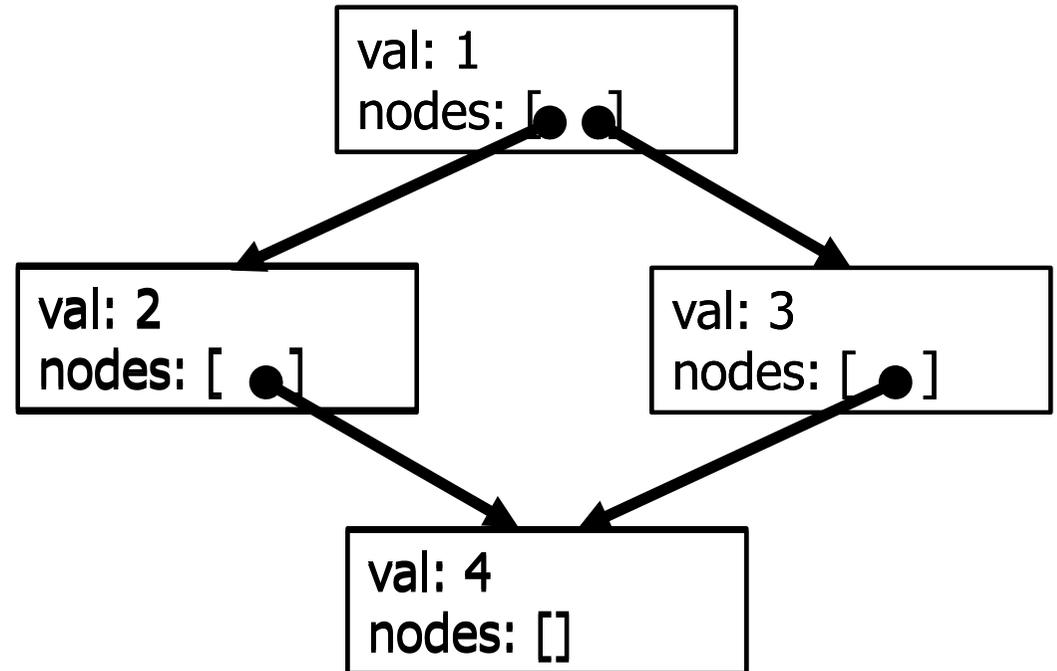
```
class Node {  
    int val;  
    Node[] nodes;  
}
```



PORQUE É QUE O MARSHALLING É COMPLEXO?

Serializamos outra vez o node com val = 4 ?
Necessário ter forma de referenciar que um objeto já foi serializado.

Necessário representar as referências.
O que serializamos a seguir?



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

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>  
  <years>1924</years>
```

```
<?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.

```
<value><i4>41</i4></value>  
</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

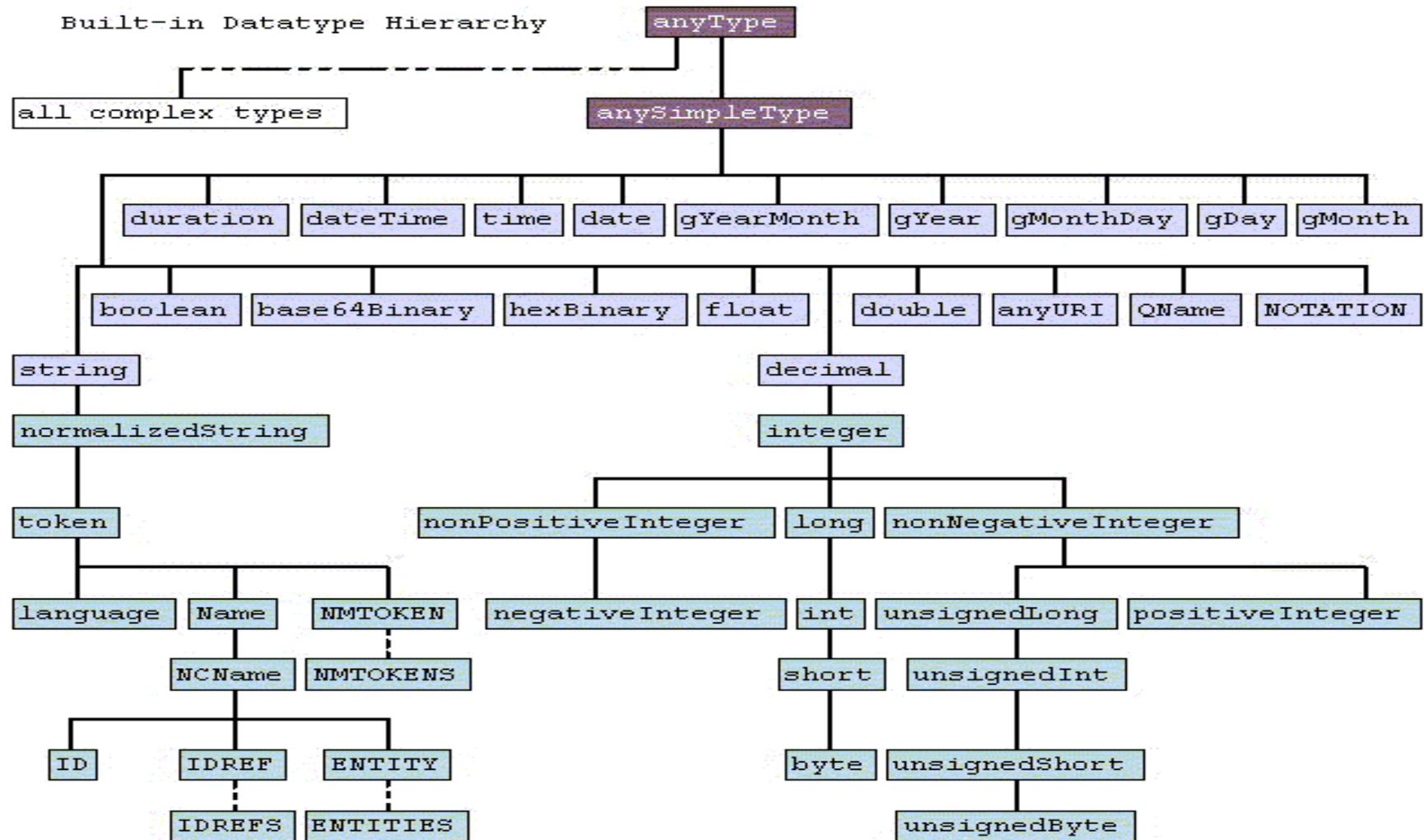
Um XML namespace permite criar espaço de nomes para os nomes dos elementos e atributos usados

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

JSON (JAVASCRIPT OBJECT NOTATION)

JSON permite descrever estruturas de dados complexas em formato de texto

Tipos primitivos

Number

String

Boolean

Tipos complexos

Array

Object (mapa chave / valor)

JSON é uma alternativa ao XML

```
{ "Person": {  
  "name": "Smith",  
  "place": "London",  
  "year": 1934,  
}
```

```
{ "Person": {  
  "name": "Smith",  
  "place": "London",  
  "year": 1934,  
  "phone": [999999999,  
            888888888],  
}
```

PROTOBUF (GOOGLE PROTOCOL BUFFERS)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2
    [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}
```

```
person {
  name: "John Doe"
  id: 13
  email: "jdoe@example.com"
}
```

Dados passam na rede em formato binário

Menor dimensão, mais rápido a processar

E.g. protobuf: 28 bytes; 100 ns
XML: 69 bytes; 5000 ns

PROTOBUF (GOOGLE PROTOCOL BUFFERS)

Dados passam na rede em formato binário

Compilador cria código para serializar/deserializar dados estruturados

Resultado: menor dimensão, mais rápido a processar

E.g. protobuf: 28 bytes; 100-200 ns

XML: 69 bytes; 5000-10000 ns

... E MUITOS MAIS

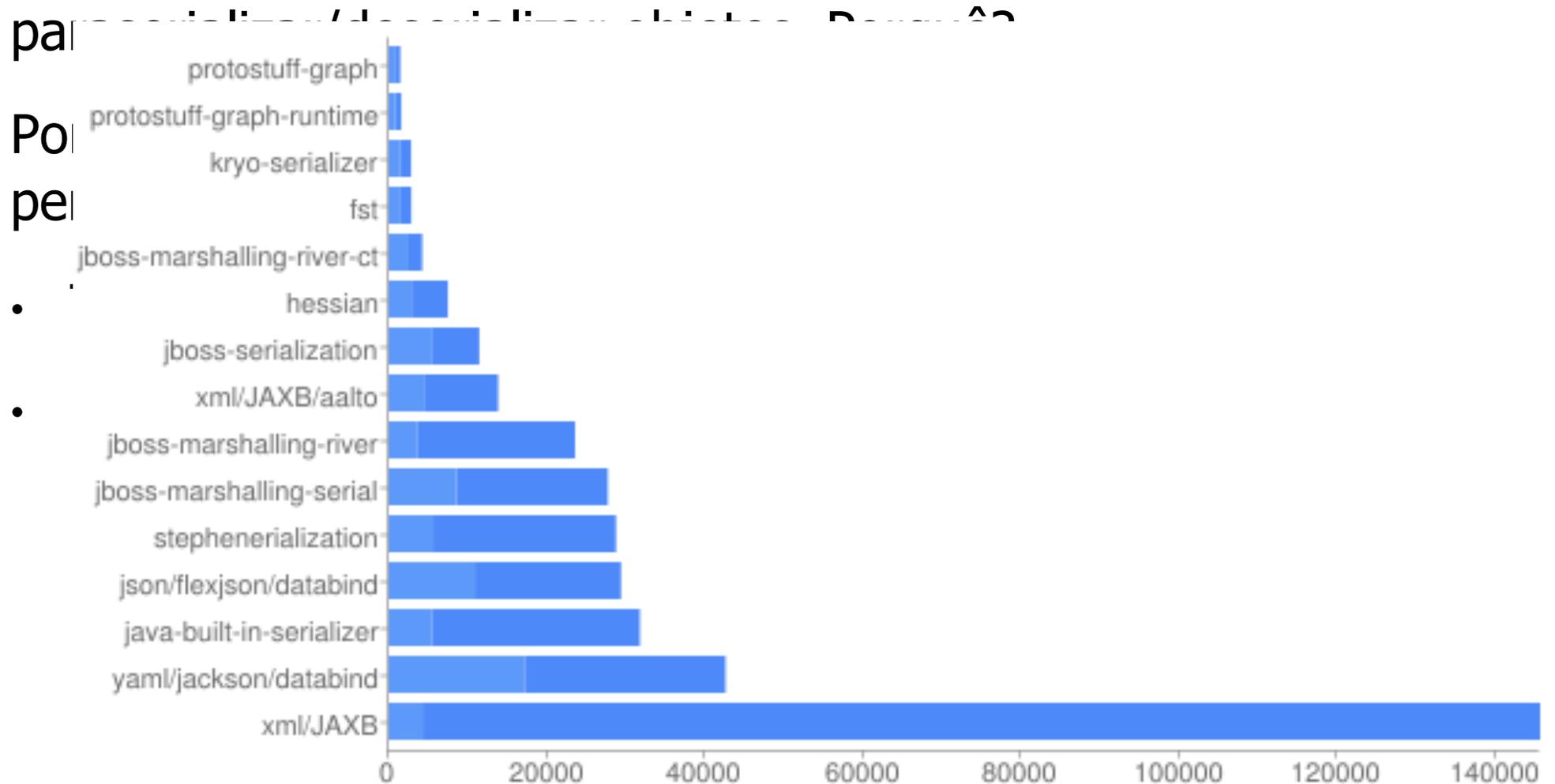
Existe um grande número de soluções para serializar/deserializar objetos. Porquê?

Porque a serialização/deserialização pode ter impacto na performance dos sistemas distribuídos:

- Tempo de serialização/deserialização
- Dimensão das mensagens => tempo de propagação das mensagens

... E MUITOS MAIS

Existe um grande número de soluções



Source: <https://github.com/eishay/jvm-serializers/wiki>

REPRESENTAÇÕES DOS DADOS: CLASSIFICAÇÃO

Conteúdo da representação

- Formato binário – Java, protobuf
- Formato de texto – XML, JSON

Integração com linguagem

- Independente – XML, JSON, protobuf
- Integrado – Java, JSON

Informação de tipos

- Incluída – Java, XML
- Não incluída – JSON, protobuf

MÉTODOS DE PASSAGEM DE PARÂMETROS

Independentemente dos tipos dos parâmetros, os mesmos podem ser:

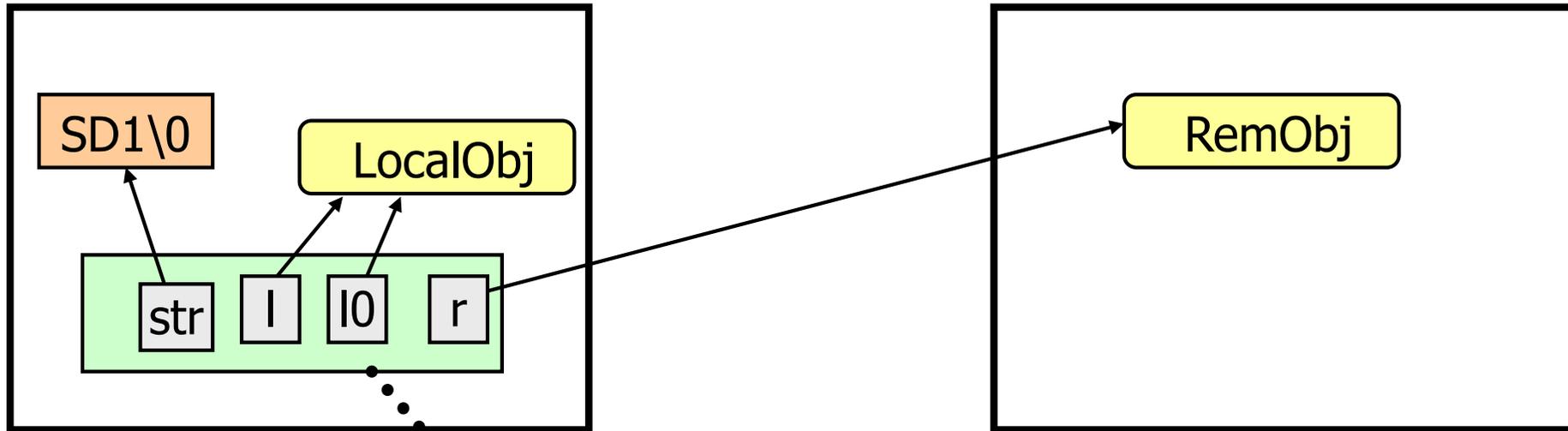
- parâmetros de entrada (in) : cópia no pedido
- parâmetros de saída/resultado (out) : cópia na resposta
- parâmetros de entrada/saída (in/out) : cópia no pedido e na resposta

MÉTODOS DE PASSAGEM DE PARÂMETROS (CONT.)

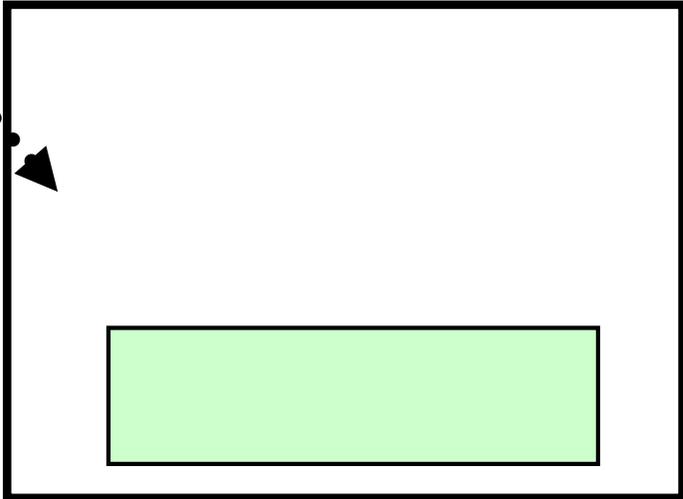
Aproximação comum nos sistemas de RPC/RMI:

- Passagem por valor para tipos primitivos, arrays, estruturas e objetos
 - Apontadores/referências para arrays, objetos, etc. são seguidas
 - Estado dos objectos é copiado (ex: Java RMI)
 - Porque não passar tipos básicos por referência?
- Passagem por referência para objetos remotos
 - quando o tipo de um parâmetro é um objeto remoto, uma referência para o objeto é transferida
 - Porque não passar objetos remotos por valor?

MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO

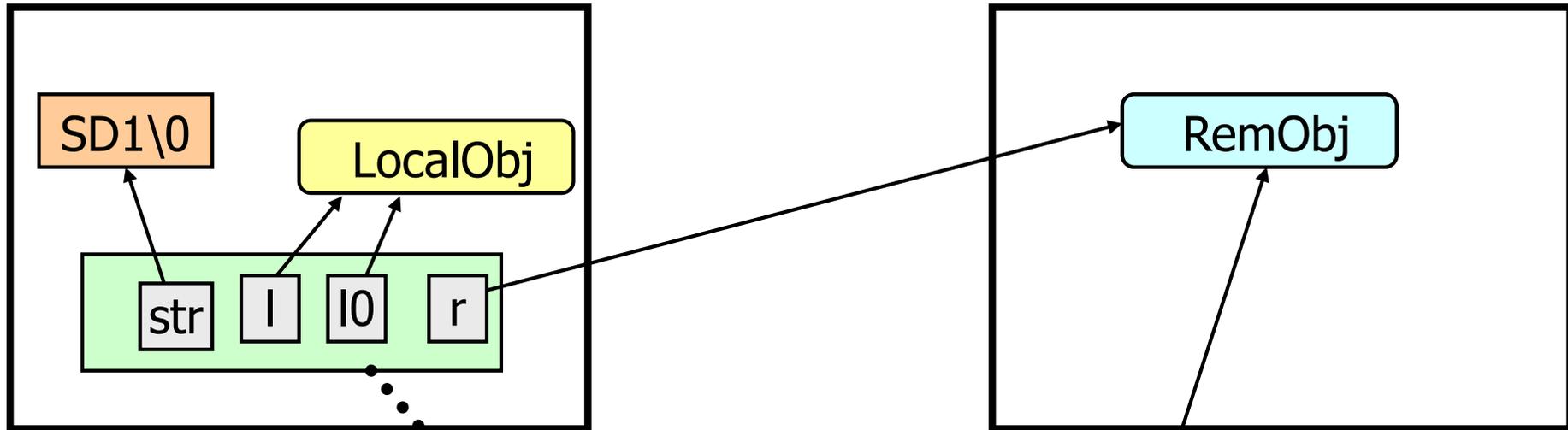


```
LocalObj I = ...  
char[] str = "SD1";  
RemObj r = ...  
  
I = server.f( I, str,r);
```

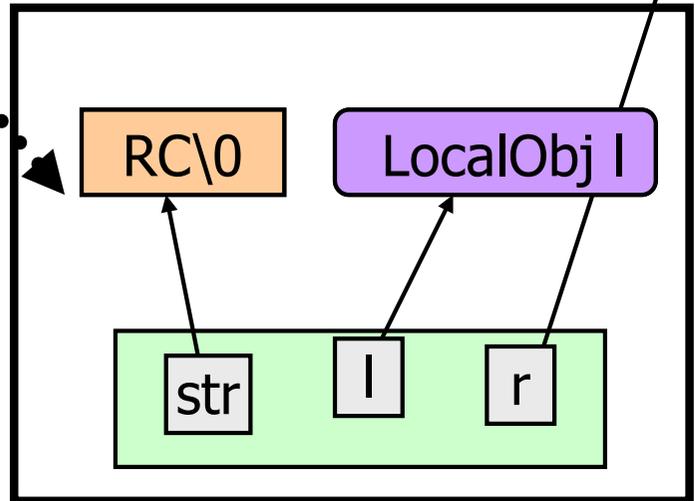


```
Server  
LocalObj f( in LocalObj I,  
in char[] str,in RemObj r){  
...  
}
```

MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



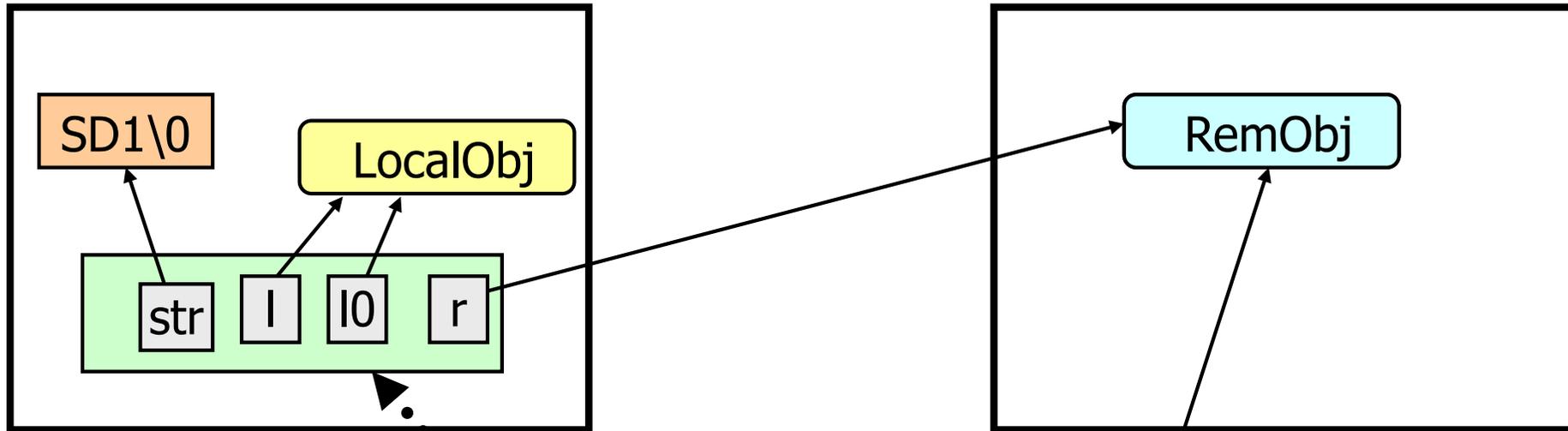
```
LocalObj l = ...  
char[] str = "SD1";  
RemObj r = ...  
  
l = server.f( l, str,r);
```



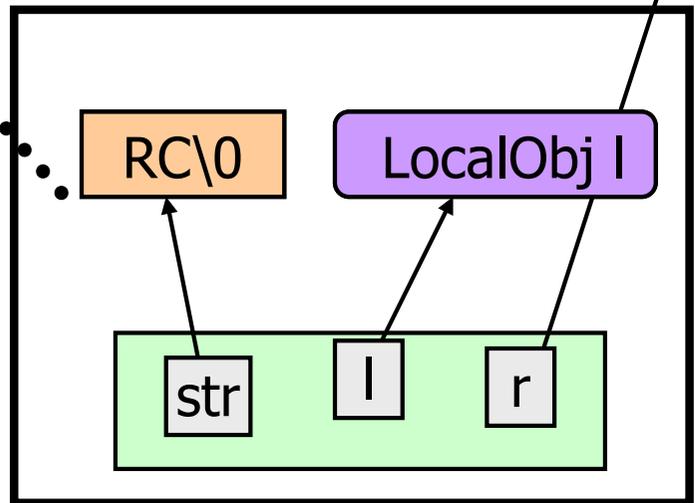
Server

```
LocalObj f( in LocalObj l,  
in char[] str,in RemObj r){  
...  
}
```

MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



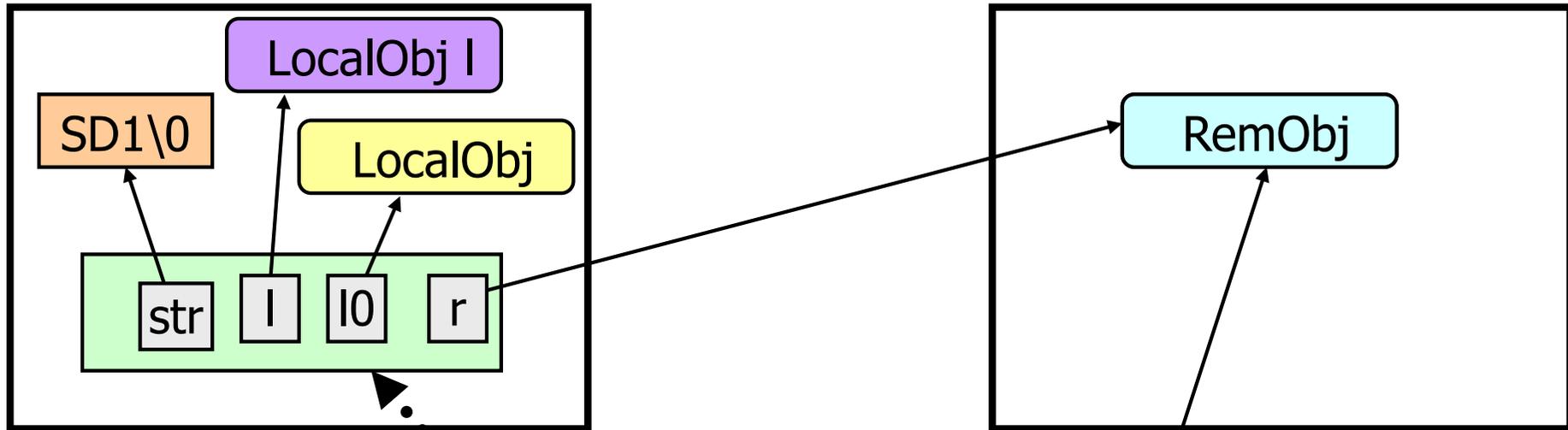
```
LocalObj l = ...  
char[] str = "SD1";  
RemObj r = ...  
  
l = server.f( l, str,r);
```



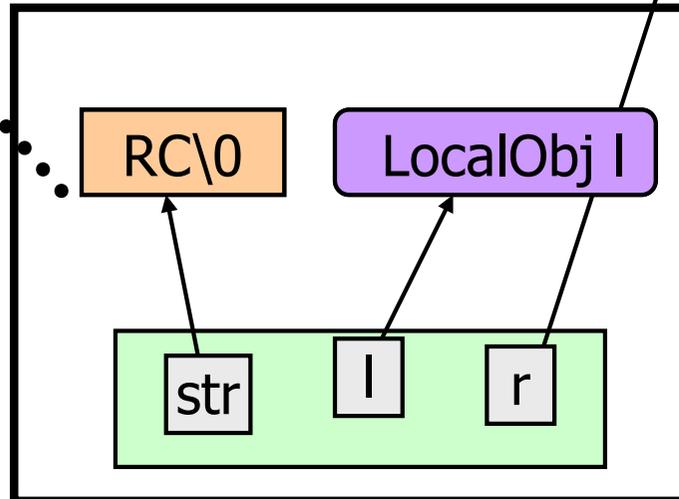
Server

```
LocalObj f( in LocalObj l,  
in char[] str,in RemObj r){  
...  
}
```

MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



```
LocalObj l = ...  
char[] str = "SD1";  
RemObj r = ...  
  
l = server.f( l, str,r);
```



Server

```
LocalObj f( in LocalObj l,  
in char[] str,in RemObj r){  
...  
}
```

PASSAGEM DE OBJECTOS REMOTOS EM PARÂMETRO

Nos sistemas de RMI é, em geral, possível passar (referências para) objetos 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, uma cópia da referência remota é enviada
- Passando como parâmetro/resultado, uma 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 referência remota inclui, pelo menos, a seguinte informação:

- Endereço/porta do servidor
- Tipo do servidor
- Identificador único

DESAFIO

Como medir a latência da disseminação deste vídeo?

SISTEMAS DISTRIBUÍDOS I

Capítulo 4 – aula 6

Invocação de procedimentos e de métodos remotos

NA ÚLTIMA AULA... REPRESENTAÇÕES DOS DADOS

Conteúdo da representação

- Formato binário – Java, protobuf
- Formato de texto – XML, JSON

Integração com linguagem

- Independente – XML, JSON, protobuf
- Integrado – Java, JSON

Informação de tipos

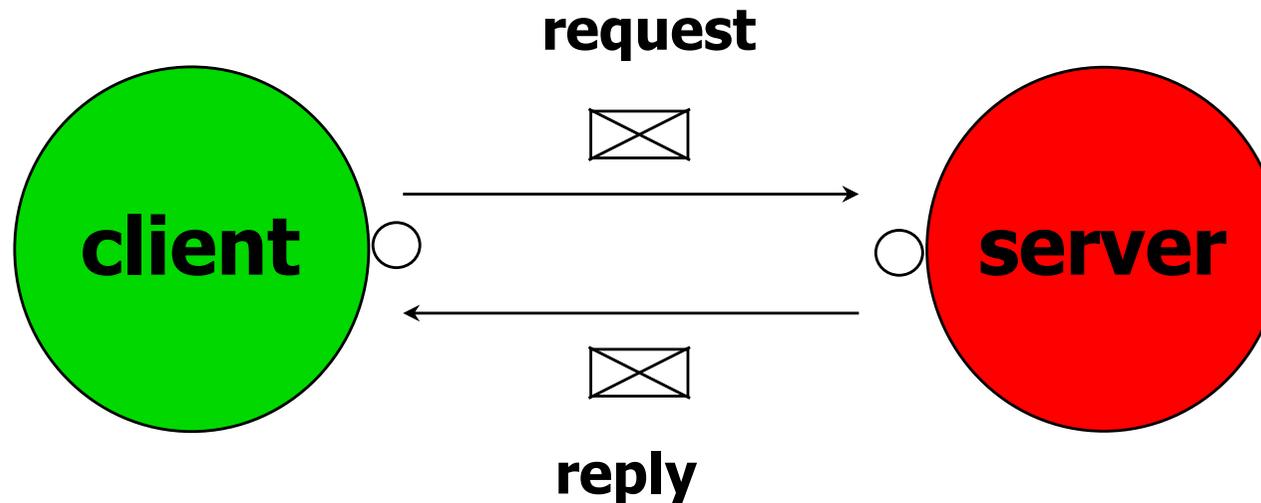
- Incluída – Java, XML
- Não incluída – JSON, protobuf

NA AULA DE HOJE

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, uma referência corresponde ao endereço 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.: REST, Web services, .NET remoting

Servidor de nomes regista associação entre nome e referência remota

- Ex.: Java RMI

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

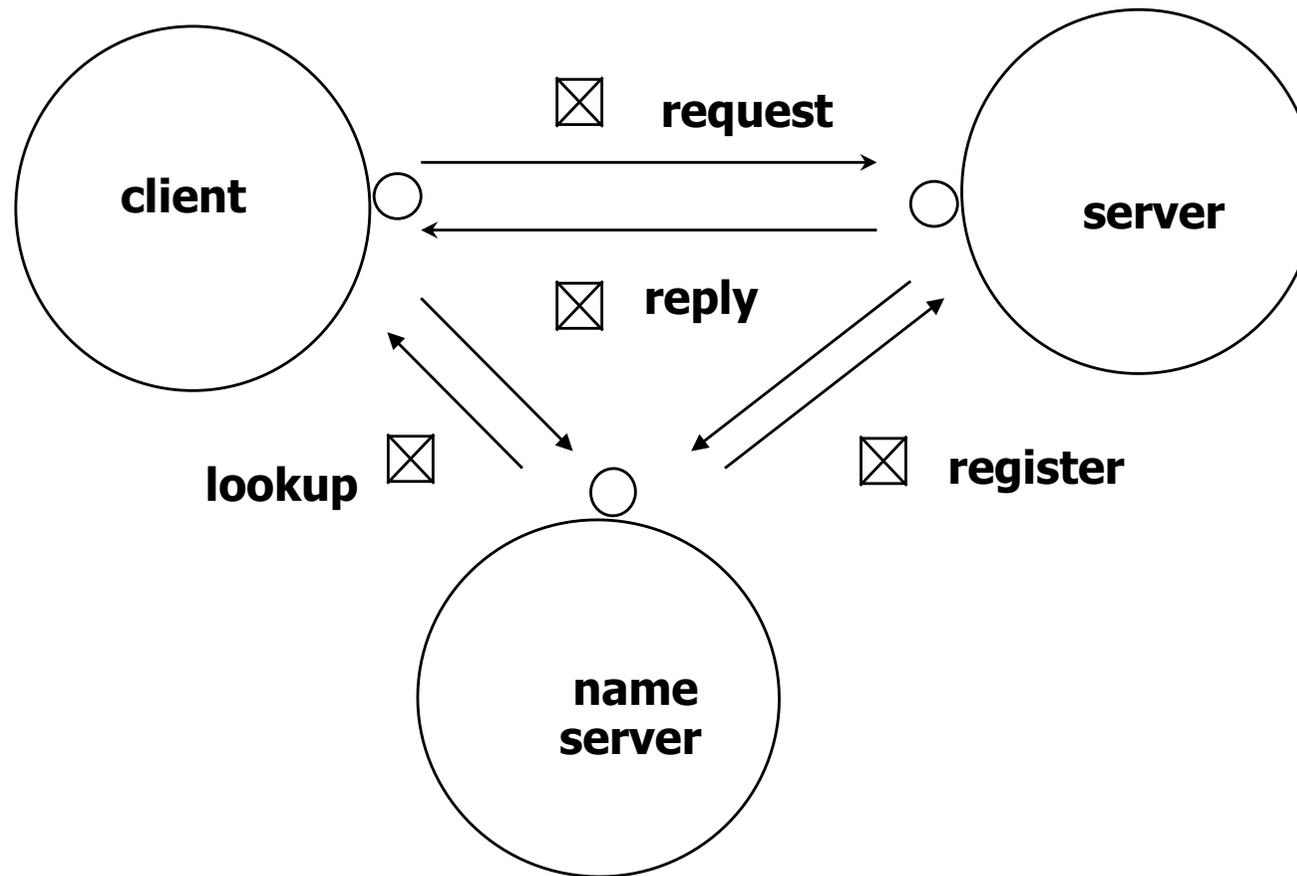
```
ChannelServices.RegisterChannel(new TcpChannel());  
HelloServer obj = (HelloServer)Activator.GetObject(  
    typeof(Examples.HelloServer),  
    "tcp://localhost:8085/SayHello");
```

Web services

```
URL wsdlURL = new URL("http://localhost:8080/indexer?wsdl");  
Service service = Service.create(wsdlURL, IndexerService.QNAME);  
proxy = service.getPort(IndexerAPI.class);
```

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

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 servidor de nomes?

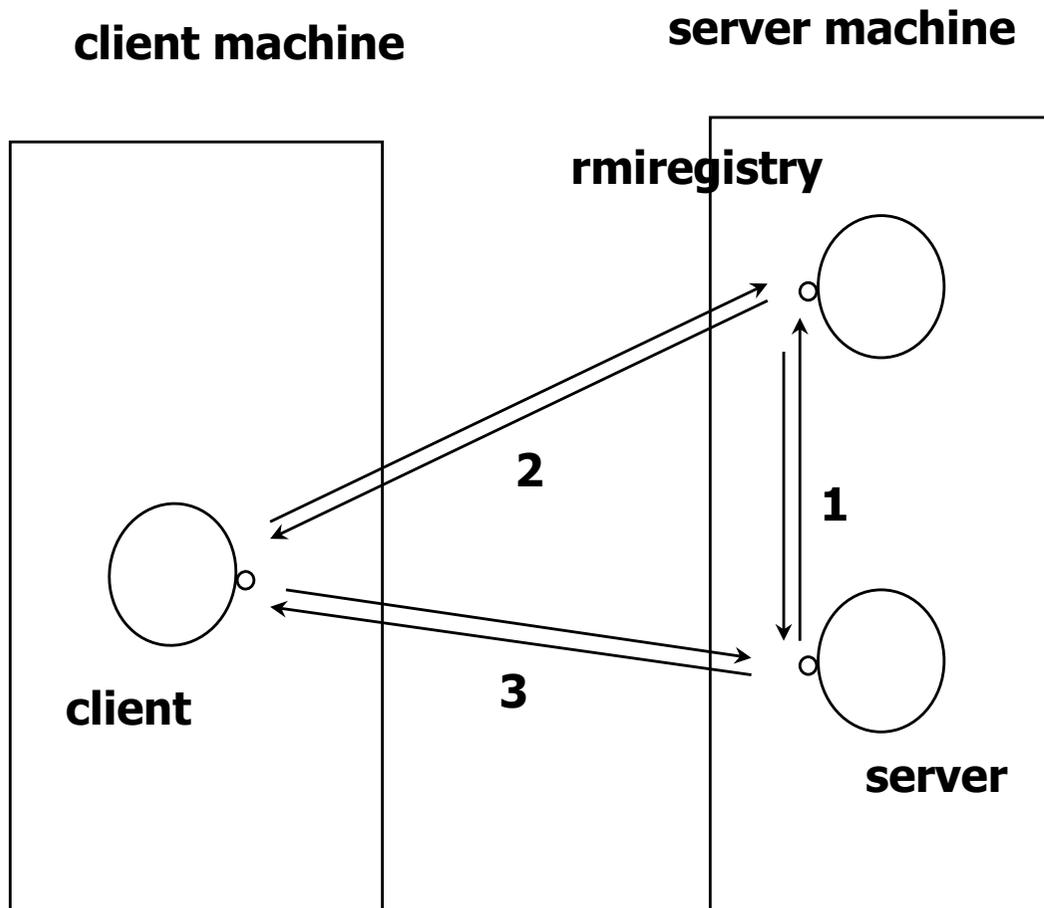
- Nome do objecto indica máquina em que está o servidor de nomes
- Servidor de nomes descoberto por multicast/broadcast

O servidor de nomes pode ser único ?

- No Java RMI não
- No SOAP, o espaço de nomes do UDDI é único e global

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

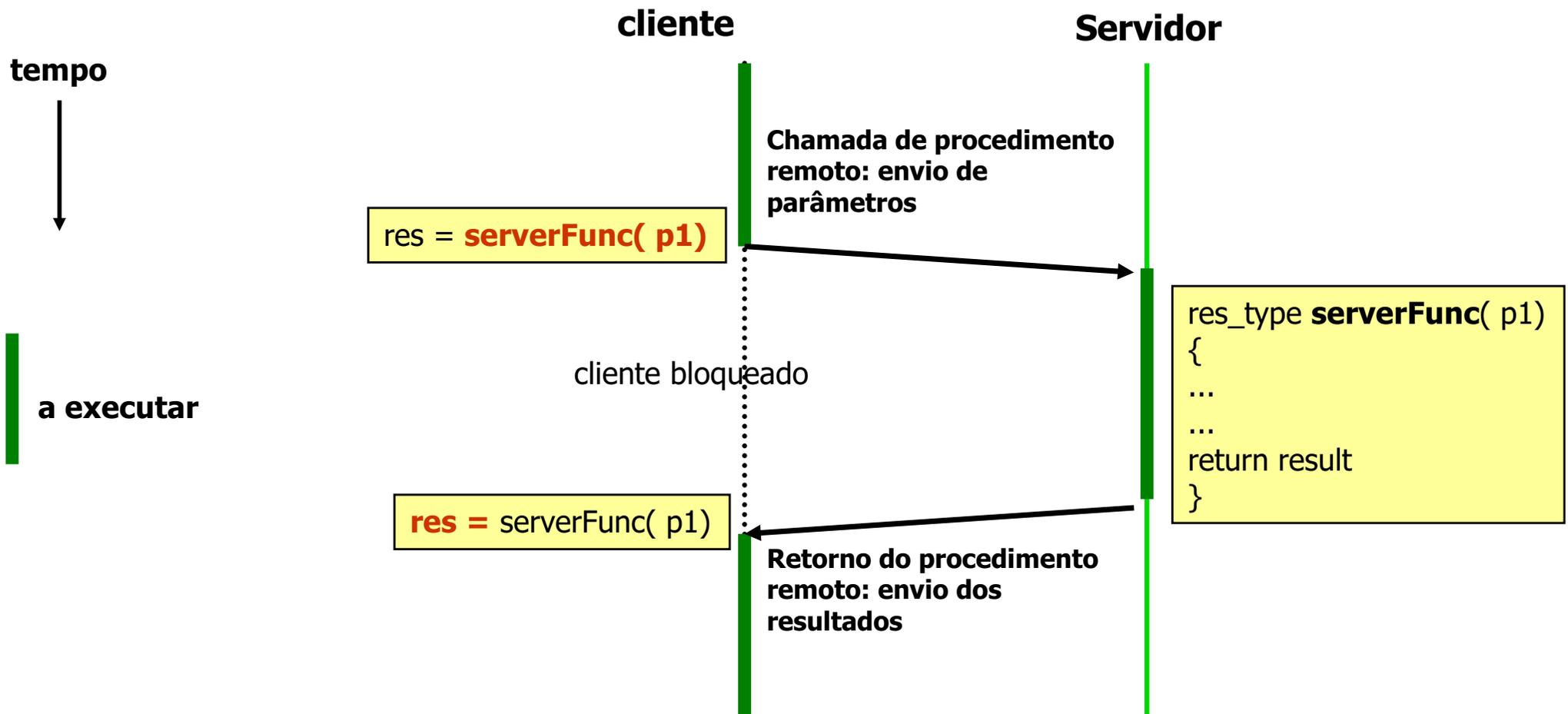
Porque é que esta solução diminui os problemas de segurança?

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 INVOCAÇÃO 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.

DIMENSÕES DO PROBLEMA

Protocolo de transporte

Semântica da invocação

PROTOCOLO UDP vs. TCP/HTTP

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 TCP / HTTP

O cliente usa conexão TCP (ou HTTP) para contactar o servidor e enquanto não receber resposta não avança com outro pedido

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:

- 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)

- Não se sabe se a operação foi executada ou não.

PROTOCOLO UDP

O cliente usa o protocolo UDP para contactar o servidor e enquanto não receber resposta a 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 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

- Não se sabe se a operação foi executada ou não.

DIMENSÕES DO PROBLEMA

Protocolo de transporte

Semântica da invocação

- May be
- At Least once
- Operações idempotentes
- At most once
- Exactly once

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?

- Envio simples sem retransmissões

```
clt: s.send(msg)
```

```
srv: msg = s.receive()  
exec(msg)
```

SEMÂNTICA DA INVOCAÇÃO REMOTA

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

```
clt: forever
    s.send(msg)
    try
        [id,res] = s.receive()
        if( id == msg.id)
            // executou 1+ vezes
            break
    catch InterruptedException
        continue
```

```
srv: msg = s.receive()
    res = exec(msg)
    s.send( [msg.id,res])
```

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.

SEMÂNTICA DA INVOCAÇÃO REMOTA

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 ...

```
clt: s.send(msg)
      try
          ack[id] = s.receive()
          // executou 1 vez
      catch InterruptedException
          // executou 0 ou 1 vez
```

```
srv: msg = s.receive()
      res = exec(msg)
      s.send( ack[msg.id])
```

SEMÂNTICA DA INVOCAÇÃO REMOTA

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 (código assume: servidor com apenas um thread)

```
clt: forever ou for n = 1 to K
    s.send(msg)
    try
        [id,res] = s.receive()
        if( id == msg.id)
            // executou 1 vez
            break
    catch InterruptedException
        continue
```

```
srv: msg = s.receive()
    if( ! cache.contains( msg.id))
        res = exec(msg)
        cache.put( msg.id, res)
    res = cache.get( msg.id)
    s.send( [msg.id,res])
```

SEMÂNTICA DA INVOCAÇÃO REMOTA

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 (código assume: servidor com apenas um thread). O que acontece quando servidor falha?

```
clt: forever ou for n = 1 to K
    s.send(msg)
    try
        [id,res] = s.receive()
        if( id == msg.id)
            // executou 1 vez
            break
    catch InterruptedException
        continue
```

```
srv: msg = s.receive()
    if( ! cache.contains( msg.id))
        res = exec(msg)
        cache.put( msg.id, res)
    res = cache.get( msg.id)
    s.send( [msg.id,res])
```

SEMÂNTICA DA INVOCAÇÃO REMOTA

Exactly once (exactamente uma vez):

- Garante-se a execução exatamente uma vez (a menos que existam falhas permanentes)

Protocolo?

- Protocolo com re-emissões, filtragem de duplicados, estado gravado em memória estável e re-execução quando cliente reinicia (código assume: servidor com apenas um thread)

```
clt: log.log( [msg])  
  forever  
    s.send(msg)  
  try  
    [id,res] = s.receive()  
    if( id == msg.id)  
      log.log( [msg,res])  
      //executou 1 vez  
      break  
  catch InterruptedException  
    continue
```

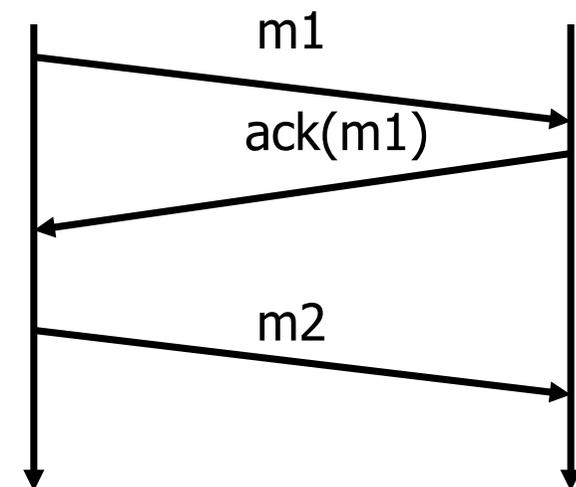
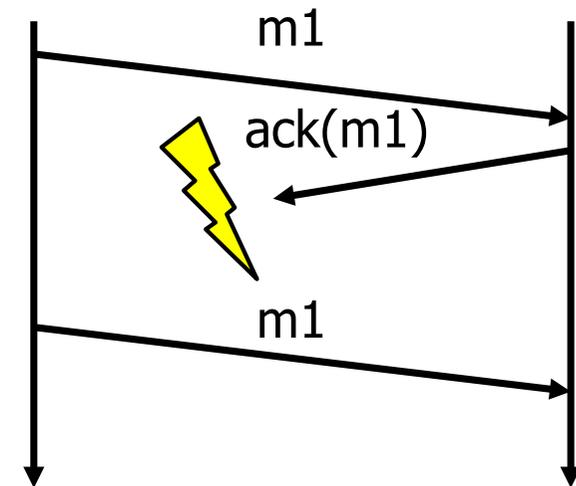
```
srv: msg = s.receive()  
  atomic  
    if( ! log.contains( msg.id))  
      res = exec(msg)  
      log.put( msg.id, res)  
  
  res = cache.get( msg.id)  
  s.send( [msg.id,res])
```

FILTRAGEM DE DUPLICADOS (1)

Problema: Como é que se identifica que uma mensagem recebida é um duplicado?

As mensagens devem ter um identificador. Quem recebe a mensagem guarda o identificador.

Para o servidor estas situações são indistinguíveis, se m_1 puder ser igual a m_2 . Como resolver?

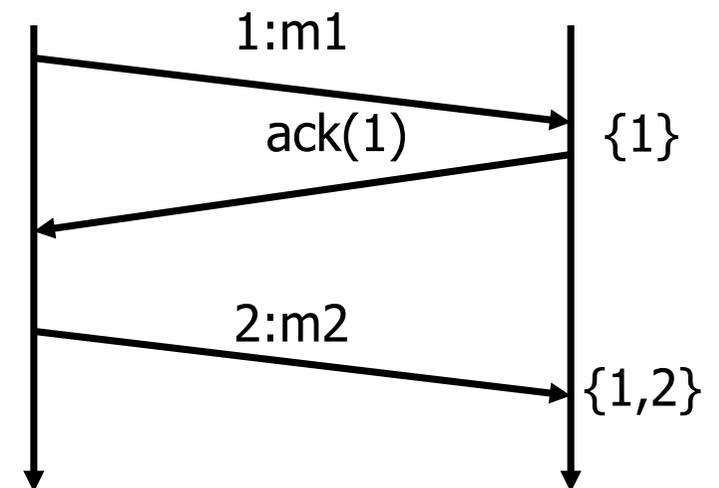
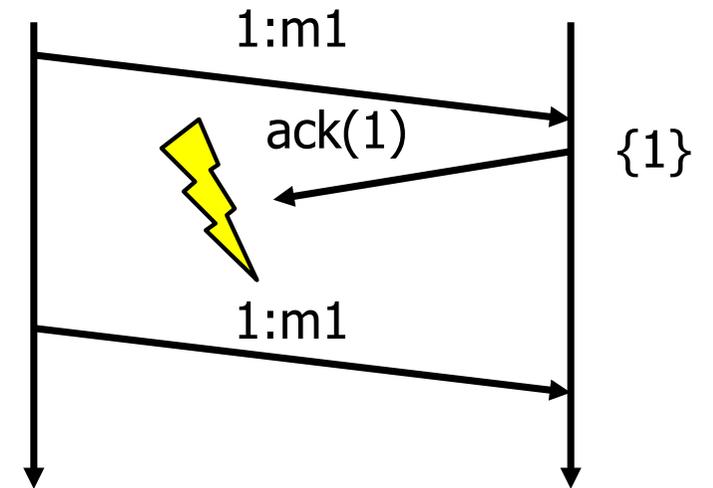


FILTRAGEM DE DUPLICADOS (2)

Problema: É necessário guardar todos os identificadores?

Se os identificadores forem sequenciais basta guardar o último (de cada emissor).
Porquê?

Se o emissor não voltar a enviar uma mensagem, pode-se remover a informação após um período alargado, ao fim do qual se saiba que a mensagem não pode ser enviada novamente.



OPERAÇÕES IDEMPOTENTES

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.

Pode ser usado imediatamente com semântica "pelo menos uma vez"

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).

Uma operação é **idempota** se o mesmo efeito produzido (deixando o 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.

Pode ser usado imediatamente com semântica "pelo menos uma vez"

SOLUÇÕES GERALMENTE ADOPTADAS

Java RMI

- “At most once” sobre TCP

.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”

REST

- “At most once” sobre HTTP

TRABALHO PRÁTICO (1)

“A solução não precisa de tolerar falhas de componentes.”

“Funcionalidade completa - Servidor REST funcional, com todas as operações a executar corretamente quando as falhas de comunicação não excedem 10 segundos.”

Que semântica/protocolo usar?

TRABALHO PRÁTICO (2)

“A solução não precisa de tolerar falhas de componentes.”

“Tratamento de falhas de longa duração (max: 2 valores) - O sistema deve funcionar corretamente quando as falhas de comunicação podem ser longas.”

Podemos usar o protocolo anterior? Se não que protocolo usar?

SISTEMAS DISTRIBUÍDOS I

Capítulo 4 – aula 7

Invocação de procedimentos e de métodos remotos

NA ÚLTIMA AULA: 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

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

SEMÂNTICAS DE INVOCAÇÃO

May be

At Least once

Operações idempotentes

At most once

Exactly once

Requer operação atômica

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

Ativação dos servidores

- Servidor a executar continuamente
- Servidor ativado quando necessário

Organização interna

- Iterativo vs. concorrente

ORGANIZAÇÃO DOS SERVIDORES: ATIVAÇÃO

Existem duas formas para lidar com os pedidos dos clientes:

- Existe apenas uma instância do código do servidor para atender todos os clientes
 - Aproximação mais comum
- Cria-se uma instância do código do servidor para atender cada cliente
 - E.g. .NET remoting: servidor *SingleCall*
 - REST em Java: cada pedido é tratado por um objeto criado no momento

Em Java REST, pode-se indicar que se pretende apenas uma instância

```
20
21 @Singleton
22 public class MessageResource implements MessageService {
23
24     private Random randomNumberGenerator;
25
```

VANTAGENS / DESVANTAGENS

Uma instância por pedido

- Não existem problemas de concorrência devido a múltiplos pedidos
- Não é possível manter estado na instância do servidor
 - Em geral, o estado duma aplicação é guardado numa base de dados

Uma instância apenas

- Necessário lidar com concorrência devido a múltiplos pedidos
- É possível manter estado na instância do servidor

ATIVACÃO DE OBJETOS REMOTOS (E.G. JAVA RMI)

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

Solução: ativam-se os objetos remotos apenas quando necessário

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

Activator: servidor responsável por:

- Manter informação sobre os objetos ativáveis
- Ativar os objetos remotos quando solicitado por um cliente
- Manter informação sobre localização dos objetos ativados

Objeto remoto *passivo* (quando não ativado)

- Código
- Estado do objeto *marshalled*

Referência remota mantém informação necessária para solicitar a ativação do objecto

ORGANIZAÇÃO DOS SERVIDORES

Ativação dos servidores

- Servidor a executar continuamente
- Servidor ativado quando necessário

Organização interna

- **Iterativo vs. concorrente**

ORGANIZAÇÃO DOS SERVIDORES: *THREADS*

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

- Modelo simples

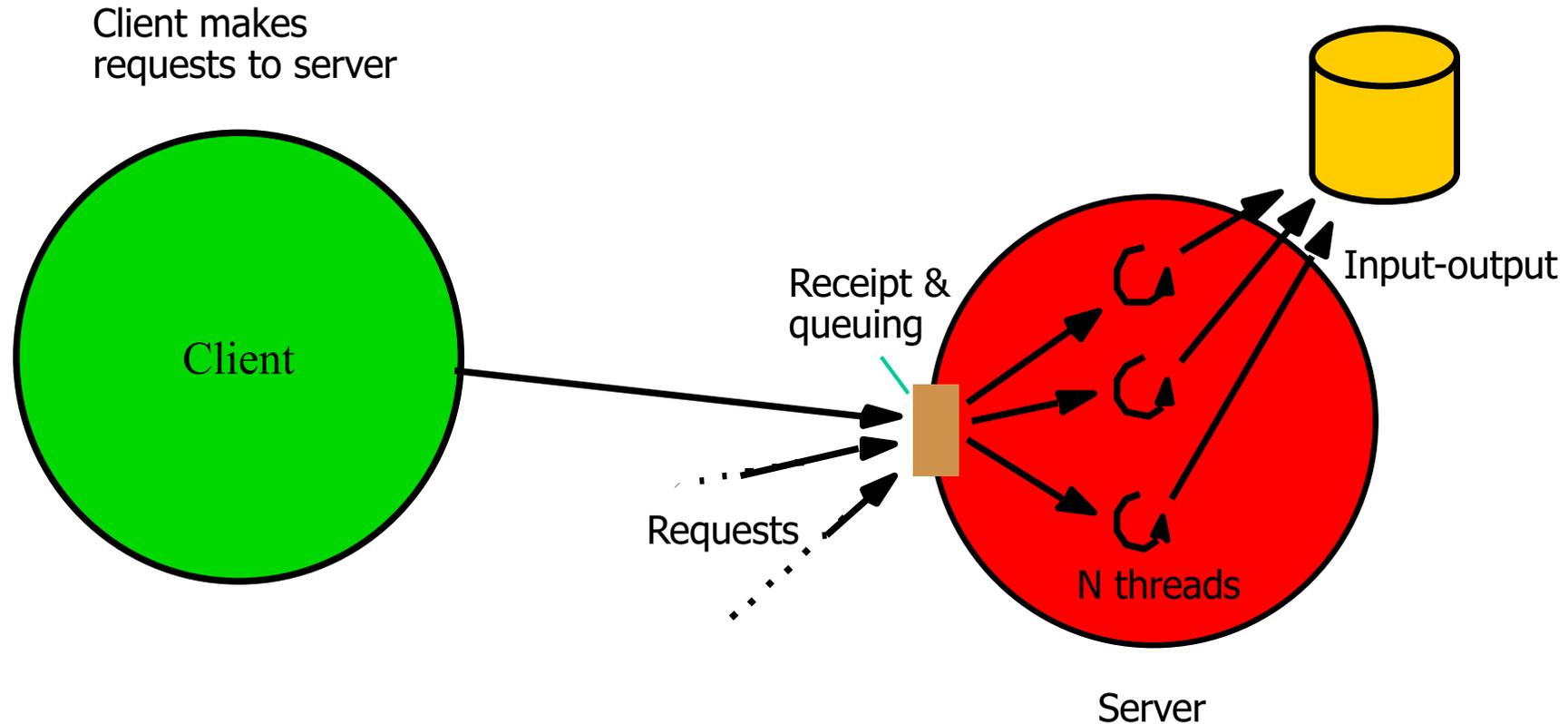
Para alguns tipos de serviços, esta aproximação pode ter um desempenho inadequado

- Exemplos: servidores de bases de dados, de ficheiros, etc. Porquê?
- Exemplo: serviços que chamam outros serviços em ambos os sentidos (A->B e B->A). Porquê?

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

Permite aproveitar os recursos computacionais da máquina.

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?

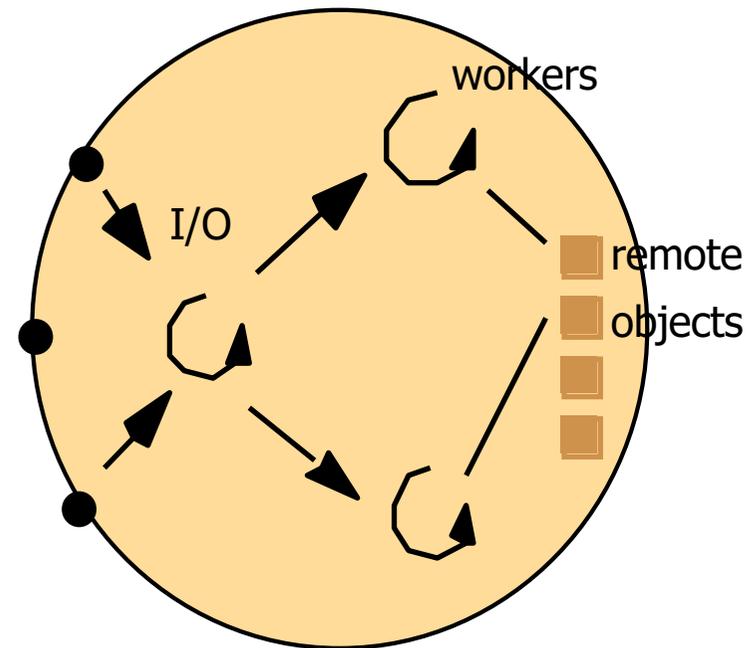
THREAD POR PEDIDO

Cada pedido é atendido por um *thread*.

Pode-se criar um *thread* quando chega um pedido ou existir um conjunto de *threads* que podem ser usadas para atender os pedidos.

Podem existir múltiplos *threads* a executar no mesmo servidor/objeto.

- Necessário executar controlo de concorrência no acesso aos dados.



a. Thread-per-request

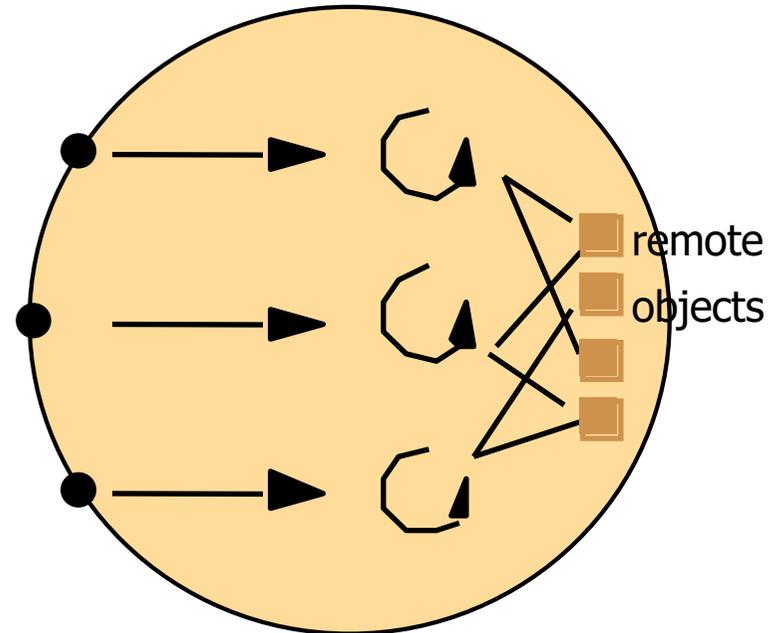
THREAD POR CONEXÃO

Cada conexão é atendida por um *thread*.

Pode-se criar um *thread* quando se cria uma conexão ou existir um conjunto de *threads* que podem ser usadas para atender os pedidos.

Podem existir múltiplos *threads* a executar no mesmo servidor/objeto.

- Necessário executar controlo de concorrência no acesso aos dados.



b. Thread-per-connection

THREAD POR OBJETO

Os pedidos de um objeto são atendidos todos pelo mesmo *thread*, de forma sequencial.

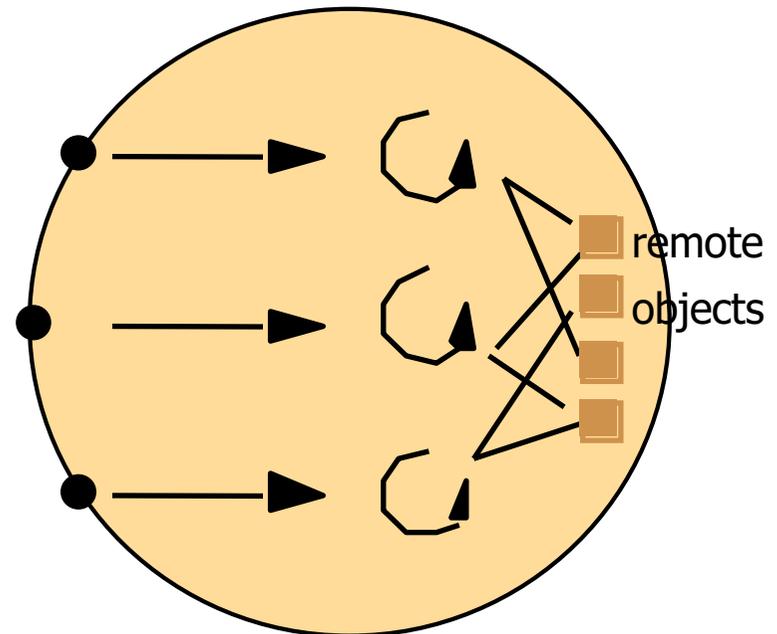
Cada objeto tem um *thread* associada.

Não existem problemas de concorrência no acesso ao estado dum servidor/objeto.

- Podem existir problemas se um servidor puder aceder a outros objetos.

Pode levar a problemas de deadlocks se comunicação com outros servidores for síncrona.

- Modelo de atores e CSP disponível em linguagens como o Erlang e o Go.



b. Thread-per-connection

ORGANIZAÇÃO DOS *THREADS*

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*
- Esta aproximação permite dimensionar o número de *threads* à capacidade da máquina em que o servidor corre.

CONTROLO DE CONCORRÊNCIA NOS SERVIDORES

Quando existem múltiplos threads a executar concorrentemente e a aceder aos mesmos recursos é necessário controlar estes acessos

- Porquê?

Porque durante a execução dum operação no servidor, o estado das variáveis pode ser alterado por outro *thread*

PROBLEMA: EXEMPLO

@WebService

```
public class ServerWS {  
    private static final int SIZE=10  
    private List<String> values;  
    public ServerWS() {  
        values = new ArrayList<>(SIZE);  
    }
```

@WebMethod

```
public void add(String v) {  
    values.add(v);  
}
```

@WebMethod

```
public boolean contains( String v) {  
    for( String s : values)  
        if( v.equalsIgnoreCase(s))  
            return true;  
    return false;  
}
```

...

```
}
```

PROBLEMA: EXEMPLO

@WebService

```
public class ServerWS {
```

```
    private static final int SIZE=10
```

```
    private List<String> values;
```

```
    public ServerWS() {
```

```
        values = new ArrayList<>(SIZE);
```

```
    }
```

@W

```
pub
```

```
}
```

@W

```
pub
```

```
}
```

```
...
```

```
}
```

Execução concorrente do método *add* e *contains* pode levar a exceções
Os dois métodos acedem concorrentemente à lista values.

```
aula 3 — java — 142x39  
java.util.ConcurrentModificationException  
at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)  
at java.util.ArrayList$Itr.next(ArrayList.java:851)  
at sd.aula1.srv.ServerWS.contains(ServerWS.java:24)  
at sun.reflect.GeneratedMethodAccessor7.invoke(Unknown Source)  
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  
at java.lang.reflect.Method.invoke(Method.java:497)  
at sun.reflect.misc.Trampoline.invoke(MethodUtil.java:71)  
at sun.reflect.GeneratedMethodAccessor3.invoke(Unknown Source)  
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  
at java.lang.reflect.Method.invoke(Method.java:497)  
at sun.reflect.misc.MethodUtil.invoke(MethodUtil.java:275)  
at sun.reflect.GeneratedMethodAccessor2.invoke(Unknown Source)  
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  
at java.lang.reflect.Method.invoke(Method.java:497)  
at com.sun.xml.internal.ws.api.server.MethodUtil.invoke(MethodUtil.java:68)  
at com.sun.xml.internal.ws.api.server.InstanceResolver$1.invoke(InstanceResolver.java:235)  
at com.sun.xml.internal.ws.server.InvokerTube$2.invoke(InvokerTube.java:134)  
at com.sun.xml.internal.ws.server.sei.SEIInvokerTube.processRequest(SEIInvokerTube.java:73)
```

TÉCNICAS DE CONTROLO DE CONCORRÊNCIA

Monitores (e.g. métodos/blocos synchronized no Java)

Locks

Estruturas de dados concorrentes (e.g. java.util.concurrent)

Transações

- É comum nos servidores aplicativos todo o estado persistente estar numa base de dados, podendo-se usar as transações da base de dados como mecanismo de controlo de concorrência

MÉTODOS SINCRONIZADOS

Num dado momento, apenas um thread pode executar nos métodos marcados como sincronizados

@WebService

public class ServerWS {

private static final int SIZE=10

private List<String> values;

public ServerWS() {

values = **new ArrayList<>(SIZE);**

}

@WebMethod

public synchronized void add(String v) {

values.add(v);

}

@WebMethod

public synchronized boolean contains(String v) {

for(String s : values)

if(v.equalsIgnoreCase(s))

return true;

return false;

}

...

}

BLOCOS SINCRONIZADOS

Num dado momento apenas um thread pode executar nos vários blocos synchronized relativos a um dado objeto. Permite aumentar a concorrência.

@WebService

```
public class ServerWS {  
    private static final int SIZE=10  
    private List<String> values;  
    public ServerWS() {  
        values = new ArrayList<>(SIZE);  
    }  
}
```

@WebMethod

```
public void add(String v) {  
    synchronized ( values) {  
        values.add(v);  
    }  
}
```

@WebMethod

```
public boolean contains( String v) {  
    synchronized ( values) {  
        for( String s : values)  
            if( v.equalsIgnoreCase(s))  
                return true;  
    }  
    return false;  
}  
...  
}
```

ESTRUTURAS DE DADOS CONCORRENTES

@WebService

public class ServerWS {

private static final int SIZE=10 ;

private List<String> values;

public ServerWS() {

values = **new CopyOnWriteArrayList<>(SIZE);**

}

@WebMethod

public void add(String v) {

values.add(v);

}

@WebMethod

public boolean contains(String v) {

for(String s : values)

if(v.equalsIgnoreCase(s))

return true;

return false;

}

...

}

O pacote java.util.concurrent tem implementações que permitem acesso concorrente, incluindo a iteradores. e.g. CopyOnWriteArrayList cria uma cópia da lista sempre que a mesma é alterada.

JAVA.UTIL.CONCURRENT

CopyOnWriteArrayList, CopyOnWriteArraySet

Criam uma cópia da estrutura de dados sempre que a mesma é alterada

ConcurrentHashMap, ConcurrentSkipListSet

Lidam com acesso concorrentes.

Nota: acessos sucessivos não são atômicos

```
ConcurrentHashMap<String,String> map = ...
```

```
if(! map.containsKey( "SD"))
```

```
    map.put("SD",str);
```

Nada garante que dois threads concorrentemente não entrem no ramo *then* do *if*, fazendo put de dois valores diferentes.

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

GARGABE-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 **nenhuma** referência para o objeto **existir**, este pode ser **removido**

Problema: como é que se sabe se existem referências ou não?

GARBAGE-COLLECTION: JAVA RMI

Entre máquinas virtuais, o sistema Java executa um algoritmo de *garbage-collection* distribuído para remover (apagar) objetos 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

GARBAGE-COLLECTION: JAVA RMI

Entre máquinas virtuais, o sistema Java executa um algoritmo de *garbage-collection* distribuído para remover (apagar) objetos 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
 - A VM reenvia a informação periodicamente
- 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.

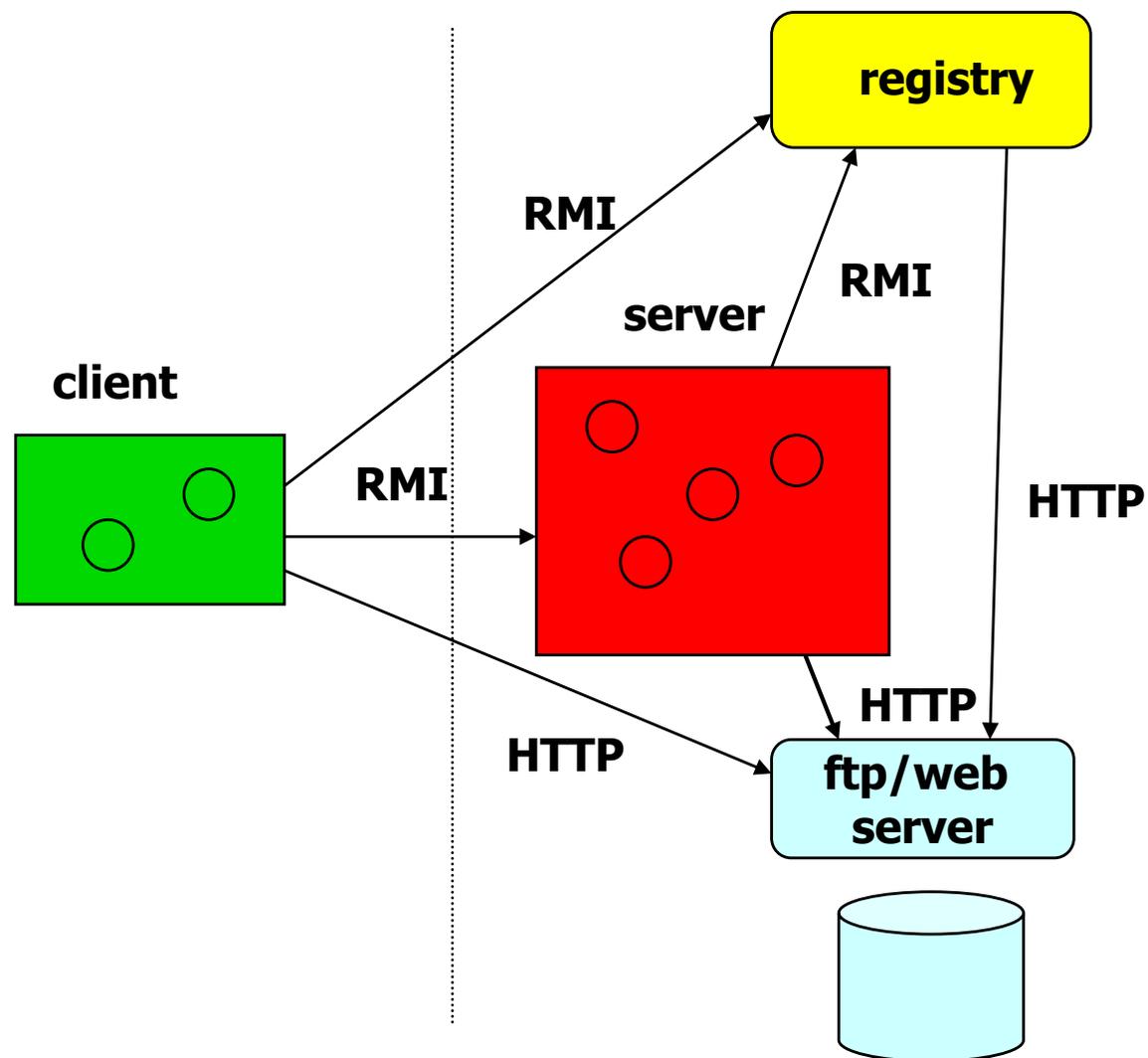
CARREGAMENTO DE CLASSES: SOLUÇÃO JAVA

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

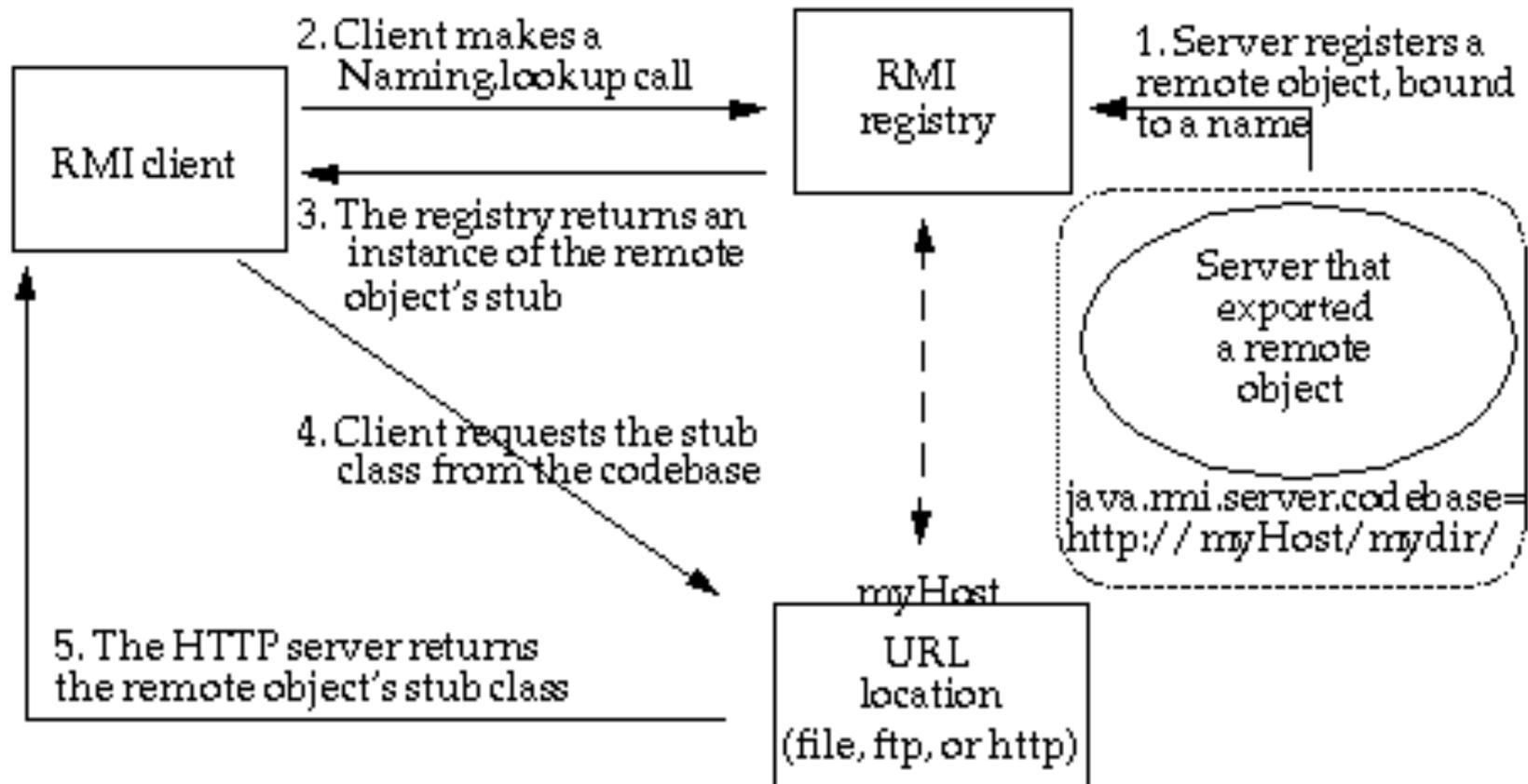
Isto aplica-se às classes do objecto remoto, do *stub*, do esqueleto, dos parâmetros e do valor de retorno dos métodos

O carregamento faz-se remotamente se necessário

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

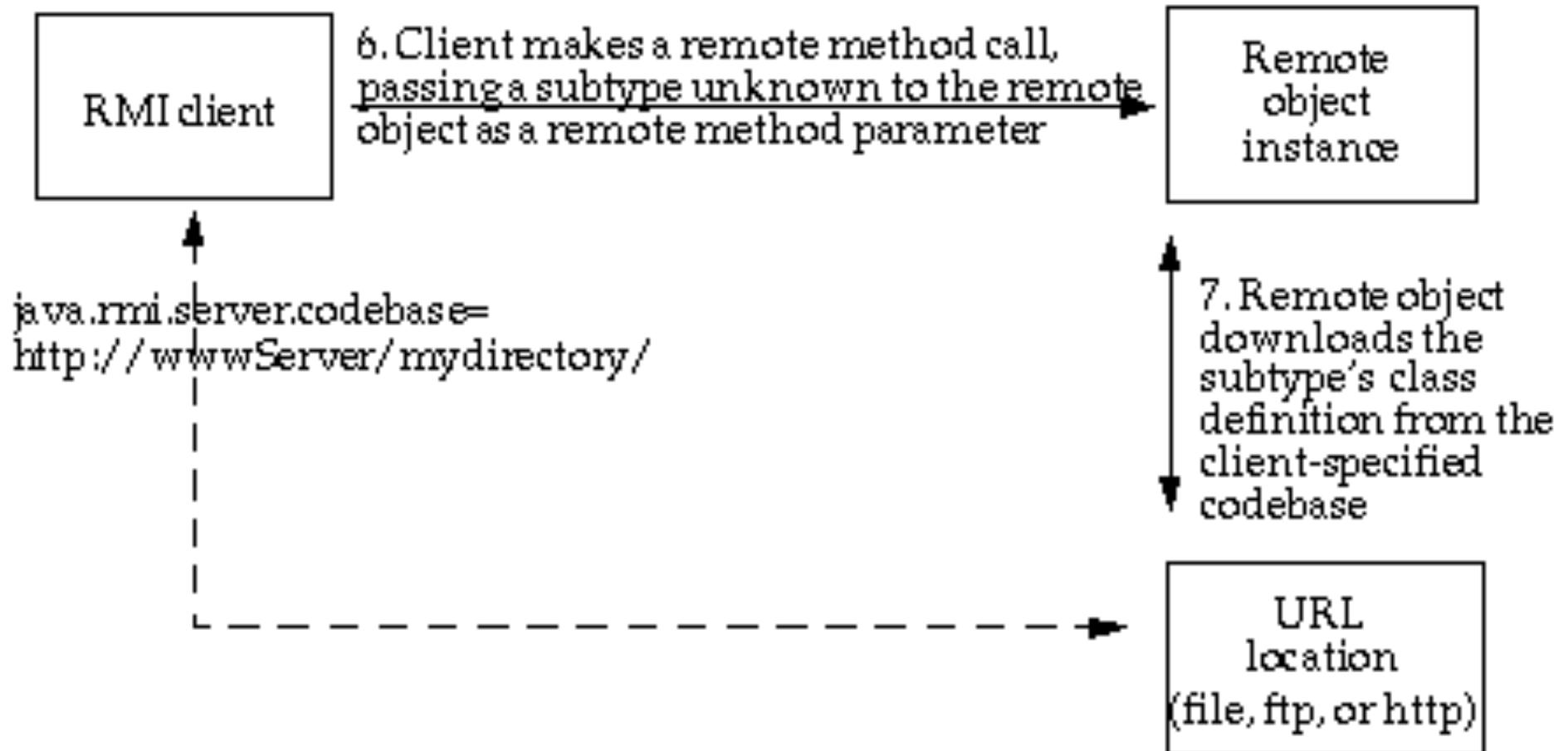


CARREGAMENTO DE CÓDIGO NO CLIENTE



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, 2011

- RMI/RPCs - capítulo 5.
- Representação de dados e protocolos - capítulo 4.3.
- Web services – capítulo 9