

# 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 aplicacional?

# INVOCACÃO 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**.

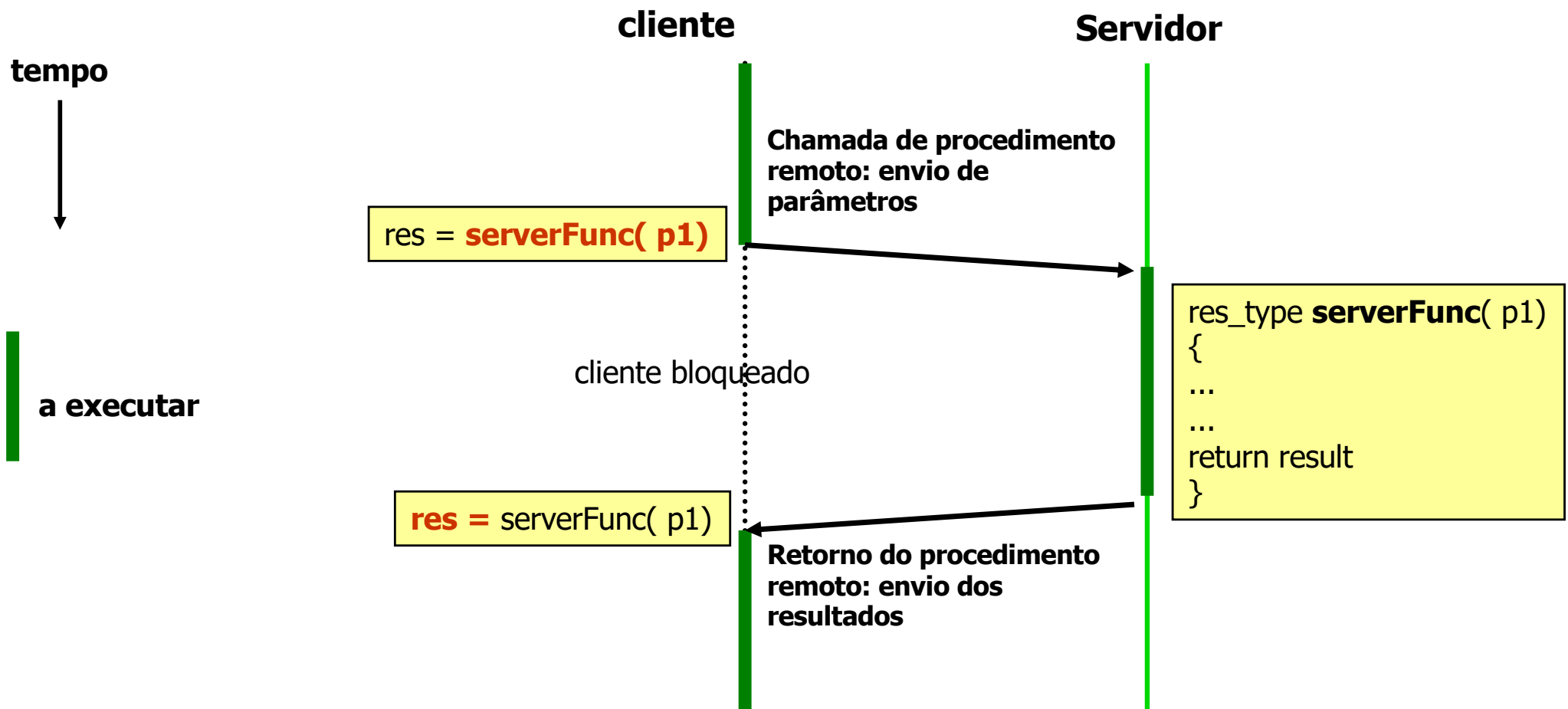
**Invocação Remota de Procedimentos (RPCs)**, quando são executados funções/procedimentos remotamente.

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

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

- JAVA RMI, .NET Remoting, Corba.
- Web Services (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

# INVOCACÃO 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)

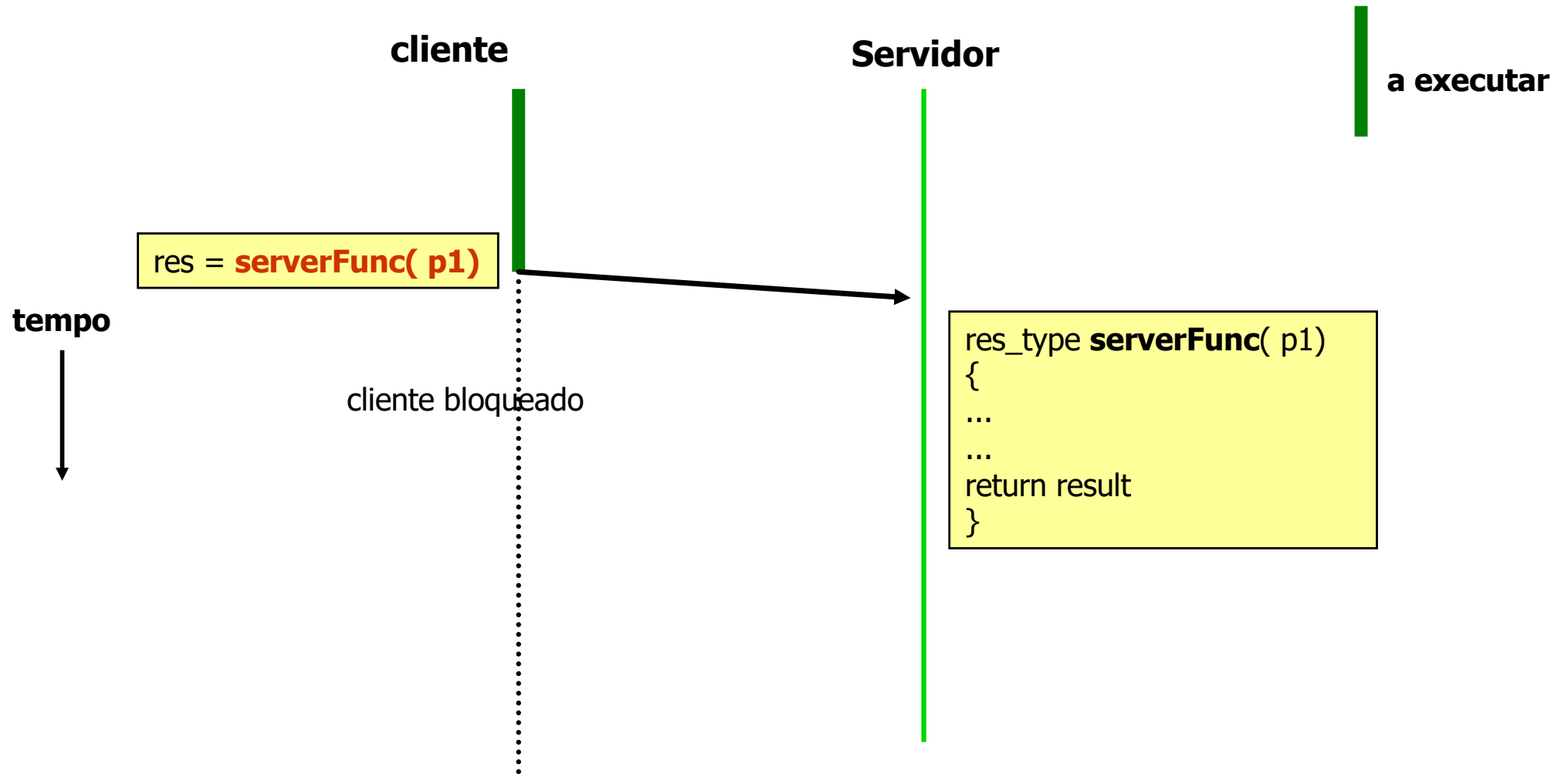
- Construção, envio, receção e tratamento das mensagens
- Tratamento básico de erros (devem ser tratados ao nível da aplicação)
- Heterogeneidade da representação dos dados

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)

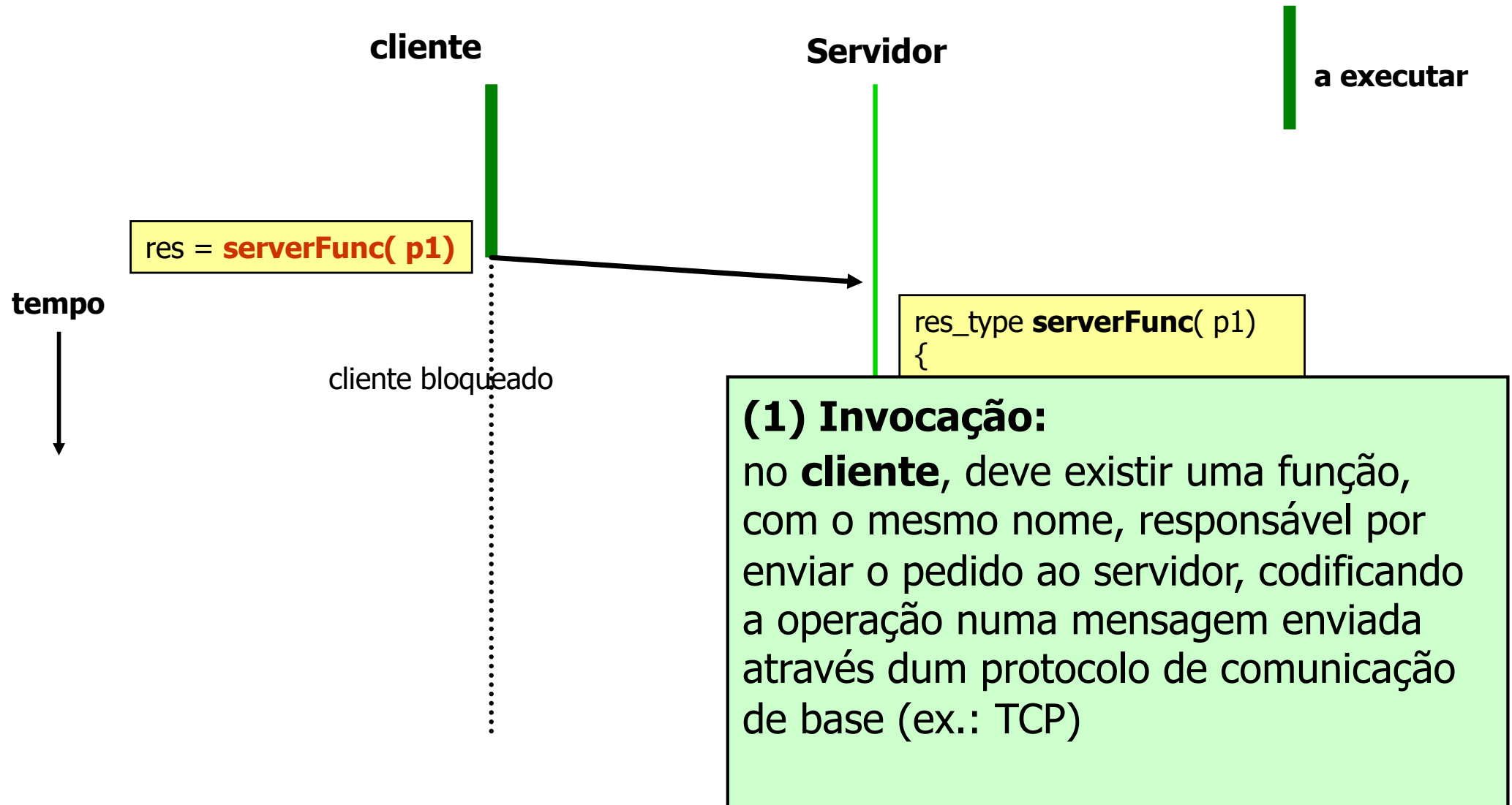
Cliente

```
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)
```

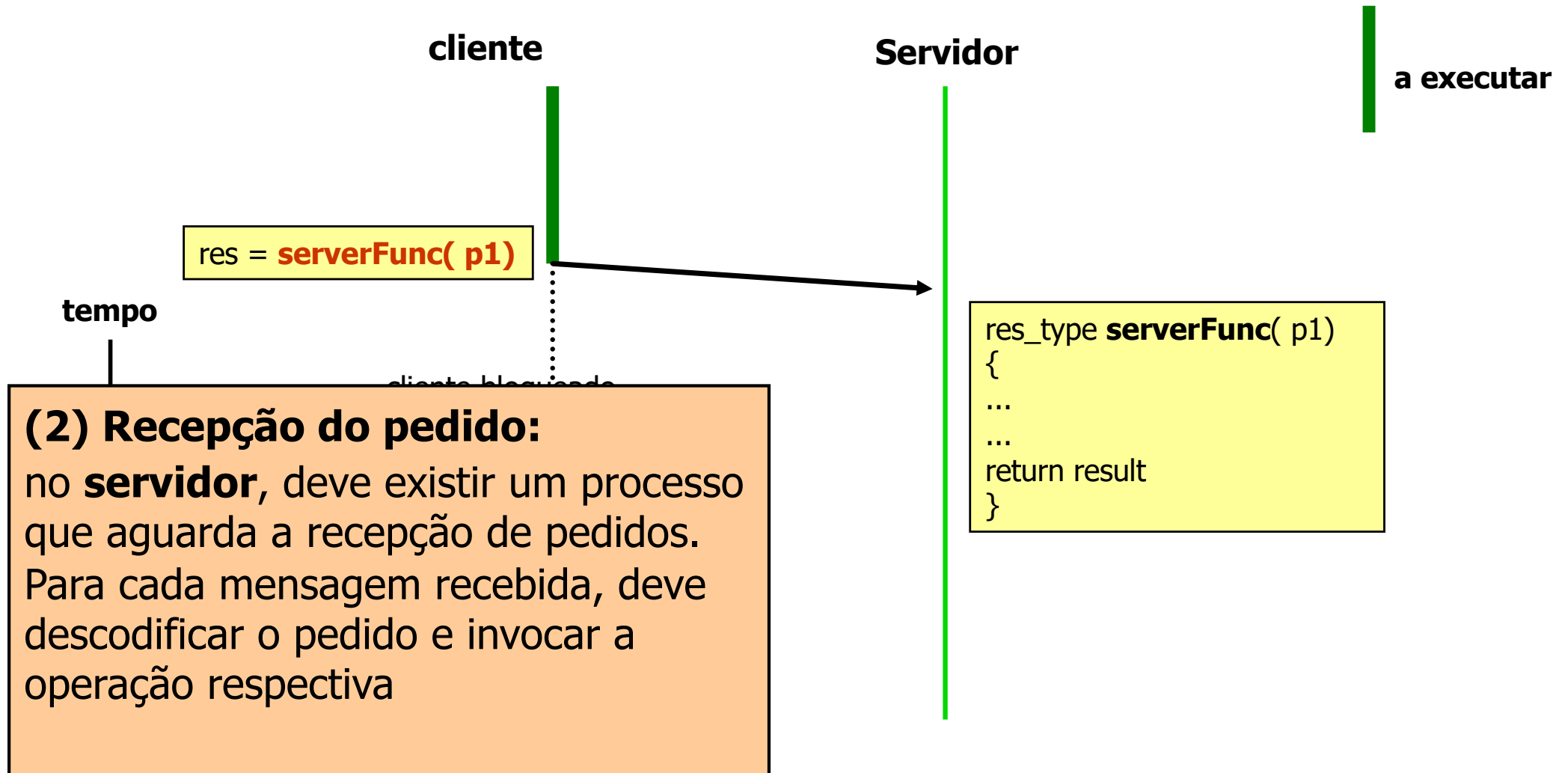
```
res_type serverFunc( T1 p1)  
    s = new Socket( host, port)  
    s.send( msg["serverFunc",[p1]])
```

## (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

(LINGUAGEM)

### 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

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

cliente bloqueado

# 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]])
```

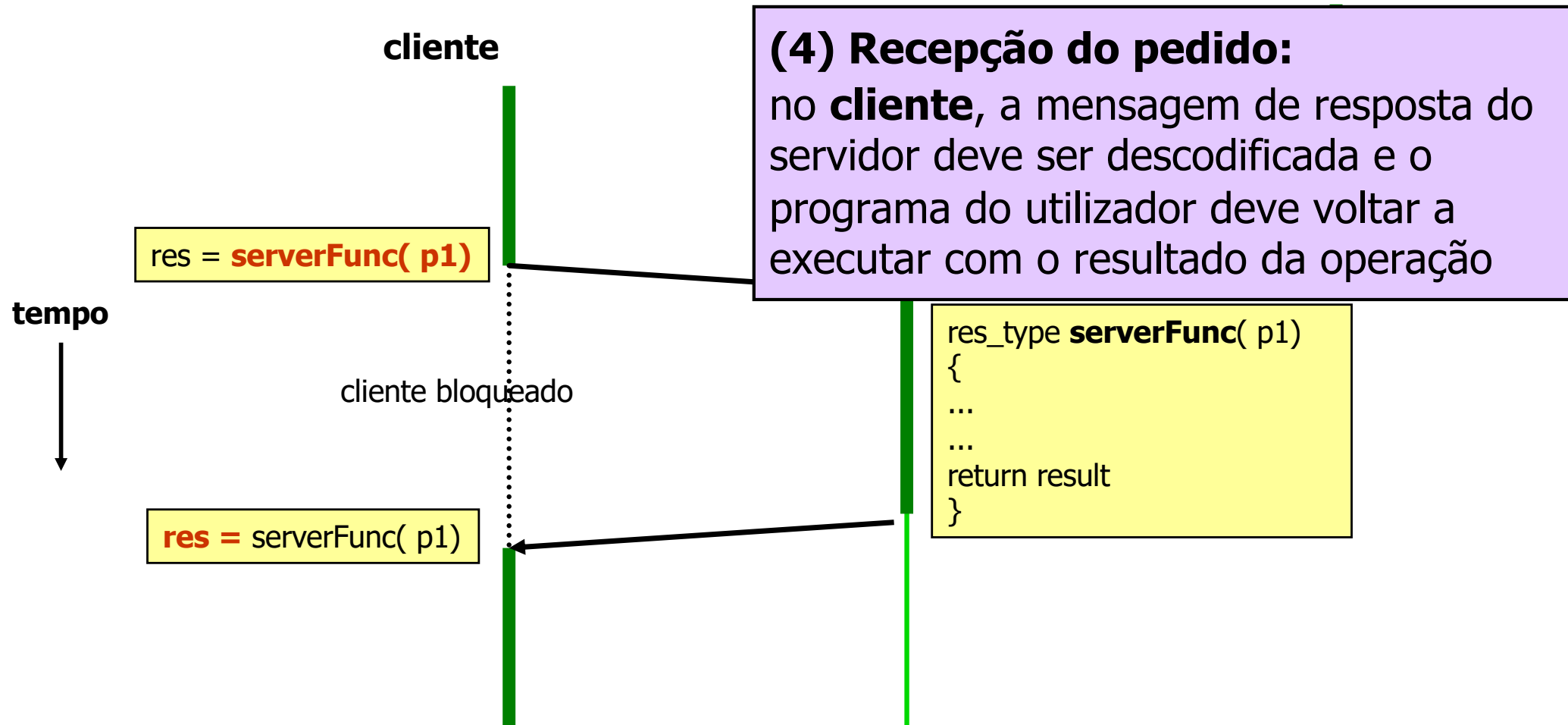
## Servidor

```
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; gRPC.

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.

# SISTEMAS DISTRIBUÍDOS

## Capítulo 4

### Invocação remota

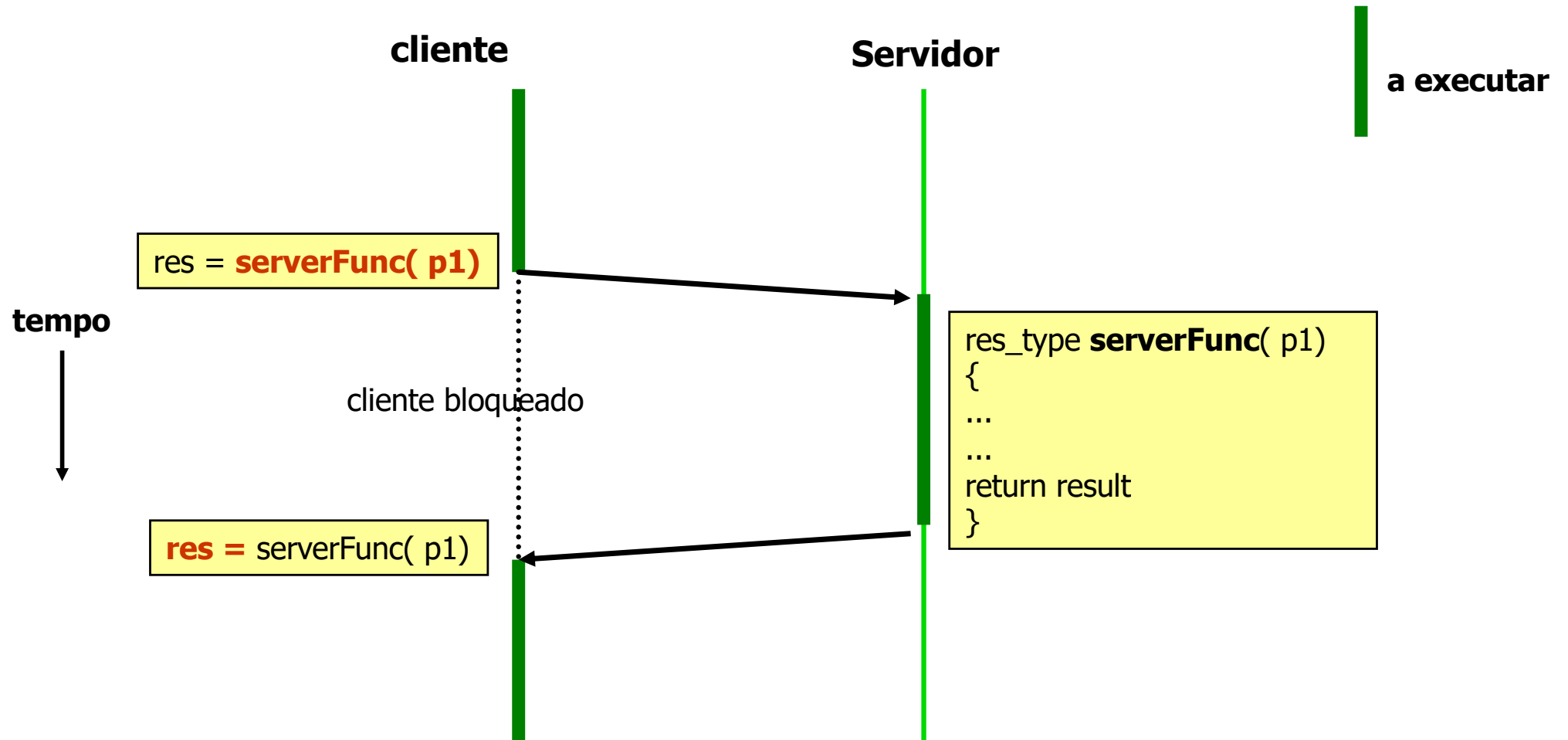
# NA ÚLTIMA AULA

## Invocação remota de procedimentos/objectos

- Motivação
- Modelo
- Concorrência no servidor
- Definição de interfaces e método de passagem de parâmetros
- Codificação dos dados
- Mecanismos de ligação (binding)
- Protocolos de comunicação
- Sistemas de objetos distribuídos



# INVOCAÇÃO REMOTA



# NA ÚLTIMA AULA

## 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
```

## *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)

# NA ÚLTIMA AULA

## **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

Os IDL são usados para definir as interfaces (não o código das operações). Aproximações possíveis:

Usar subconjunto 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

# AGENDA

## Invocação remota de procedimentos/objectos

- Motivação
- Modelo
- Definição de interfaces e método de passagem de parâmetros
- Codificação dos dados
- Organização do servidor
- Mecanismos de ligação (binding)
- Protocolos de comunicação
- Sistemas de objetos distribuídos

# INTERFACE DEFINITION LANGUAGES (IDL)

**Problema:** Necessário especificar quais as operações que estão disponíveis:

- Interface do serviço – assinatura das funções
- Tipos e constantes usados

Em alguns sistemas, os clientes e os servidores podem ser implementados em linguagens diferentes.

Os IDL são usados 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 subconjunto 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
```

Interfaces remotos  
estendem **Remote**

```
{
```

```
    public void depositar ( float quantia )  
        throws RemoteException;
```

```
    public void levantar ( float quantia )  
        throws SaldoDescoberto, RemoteException;
```

```
    public float saldoActual ( )  
        throws RemoteException;
```

```
}
```

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

```
public class ServiceClass :
    System.MarshalByRefObject
{
    public void depositar(float quantia) {
        Console.WriteLine (quantia);...
    }
    ...
}
```

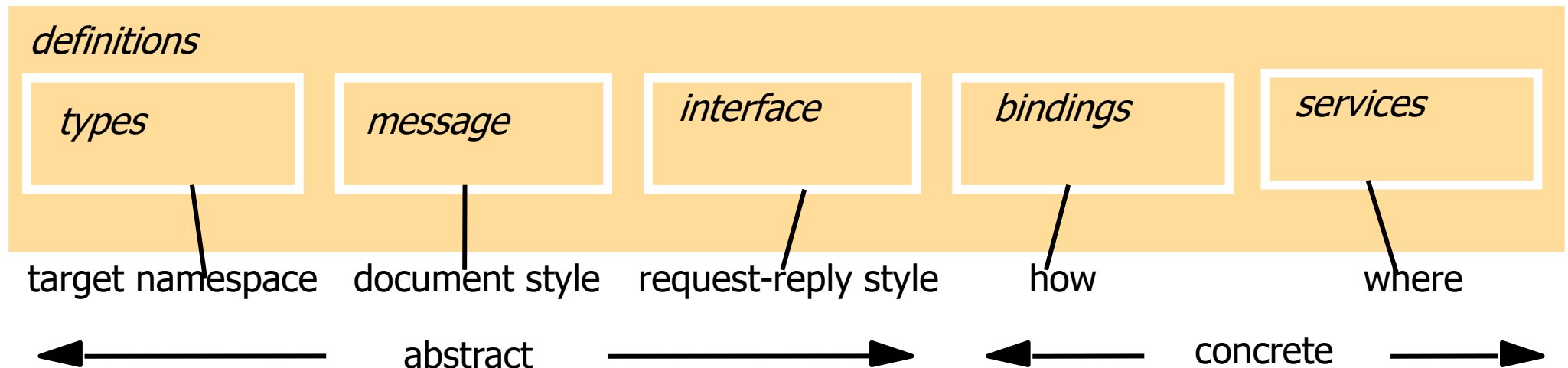
No .NET Remoting não é necessário definir qual a interface remota – esta pode ser inferida a partir da definição do servidor

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** String[] 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);

}

}

# gRPC

Interface definido num service

```
service UsersServer {  
    rpc createUser(CreateUserRequest) returns (CreateUserReply) {}  
    rpc getUser(GetUserRequest) returns (UserReply) {}  
    rpc updateUser(UpdUserRequest) returns (UserReply) {}  
    rpc deleteUser(UserPwdRequest) returns (UserReply) {}  
    rpc searchUsers(SearchRequest) returns (ListUserReply) {}  
    rpc verifyPassword(UserPwdRequest) returns (Void) {}  
}
```

Métodos definidos usando a keyword rpc. Pode ter apenas um parâmetro e um resultado, definido como mensagens protobuf

# GRPC (CONT.)

Message permite definir uma mensagem a ser transmitida.

```
message User {  
    optional string userId = 10;  
    optional string email = 11;  
    optional string fullName = 12;  
    optional string password = 13;  
}
```

Pode ter campos opcionais.

```
message CreateUserRequest {  
    User user = 20;  
}
```

Message pode ser construída à custa de outras mensagens.

# GRPC (CONT.)

```
enum ErrorCode {  
    OK = 0;  
    NO_CONTENT = 209;  
    ...  
}
```

Mensagem pode ter campos alternativos.

```
message CreateUserReply {  
    oneof status {  
        ErrorCode code = 30;  
        string userId = 31;  
    }  
}
```

# GRPC: CÓDIGO DO SERVIÇO

```
class GRPCUsersService extends UsersServerImplBase {  
    @Override  
    public void hello( CreateUserRequest request,  
                      StreamObserver<CreateUserReply> responseObserver) {  
        System.out.println("id:" + request.getUser().getUserId());  
  
        ...  
  
        CreateUserReply response = CreateUserResult.newBuilder()  
            .setUserId("47")  
            .build();  
  
        responseObserver.onNext(response);  
        responseObserver.onCompleted();  
    }  
    ...  
}
```

# GRPC: CÓDIGO DO SERVIDOR

```
public class GrpcServer {  
    public static void main(String[] args) {  
        Server server = ServerBuilder  
            .forPort(8080)  
            .addService(new GRPCUserService()).build();  
  
        server.start();  
        server.awaitTermination();  
    }  
}
```

# GRPC: CÓDIGO DO CLIENTE

```
ManagedChannel channel = ManagedChannelBuilder
    .forAddress("server_address", 8080)
    .usePlaintext()
    .build();

UsersServerBlockingStub stub
    = UsersServerGrpc.newBlockingStub(channel);

CreateUserReply reply = stub.createUser (
    CreateUserRequest.newBuilder()
        .setUserId ("47")
        ...
        .build());

channel.shutdown();
```



## gRPC (CONT.)

Ferramenta **proto** cria, a partir da especificação da interface:

- Skeleton do servidor – métodos devem ser redefinidos com implementação do método;
- Stub do cliente, que permite criar um cliente para efetuar uma invocação remota.

Diferentes variantes do proto permitem construir o código base para diferentes linguagens – nas práticas vamos usar o proto-java.

gRPC permite definir chamadas síncronas, assíncronas, callbacks do servidor para o cliente, etc.

# AGENDA

## Invocação remota de procedimentos/objectos

- Motivação
- Modelo
- Definição de interfaces e método de passagem de parâmetros
- Codificação dos dados
- Organização do servidor
- Mecanismos de ligação (binding)
- Protocolos de comunicação
- 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 de ser enviados como uma sequência/array de bytes

# REPRESENTAÇÃO DOS TIPOS PRIMITIVOS

Diferentes sistemas representam os tipos primitivos de formas diferentes

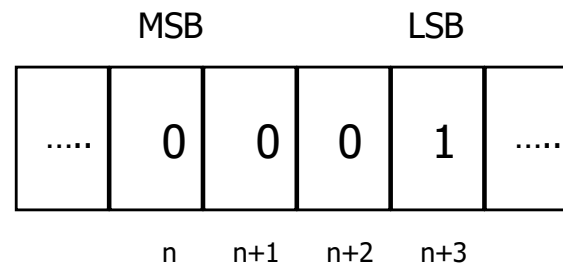
Inteiros armazenados por ordem diferente em memória – big-endian vs. little endian

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

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

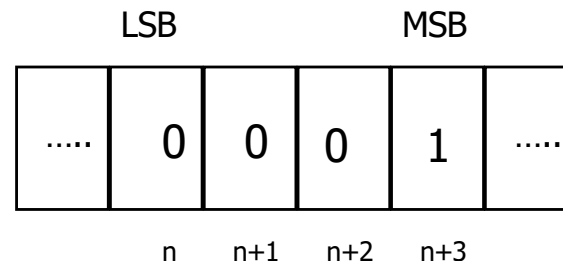
Simple transmissão dos valores armazenados pode levar a resultados errados

“big-endian”



conteúdo da palavra  
= 1

“little-endian”



conteúdo da palavra  
= 1 x 256 x 256 x 256

# 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

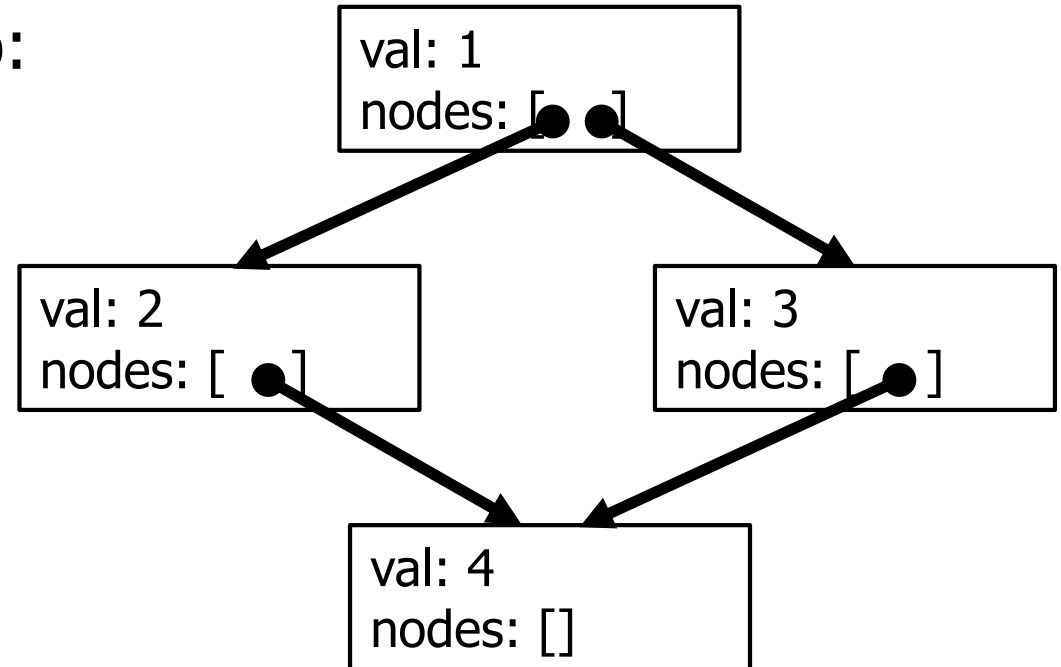
*Marshalling* – processo de codificar do formato interno para o formato rede

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

# PORQUE É QUE O MARSHALLING É COMPLEXO

Considere o seguinte exemplo:

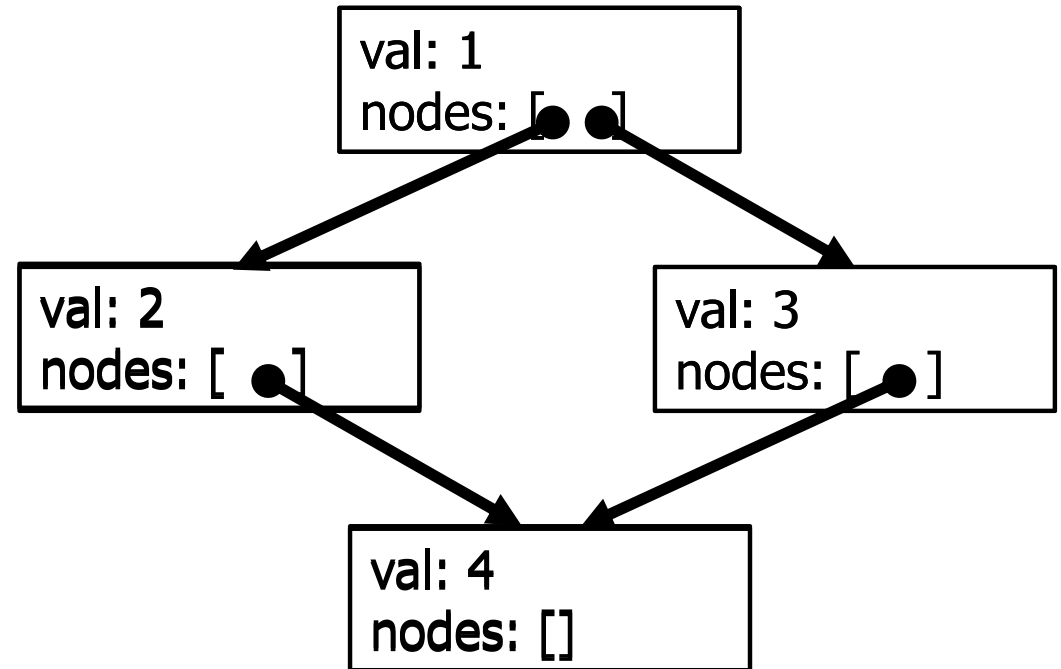
```
class Node {  
    int val;  
    Node[] nodes;  
}
```



# 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>  
  <age>1024</age>
```

```
<?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

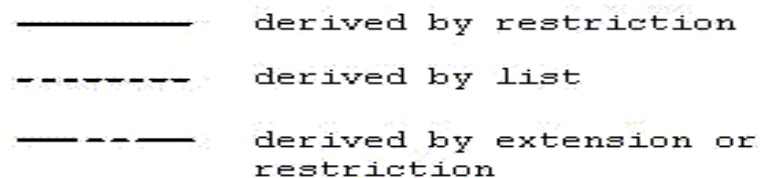
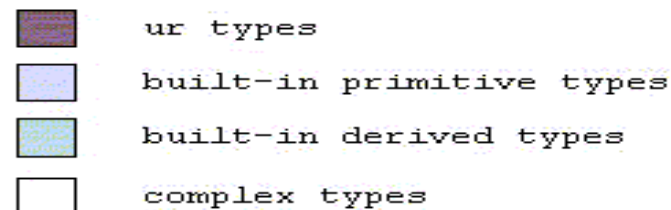
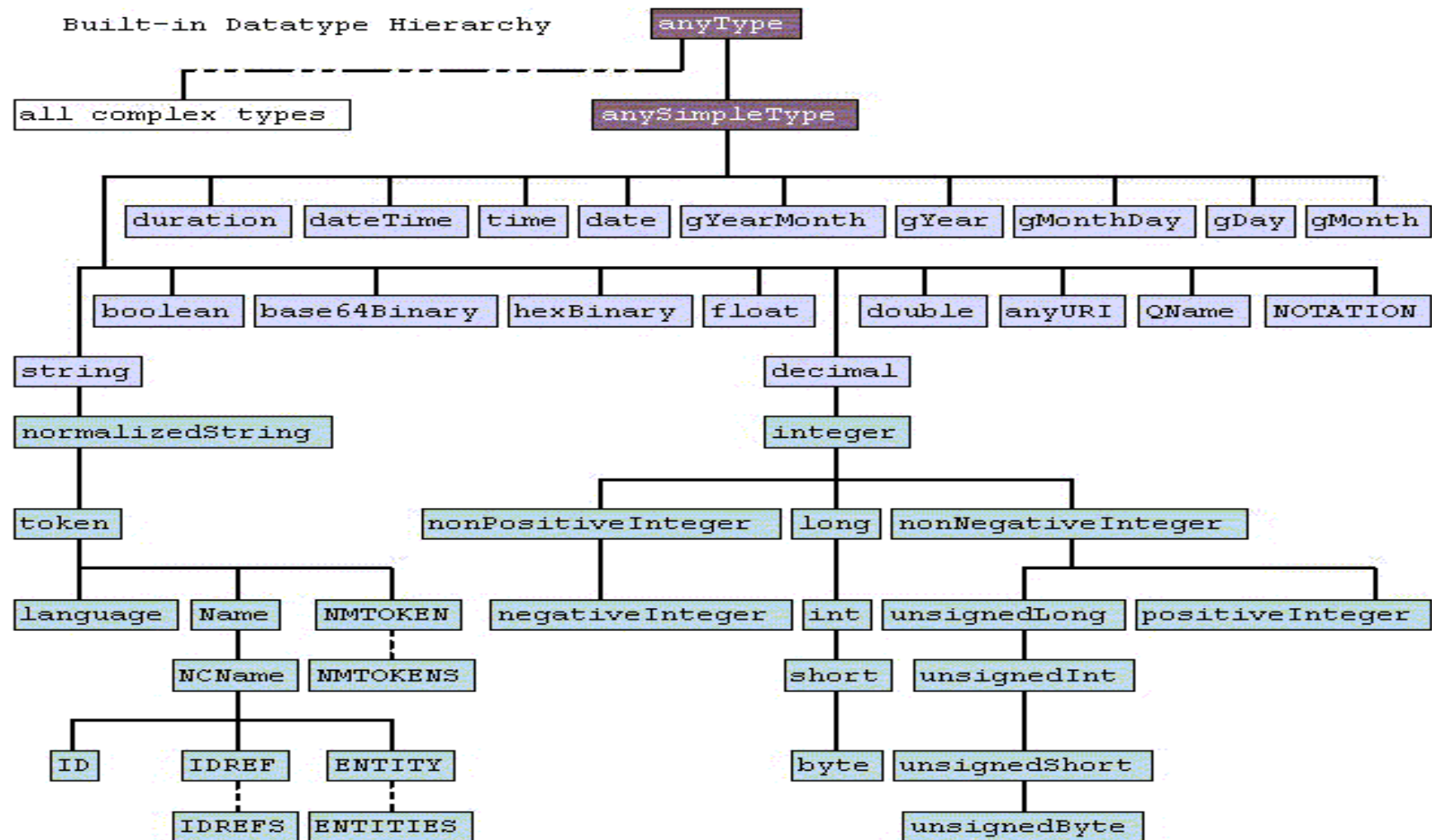
Um XML namespace permite criar espaço de nomes para os nomes dos elementos e atributos usados

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





# 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

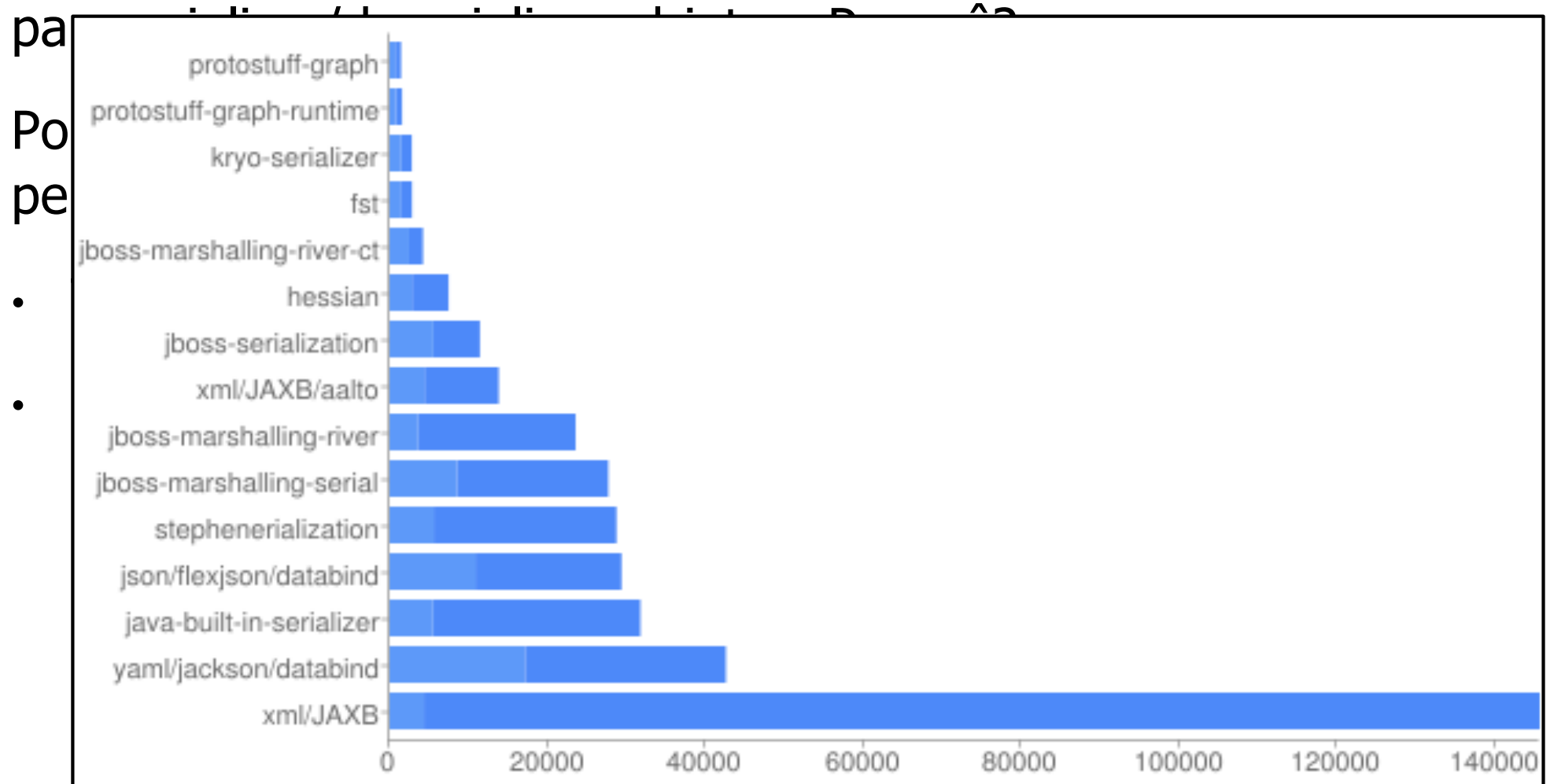
Existe um grande número de soluções para serializar/deserializar objetos. Porquê?

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