

# SISTEMAS DISTRIBUÍDOS

**João Leitão**, Sérgio Duarte, Pedro Camponês

(baseado nos slides de Nuno Preguiça)

Aula 5: Capítulo 4

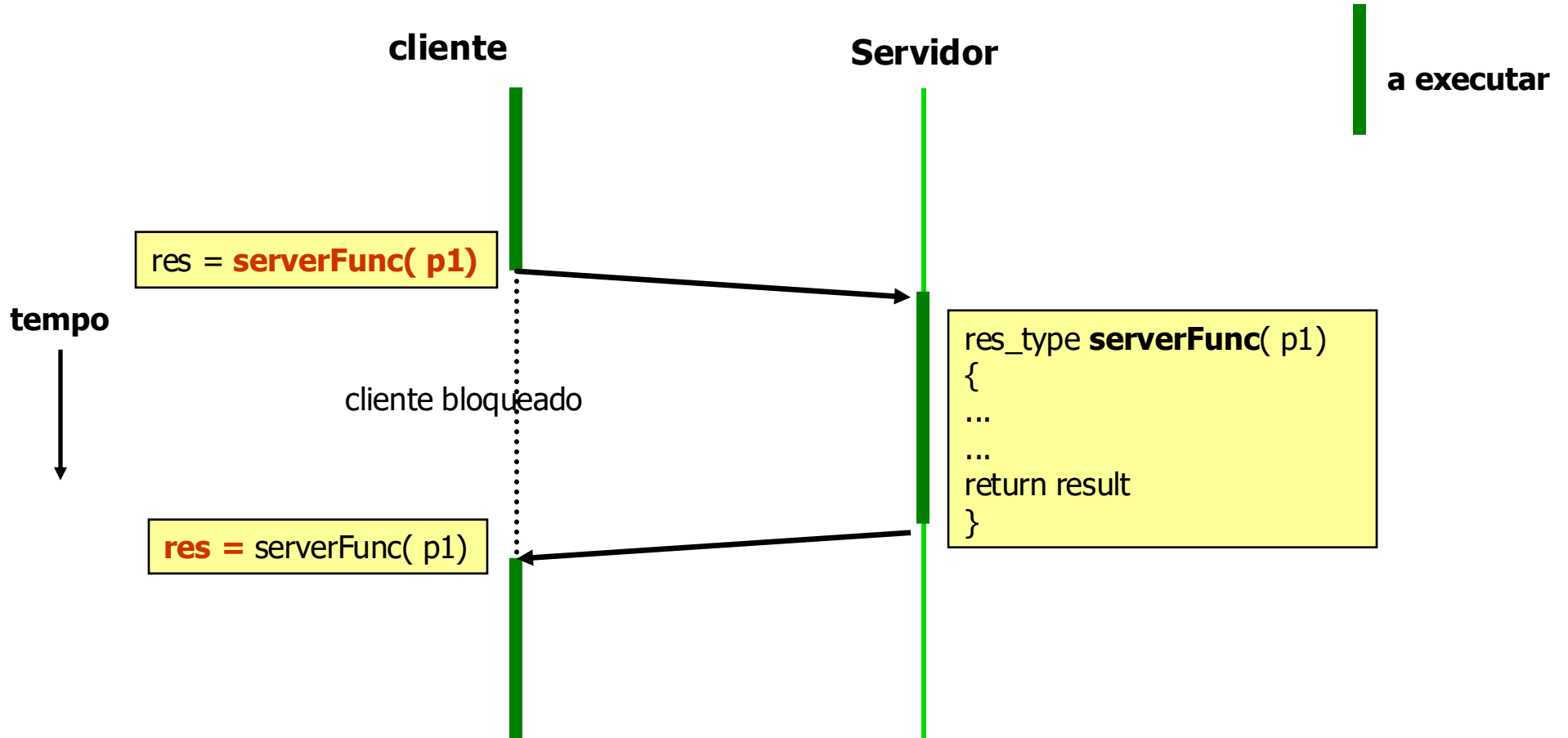
Invocação remota (continuação)

# NA ÚLTIMA AULA

## 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

# 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
```

# 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, .NET remoting

Definir linguagem específica para especificar interfaces dos servidores/objectos remotos

- Ex.: WSDL, gRPC
- Geralmente baseado numa linguagem existente
- Necessidade de mapear o IDL e as linguagens de desenvolvimento dos clientes/servidores



# INTERFACE REMOTA EM JAVA RMI

Interfaces remotos  
estendem **Remote**

```
public interface ContaBancaria
    extends Remote
{
    public void depositar ( float quantia )
        throws RemoteException;

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

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

Interfaces 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);
    }
}
```

Interface definida em C#  
comum

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

# INTERFACE DEFINIDA EM C# PARA .NET REMOTING

```
using System;
namespace IRemoting
{
    public interface ContaBar
    {
        double SaldoActual
        {
            get;
        }
        void depositar ( float q
        void levantar (float qu
    }
}
```

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

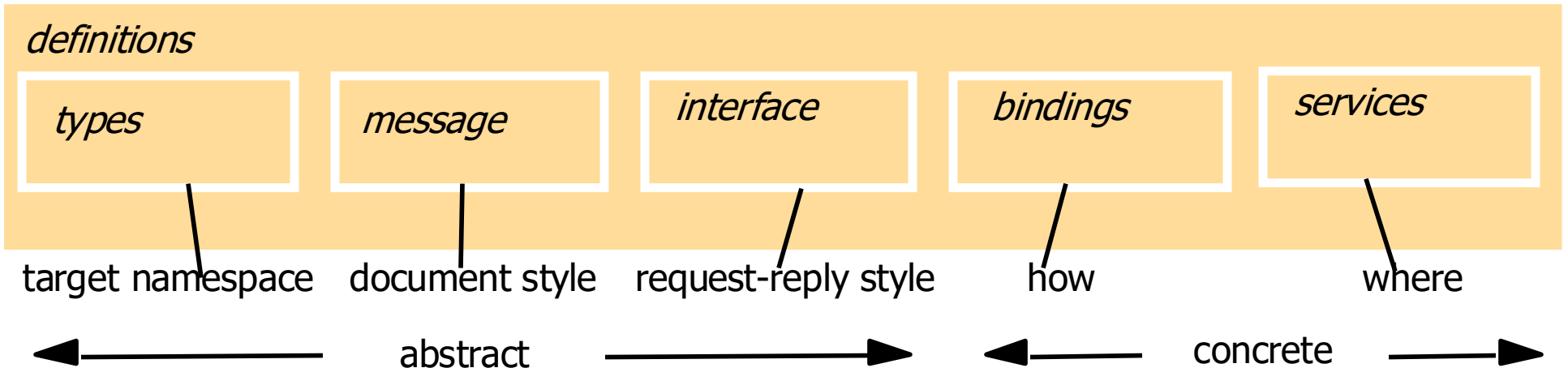
No .NET Remoting não é necessário definir qual a interface remota – esta pode ser inferida a partir da definição do servidor

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="
  </message>
  <message name="SayHelloResponse"
    <part name="greeting" type="
  </message>

  <portType name="Hello_PortType"
    <operation name="sayHello">
      <input message="tns:SayHe
      <output message="tns:SayH
    </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 - ESEMPIO

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

```
<service name="Hello_Service">
  <documentation>WSDL File for Hello_Service</documentation>
  <port binding="tns:Hello_Binding"
    soap:address="http://localhost:8080/soap/servlet/rpcrouter"/>
</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:"
        use="encoded" />
    </input>
    <output>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:examples:"
        use="encoded" />
    </output>
  </operation>
</binding>
```

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

**<definitions>: The HelloService**

**<message>:**

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

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

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

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

# 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.)

Podem definir enumerações.

```
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 createUser( 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 GRPCUsersService()).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();
```

Vamos ver um exemplo de gRPC nas práticas da próxima semana com mais detalhe.

# 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

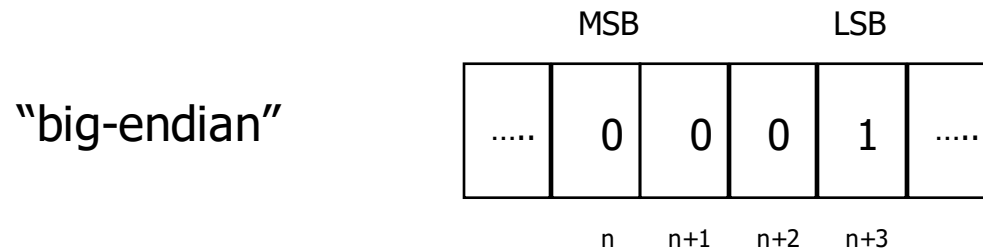
Diferentes sistemas representam os tipos primitivos de formas diferentes

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

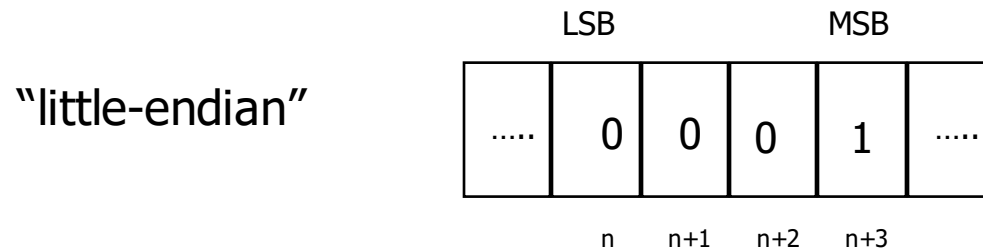
Diferentes representações para números reais – IEEE 754, decimal32, etc.

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

Simple transmissão dos valores armazenados pode levar a resultados errados



conteúdo da palavra  
= 1



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

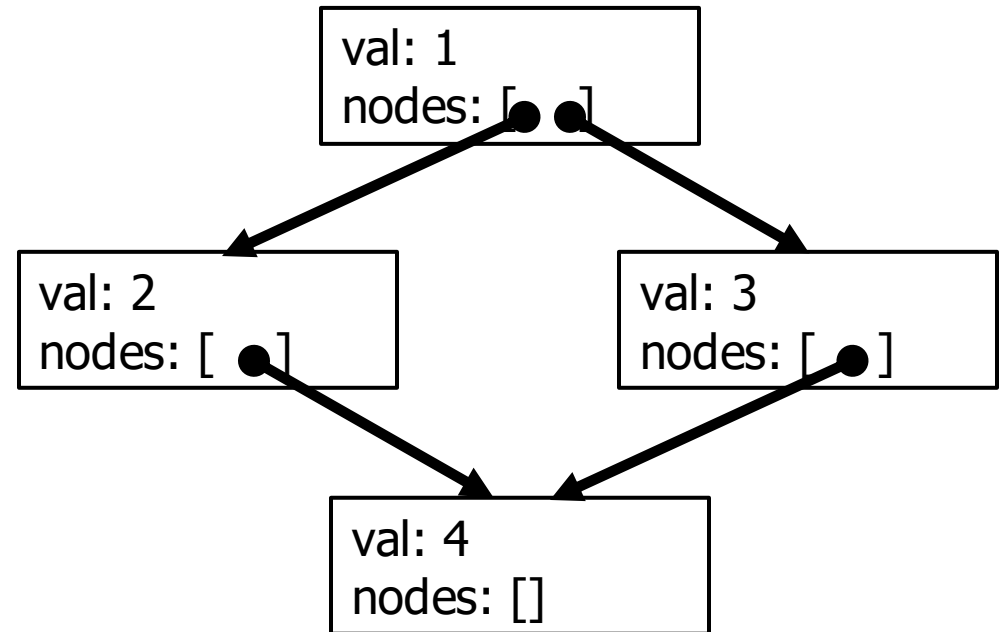
*Marshalling* – processo de codificar do formato interno para o formato rede

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

# PORQUE É QUE O MARSHALLING É COMPLEXO

Pensem no seguinte exemplo:

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

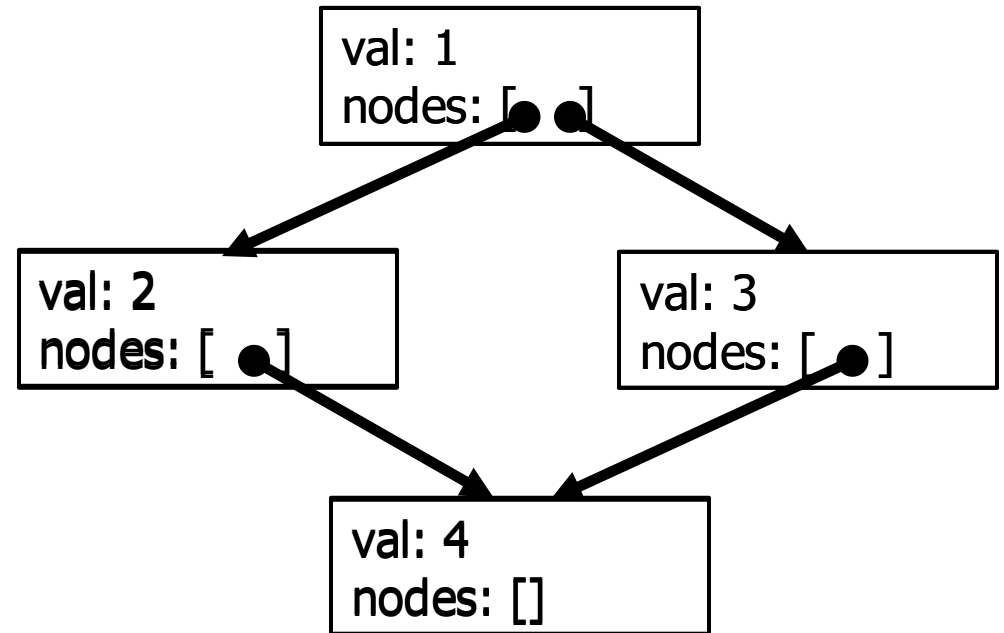




# PORQUE É QUE O MARSHALLING É COMPLEXO?

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

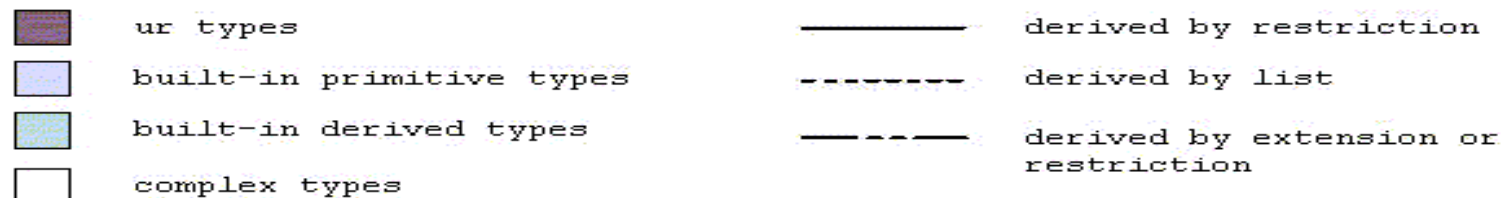
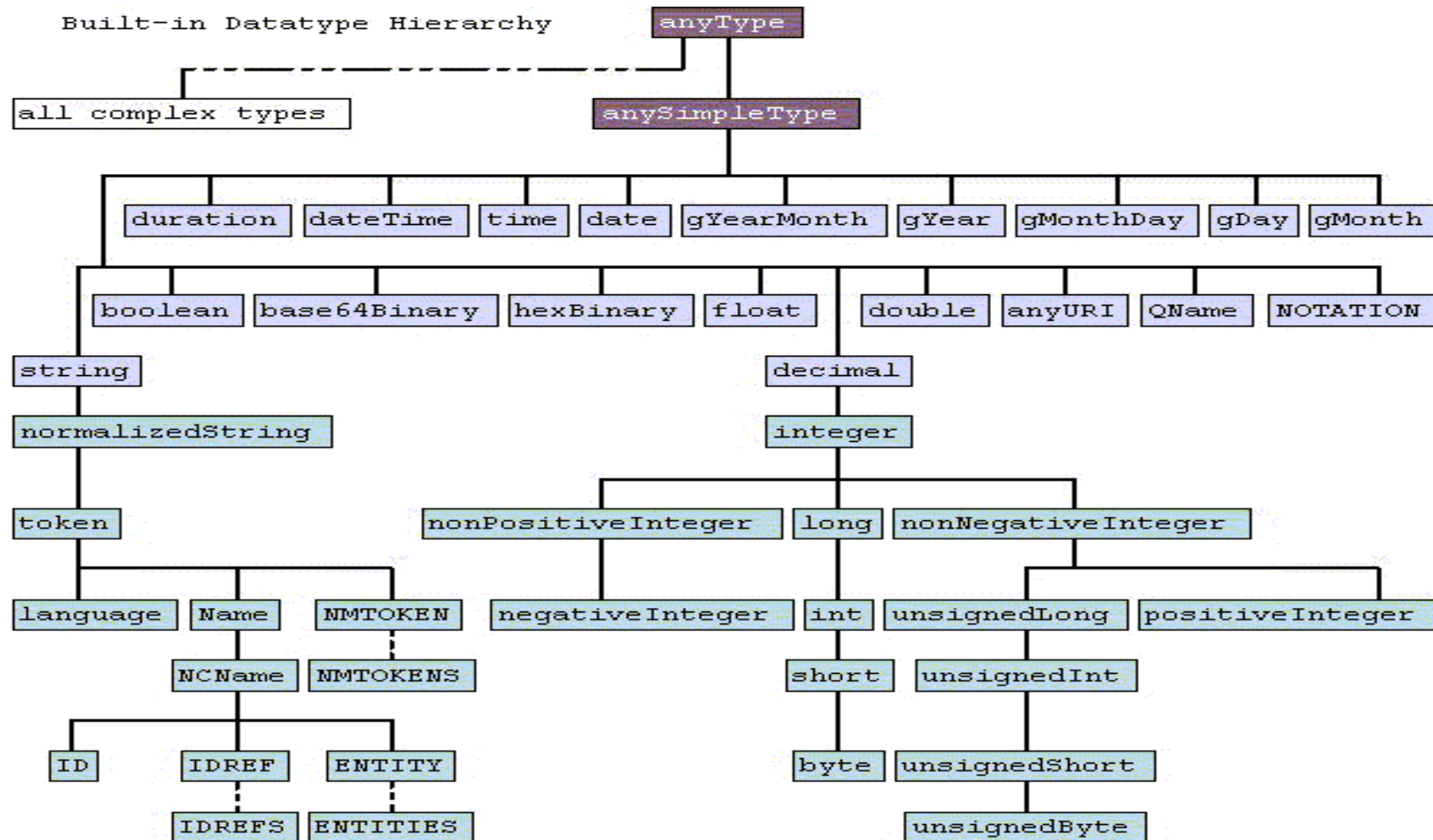
Um XML schema define os elementos aparecer num documento XML

```
<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;  
}
```

Especificação pode ser tão complexa quanto necessária, suporta campos opcionais e obrigatórios (default obrigatório) e campos que se repetem.

Assume que a especificação do formato de comunicação é conhecida por ambos os interlocutores na comunicação.

Dados passam sobre forma binária na rede (mais compacto que formato textual) em que os campos são identificados na serialização pelo código numérico associado a esse campo na especificação.

# 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

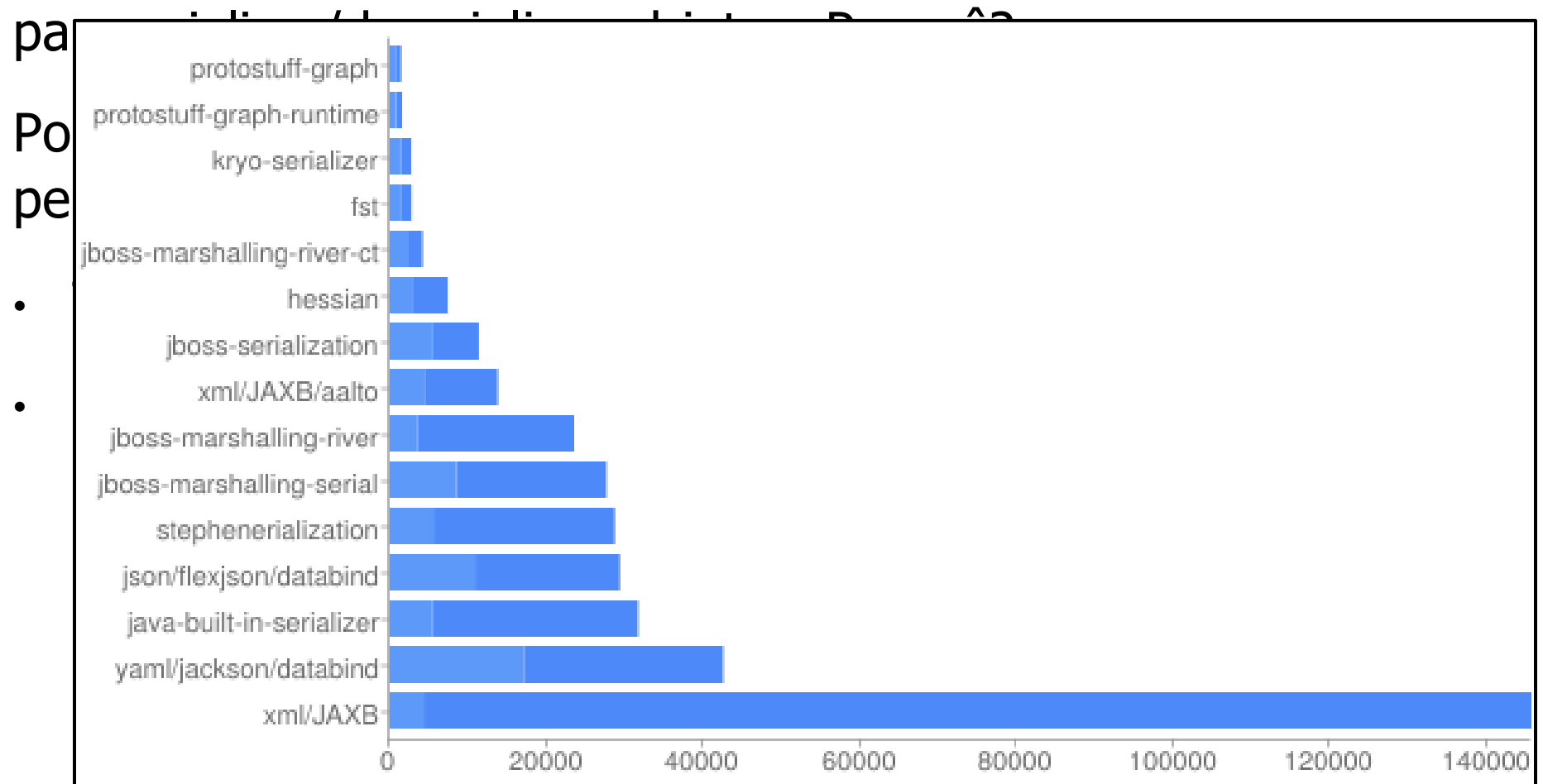
Existe um grande número de soluções para serializar/deserializar objetos. Porquê?

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

# SISTEMAS DISTRIBUÍDOS

Aula 6: Capítulo 4

Invocação remota (Cont.)



# NA ULTIMA AULA...

## 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

Numa linguagem de programação, independentemente dos tipos dos parâmetros, os mesmos podem ser:

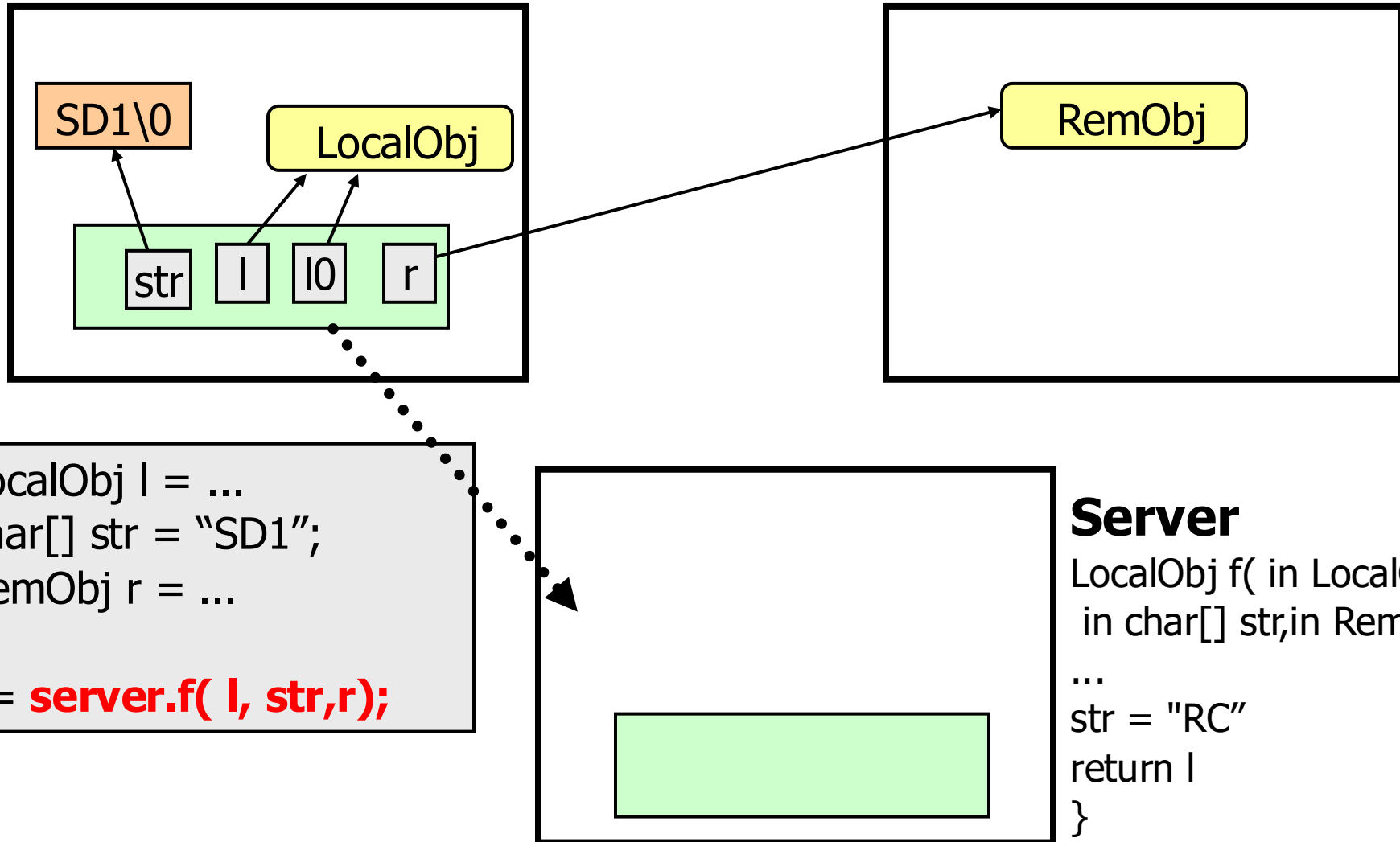
- Passados por valor
- Passados por referência

# MÉTODOS DE PASSAGEM DE PARÂMETROS (CONT.)

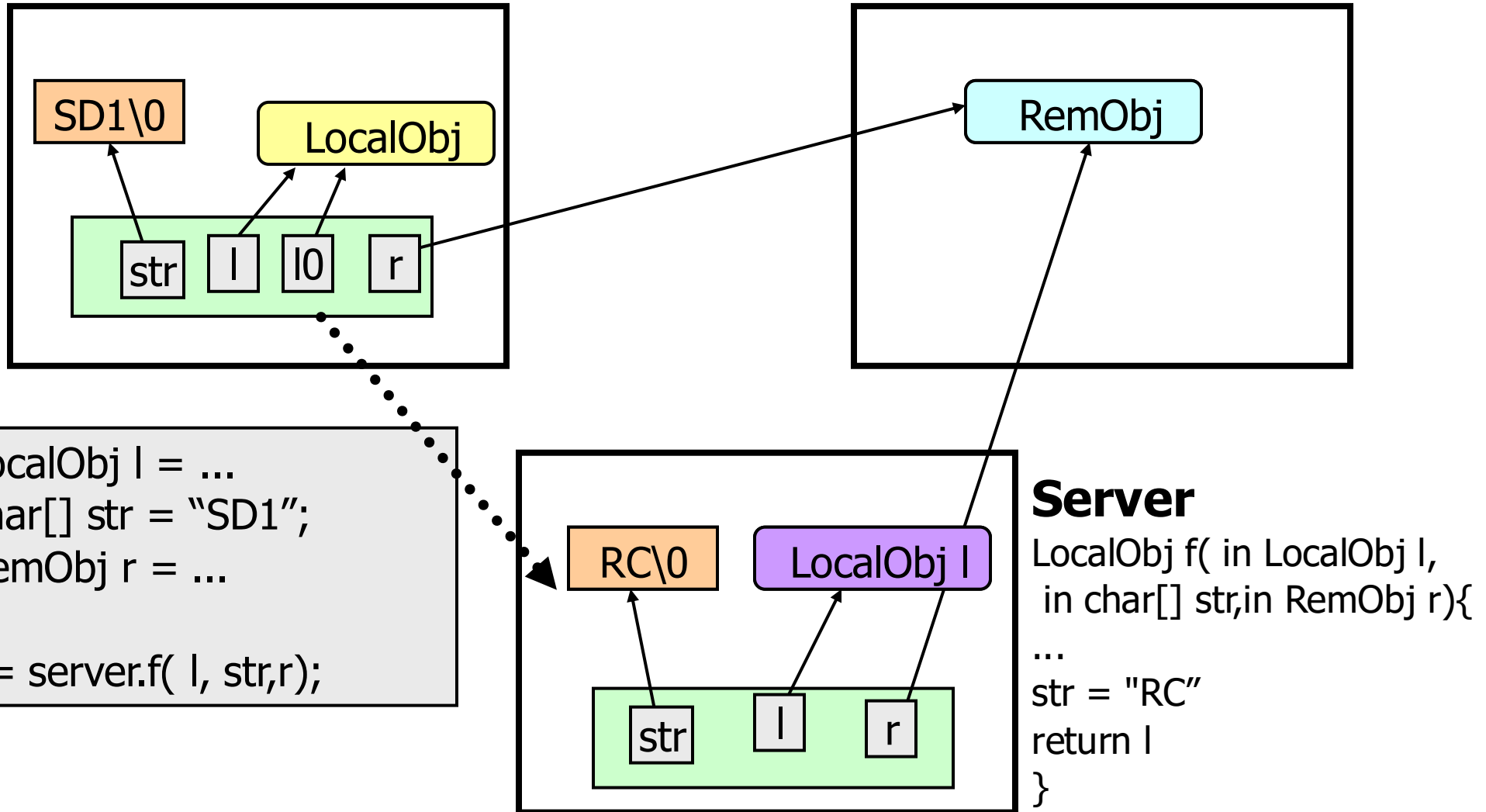
## Aproximação comum nos sistemas de RPC/RMI:

- Passagem por valor para tipos primitivos, arrays, estruturas e objetos
  - Apontadores/referências para arrays, objetos, etc. são seguidas
  - Estado dos objetos 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?

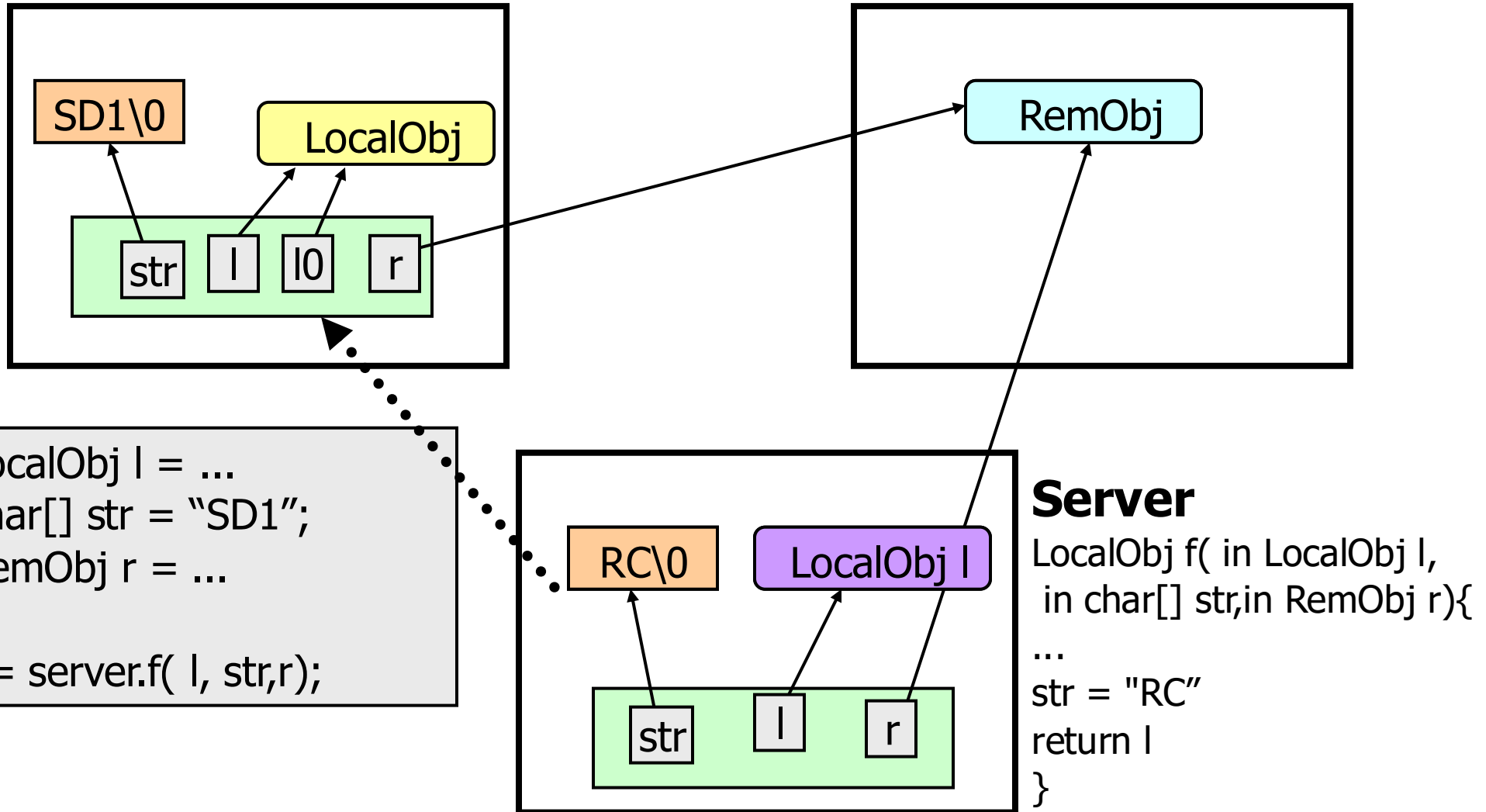
# MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



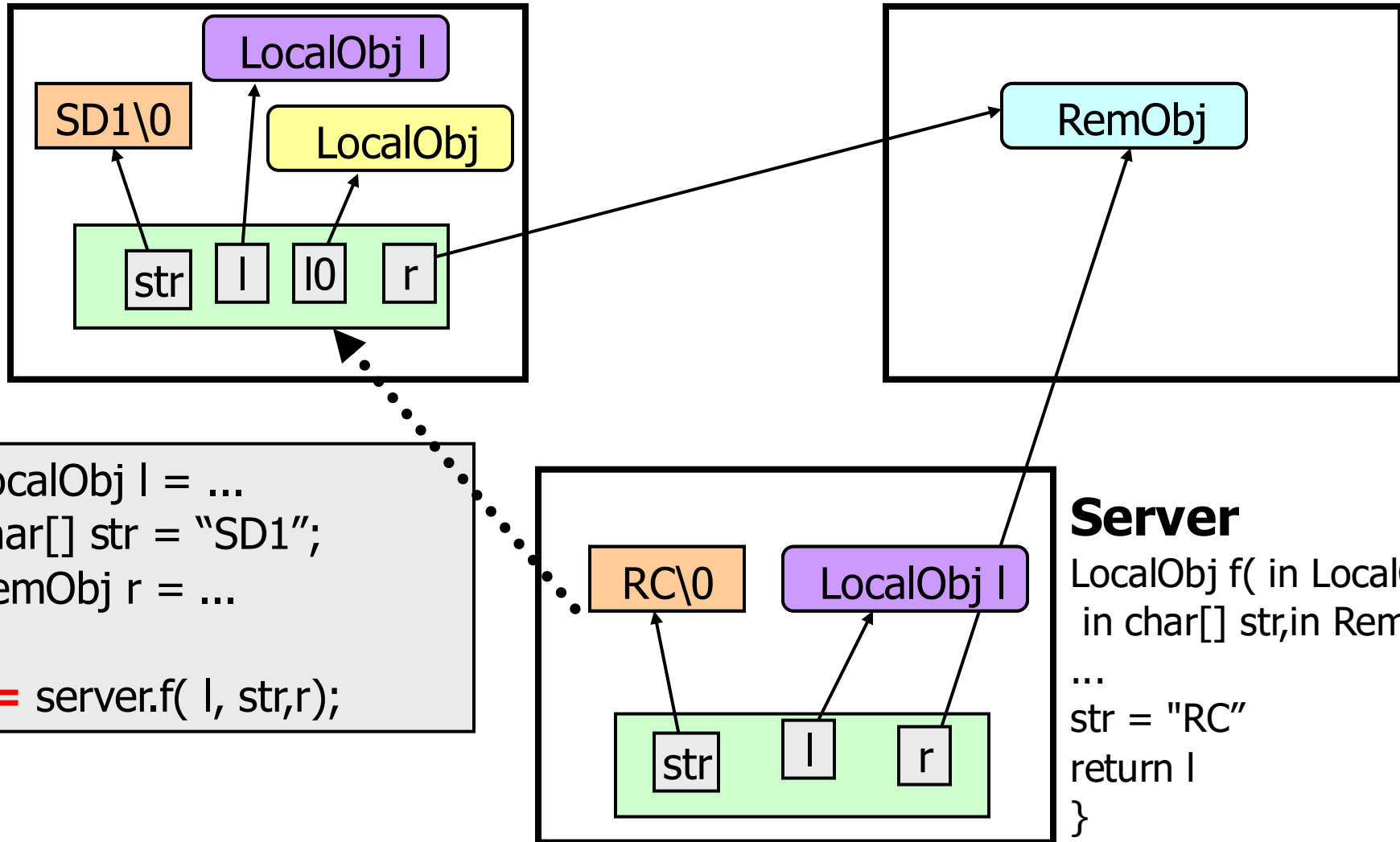
# MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



# MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



# MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



# 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, envia-se uma cópia da referência remota
- Passando como parâmetro/resultado, envia-se uma referência para o objecto remoto e não uma cópia (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



# 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

# 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

# ORGANIZAÇÃO DOS SERVIDORES: JAVA REST

No suporte REST do Java, quando se regista uma classe, são criadas múltiplas instâncias para tratar os vários pedidos.

Pode-se indicar que se pretende apenas uma instância com a anotação `@Singleton`.

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

Alternativamente, pode-se registar um objeto do recurso em vez duma class.

```
30     URI serverURI = URI.create(String.format("http://%s:%s/rest", ip, PORT));
31
32     ResourceConfig config = new ResourceConfig();
33     config.register(new UserResource(domain, serverURI));
34
35     JdkHttpServerFactory.createHttpServer(serverURI, config);
36
```

# 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 objeto

# 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

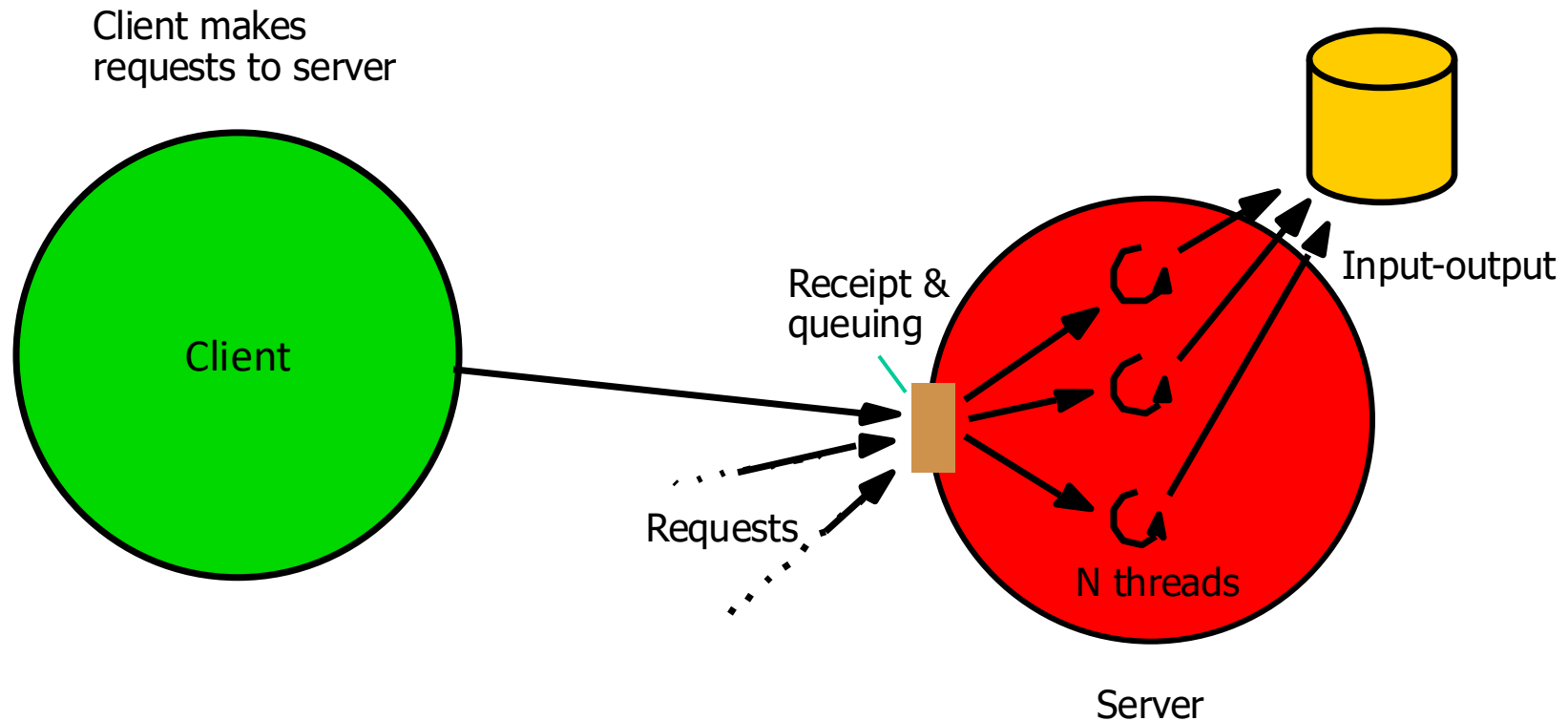
- Exemplos: servidores de bases de dados, de ficheiros, etc. Porquê?
- Exemplo: serviços que chamam outros serviços em ambos os sentidos ( $A \rightarrow B$  e  $B \rightarrow A$ ). Porquê?

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

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?

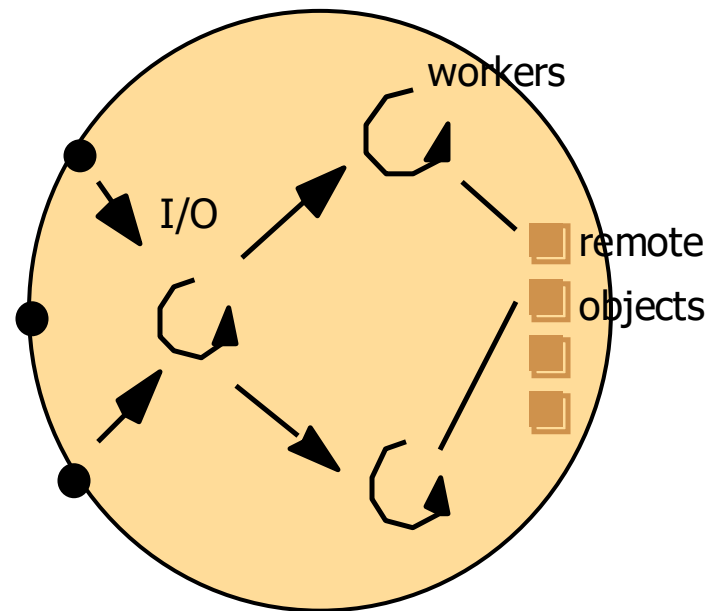
# THREAD POR PEDIDO

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

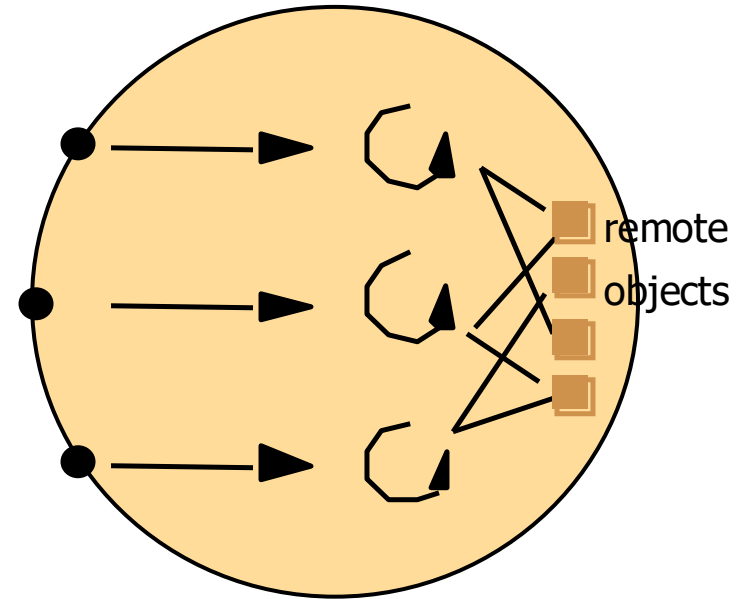
# THREAD POR CONEXÃO

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.

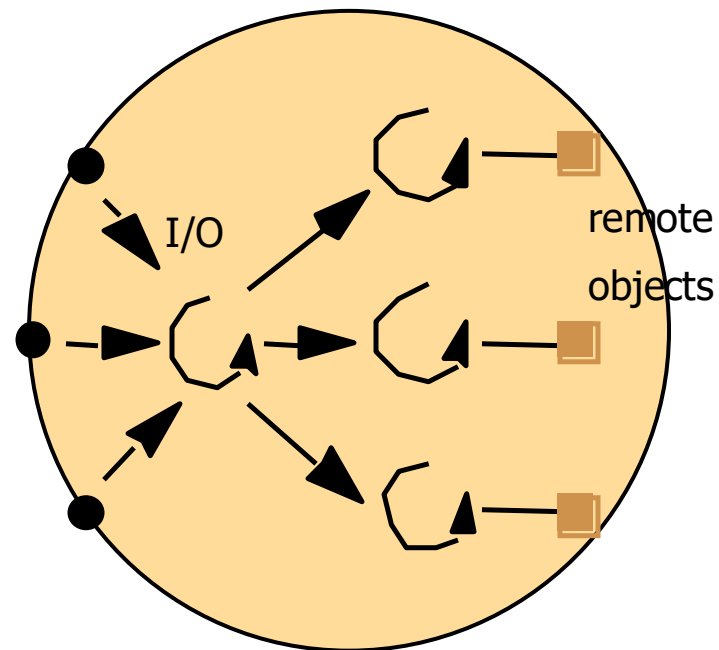
Cada objeto tem um *thread* associada.

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.



c. Thread-per-object

# 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: ACESSO A ESTADO INTERNO

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

- Porquê?

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

# ACESSO CONCORRENTE A ESTRUTURAS DE DADOS: EXEMPLO

@Singleton

```
public class UsersResource implements RestUsers {  
    private final Map<String, User> users = new HashMap<String, User>();
```

@Override

```
public String createUser(User user) {  
    if( users.containsKey(user.getId()))  
        throw new WebApplicationException( Status.CONFLICT );  
    users.put(user.getId(), user);  
    return user.getId();  
}
```

@Override

```
public List<User> searchUsers(String pattern) {  
    List<User> result = new ArrayList<User>();  
    users.values().stream().forEach( u -> { if( u.getFullName().  
        indexOf(pattern) != -1) result.add( new User(u)); });  
    return result;  
}
```

# ACESSO CONCORRENTE A ESTRUTURA DE DADOS

## EXEMPLO

@Singleton

```
public class UsersResource implements RestUsers {
```

```
    private final Map<String, User> users = new HashMap<String, User>();
```

@Override

```
    public String createUser(User user) {
```

```
        if( users.containsKey(user.getId()))
```

```
            throw new WebApplicationException( Status.CONFLICT );
```

```
        users.put(user.getId(), user);
```

```
        return user.getId();
```

```
    }
```

@Override

```
    public List<User> searchUsers()
```

```
    {
```

```
        List<User> result = new ArrayList<>();
```

```
        for( User user : users.values())
```

```
            if( user.getId().equals(id))
```

```
                result.add(user);
```

```
        return result;
```

```
    }
```

Execução concorrente do método **createUser** e **searchUsers** pode levar a exceções, por acesso concorrente ao mapa values.

```
java.util.ConcurrentModificationException
    at java.base/java.util.HashMap$ValueSpliterator.forEachRemaining(HashMap.java:1622)
    at java.base/java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:658)
    at sd2021.aula2.server.resources.UsersResource.searchUsers(UsersResource.java:45)
    at jdk.internal.reflect.GeneratedMethodAccessor5.invoke(Unknown Source)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:564)
    at org.glassfish.jersey.server.model.internal.ResourceMethodInvocationHandler.invoke(ResourceMethodInvocationHandler.java:226)
    at org.glassfish.jersey.server.model.internal.AbstractJavaResourceMethodDispatcher$1.invoke(AbstractJavaResourceMethodDispatcher.java:133)
    at org.glassfish.jersey.server.model.internal.AbstractJavaResourceMethodDispatcher.invoke(AbstractJavaResourceMethodDispatcher.java:161)
    at org.glassfish.jersey.server.model.internal.JavaResourceMethodDispatcherProxy.processAndRouteRequest(JavaResourceMethodDispatcherProxy.java:103)
    at org.glassfish.jersey.server.model.internal.AbstractJavaResourceMethodDispatcher$1.invoke(AbstractJavaResourceMethodDispatcher.java:133)
    at org.glassfish.jersey.server.model.internal.AbstractJavaResourceMethodDispatcher.invoke(AbstractJavaResourceMethodDispatcher.java:161)
    at org.glassfish.jersey.server.model.ResourceMethodInvoker.invoke(ResourceMethodInvoker.java:439)
    at org.glassfish.jersey.server.model.ResourceMethodInvoker.apply(ResourceMethodInvoker.java:491)
    at org.glassfish.jersey.server.model.ResourceMethodInvoker.apply(ResourceMethodInvoker.java:491)
    at org.glassfish.jersey.server.ServerRuntime$1.run(ServerRuntime.java:255)
```



# ACESSO CONCORRENTE A ESTRUTURAS DE DADOS: EXEMPLO

@Singleton

```
public class UsersResource implements RestUsers {  
    private final Map<String, User> users = new HashMap<String, User>();
```

@Override

```
public String createUser(User user) {  
    if( users.containsKey(user.getUserId()))  
        throw new WebApplicationException( Status.CONFLICT );  
    users.put(user.getUserId(), user);  
    return user.getUserId();  
}
```

@Override

```
public List<User> search  
    List<User> result = new ArrayList<>();  
    users.values().stream()  
        .filter(user -> user.getUserId().indexOf(pattern) != -1)  
        .forEach(result::add);  
    return result;  
}
```

Execução concorrente do método **createUser** por dois threads, com users diferentes mas o mesmo userId pode levar a que ambos os threads verifiquem que o userId não existe e pensem que inseriram o utilizador, quando apenas o último ficará guardado.

Poderá existir outro problema?

# TÉCNICAS DE CONTROLO DE CONCORRÊNCIA

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

Locks

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

Transações

# MÉTODOS SYNCHRONIZED

@Singleton

```
public class UsersResource implements RestUsers {  
    private final Map<String, User> users = new HashMap<String, User>();
```

@Override

```
public synchronized String createUser(User user) {  
    if( users.containsKey(user.getId()))  
        throw new WebApplicationException( Status.CONFLICT );  
    users.put(user.getId(), user);  
    return user.getId();  
}
```

@Override

```
public synchronized List<User> searchUsers(String pattern) {  
    List<User> result = new ArrayList<User>();  
    users.values().stream().forEach( u -> { if( u.getFullName().  
        indexOf(pattern) != -1) result.add( new User(u)); });  
    return result;  
}
```

# MÉTODOS SYNCHRONIZED

@Singleton

```
public class UsersResource implements RestUsers {  
    private final Map<String, User> users = new HashMap<>();
```

@Override

```
public synchronized String createUser(User user) {  
    if( users.containsKey(user.getId()))  
        throw new WebApplicationException( Status.CONFLICT );  
    users.put(user.getId(), user);  
    return user.getId();  
}
```

@Override

```
public synchronized List<User> searchUsers(String pattern) {  
    List<User> result = new ArrayList<User>();  
    users.values().stream().forEach( u -> { if(  
        indexOf(pattern) != -1) result.add(u); } );  
    return result;  
}
```

Num dado momento, apenas um *thread* pode executar nos métodos marcados como synchronized. Se todos os métodos forem synchronized, apenas um thread modifica o servidor em cada momento, resolvendo os problema anteriores.

## Potenciais problemas:

- Se um método for lento, limita o desempenho do servidor;
- Se um método invocar outro servidor pode, no limite, levar a problemas de deadlock no caso de pedidos cruzados entre servidores (ou ciclos de invocação).

# BLOCOS SYNCHRONIZED

@Singleton

```
public class UsersResource implements RestUsers {  
    private final Map<String, User> users = new HashMap<String, User>();
```

@Override

```
    public String createUser(User user) {  
        if (user.getUserId() == null || user.getPassword() == null ||  
            user.getFullName() == null || user.getEmail() == null)  
            throw new WebApplicationException( Status.BAD_REQUEST );  
        synchronized( users ) {  
            if ( users.containsKey(user.getUserId()) )  
                throw new WebApplicationException( Status.CONFLICT );  
            users.put(user.getUserId(), user);  
        }  
        return user.getUserId();  
    }  
}
```

É possível reduzir a zona que executa em exclusão mútua usando um bloco *synchronized* apenas para proteger as operações que podem causar problema. Neste exemplo, as verificações se o objeto User está correto não precisam de ser protegidas – apenas o acesso a variáveis que podem ser acedidas concorrentemente precisa.

```
private final Map<String, User> users = new HashMap<String, User>();

@Override
public String createUser(User user) {

    if(user.getUserId() == null || user.getPassword() == null ||

        user.getFullName() == null || user.getEmail() == null)

        throw new WebApplicationException( Status.BAD_REQUEST );

    synchronized( users ) {

        if( users.containsKey(user.getUserId()))

            throw new WebApplicationException( Status.CONFLICT );

        users.put(user.getUserId(), user);

    }

    return user.getUserId();

}
```

Em cada momento, apenas um thread pode aceder a um bloco synchronized relativo à variável X.

# BLOCOS SYNCHRONIZED

```
public class Example {  
    private final Queue<String> strs = new LinkedList<String>();  
  
    public void addOne(String s) {  
        synchronized( strs) {  
            strs.put( s);  
            try {  
                strs.notifyAll();  
            } catch( InterruptedException e) { }  
        }  
    }  
  
    public String getOne() {  
        synchronized( strs) {  
            while (strs.peek() == null)  
                try {  
                    strs.wait();  
                } catch( Exception e) { }  
            return strs.poll();  
        }  
    }  
}
```

# BLOCOS SYNCHRON

```
public class Example {  
    private final Queue<String> strs;  
  
    public void addOne(String s) {  
        synchronized( strs) {  
            strs.put( s);  
            try {  
                strs.notifyAll();  
            } catch( InterruptedException e) { }  
        }  
    }  
  
    public String getOne() {  
        synchronized( strs) {  
            while (strs.peek() == null)  
                try {  
                    strs.wait();  
                } catch( Exception e) { }  
            return strs.poll();  
        }  
    }  
}
```

Os monitores permitem ainda que um *thread* se bloqueie no meio dum bloco synchronized, permitindo a outros *threads* entrarem na região crítica e mais tarde desbloquear o *thread* bloqueado – esta funcionalidade é usada, por exemplo, quando se quer esperar que alguma condição ocorra.



# TÉCNICAS DE CONTROLO DE CONCORRÊNCIA

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

Locks

- Estudado em FSO

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

Transações

# ESTRUTURAS DE DADOS CONCORRENTES

```
public class Example {  
    private final Queue<String> strs = new LinkedBlockingQueue<String>();  
  
    public void addOne(String s) {  
        strs.put( s);  
    }  
    public String getOne() {  
        return strs.take();  
    }  
}
```

O pacote **java.util.concurrent** tem implementações que permitem acesso concorrente, incluindo a iteradores. Não resolve todos os problemas de concorrência.

# ESTRUTURAS DE DADOS CONCORRENTES (CONT.)

@Singleton

```
public class UsersResource implements RestUsers {
```

```
    private final Map<String, User> users = new ConcurrentHashMap<String, User>();
```

@Override

```
    public String createUser(User user) {
```

```
        if( users.containsKey(user.getId()))
```

```
            throw new WebApplicationException( Status.CONFLICT );
```

```
        users.put(user.getId(), user);
```

```
        return user.getId();
```

```
    }
```

@Override

```
    public List<User> searchUsers(String pattern) {
```

```
        List<User> result = new ArrayList<User>();
```

```
        users.values().stream().forEach( u -> { if( u.getFullName().
```

```
            indexOf(pattern) != -1) result.add( new User(u)); });
```

```
        return result;
```

```
    }
```

## **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,User> map = ...
```

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

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

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

# TÉCNICAS DE CONTROLO DE CONCORRÊNCIA

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

Locks

- Estudado em FSO

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

Transações

- É comum os servidores aplicativos serem *stateless* e todo o estado persistente estar numa base de dados.
- Controlo de concorrência pode ser delegado para a base de dados – operação da aplicação origina transação na base de dados - controlo de concorrência efetuado na base de dados.

# CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: DEADLOCKS

Quando se utiliza um mecanismo de controlo de concorrência baseado em *locks* – e.g. monitores , *locks* – é necessário lidar com potenciais problemas de deadlocks.

Os deadlock podem surgir dentro dum servidor ou entre servidores.

# CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: DEADLOCKS NUM SERVIDOR (EXEMPLO)

```
private final User user1 = ...  
private final User user2 = ...
```

```
public String copy1To2 () {  
    synchronized(user1) {  
        synchronized(user2) {  
            User aux = user1;  
            user1 = user2;  
            user2 = aux;  
        }  
    }  
}
```

```
public String copy2To1 () {  
    synchronized(user2) {  
        synchronized(user1) {  
            User aux = user2;  
            user2 = user1;  
            user1 = aux;  
        }  
    }  
}
```

# CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: DEADLOCKS

Para evitar deadlocks num servidor, podem-se usar diferentes aproximações:

1. Ter apenas um lock;
2. Obter locks sempre por ordem – e.g. se tivermos os locks  $l_1, l_2, l_3, \dots$ , uma operação obtém os locks que precisa por ordem, i.e., após obter o lock  $l_i$ , nunca se obtém o lock  $l_j$ , tal que  $j < i$ .



# CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: DEADLOCKS DISTRIBUÍDOS

Os deadlocks podem surgir entre servidores.

```
public class UsersResource
    implements Users {

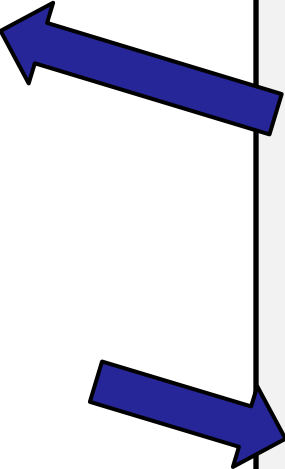
    @Override
    public synchronized User getUser(...) {
        ...
    }

    @Override
    public synchronized User deleteUser(...) {
        ...
        shorts.deleteUserShorts(...);
    }
}
```

```
public class ShortsResource
    implements Shorts {

    @Override
    public synchronized FileInfo newShort (...) {
        ...
        User u = users.getUser( ...);
    }

    @Override
    public synchronized User deleteUserShorts(..){
        ...
    }
}
```



# CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: DEADLOCKS DISTRIBUÍDOS

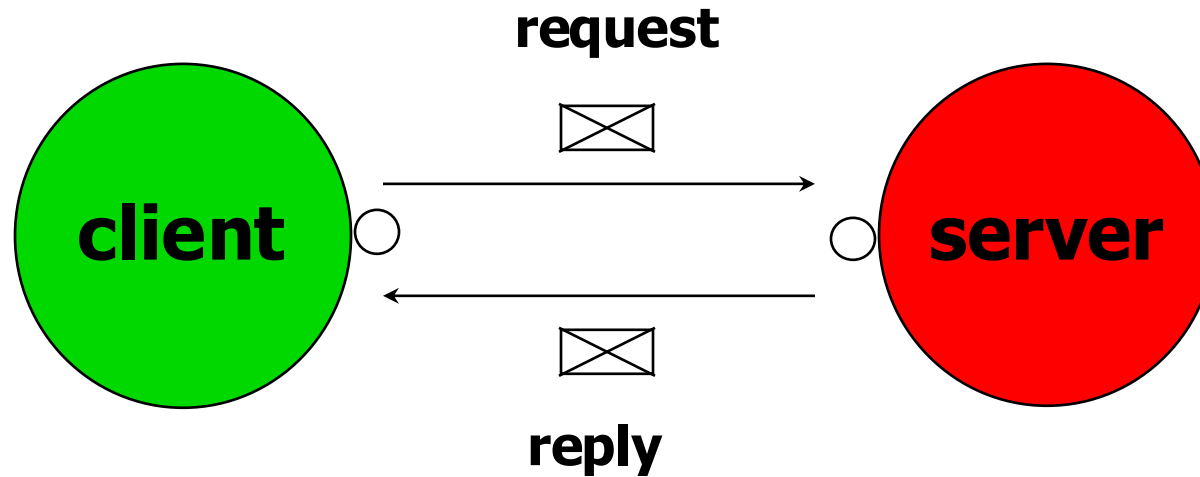
Para evitar deadlocks distribuídos, deve-se evitar fazer uma invocação remota enquanto se tem um *lock*.

# 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
- 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

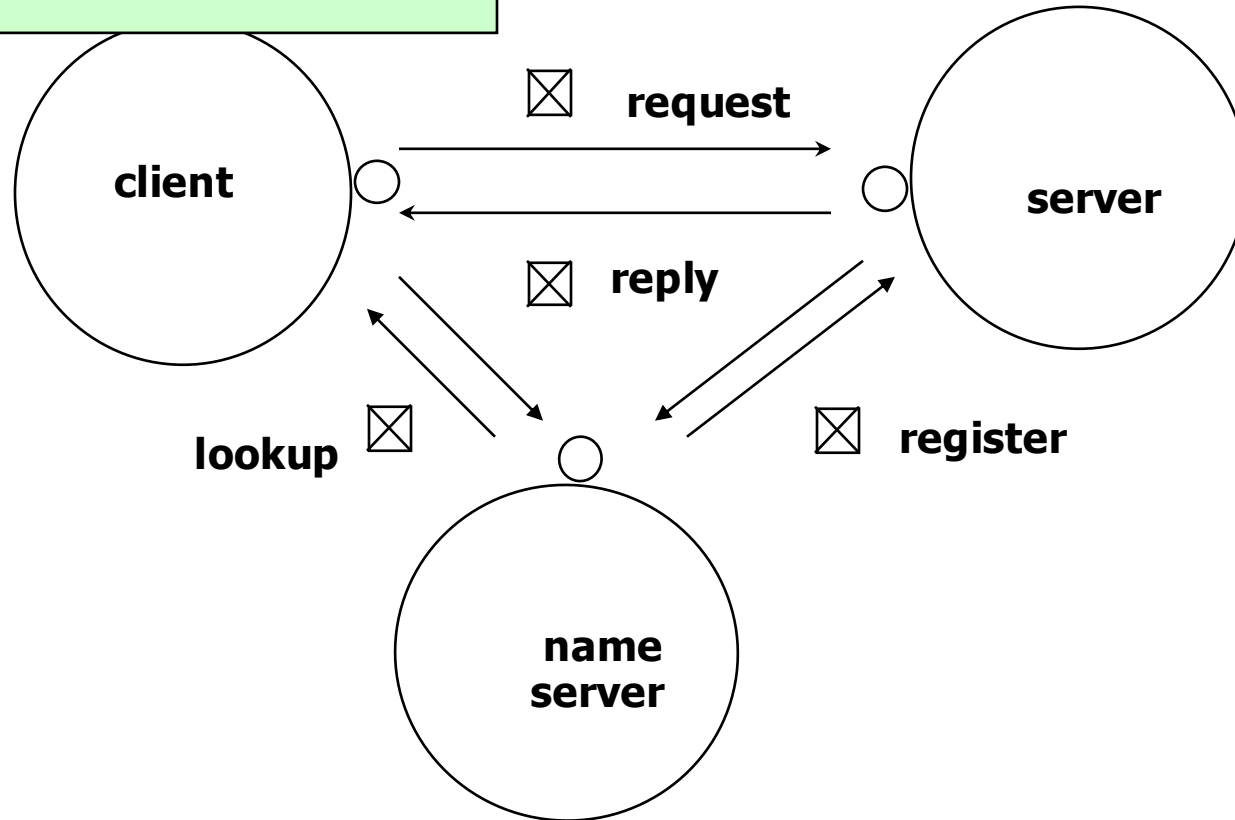
# USAR UM SERVIÇO (*BINDING, NAMING OR TRADING*)

## Java RMI

```
IServer s = Naming.lookup ( "service")  
res = s.fun( obj)
```

## Java RMI

```
Naming.rebind( "service", this)
```



# 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?

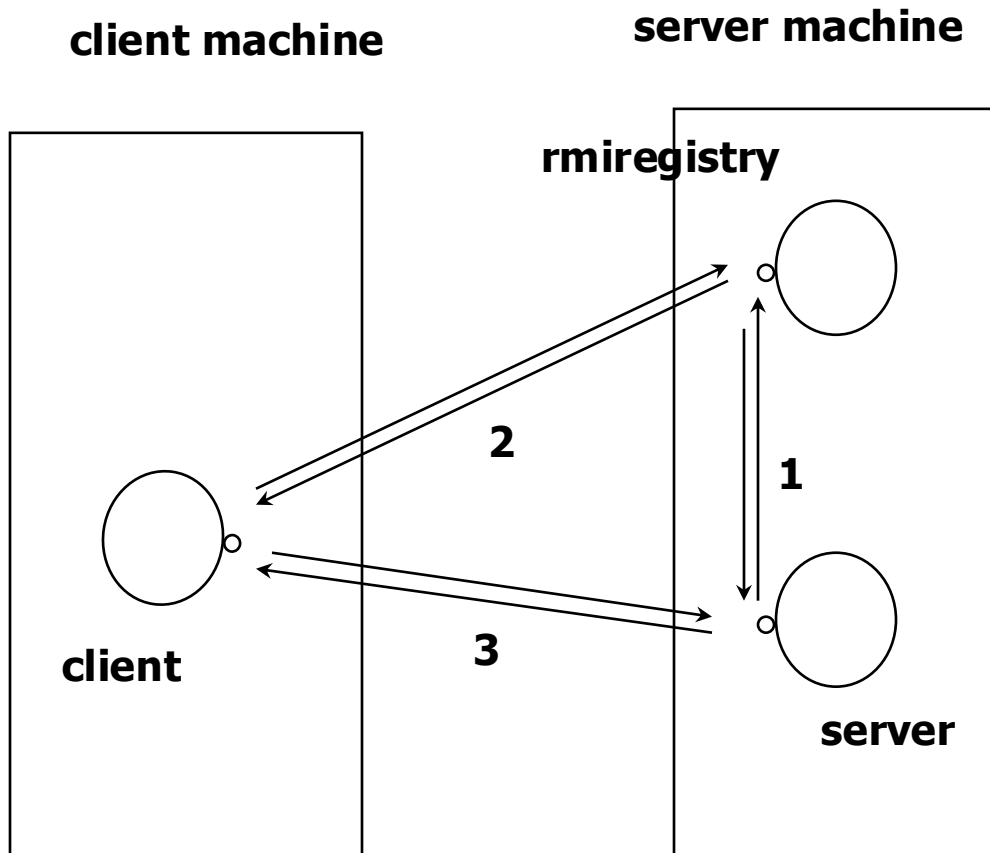
- Nome do objecto indica máquina em que está o servidor de nomes
- Servidor de nomes descoberto por multicast/broadcast

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
- 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

# 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