Chamadas de Procedimentos Remotos (RPC)

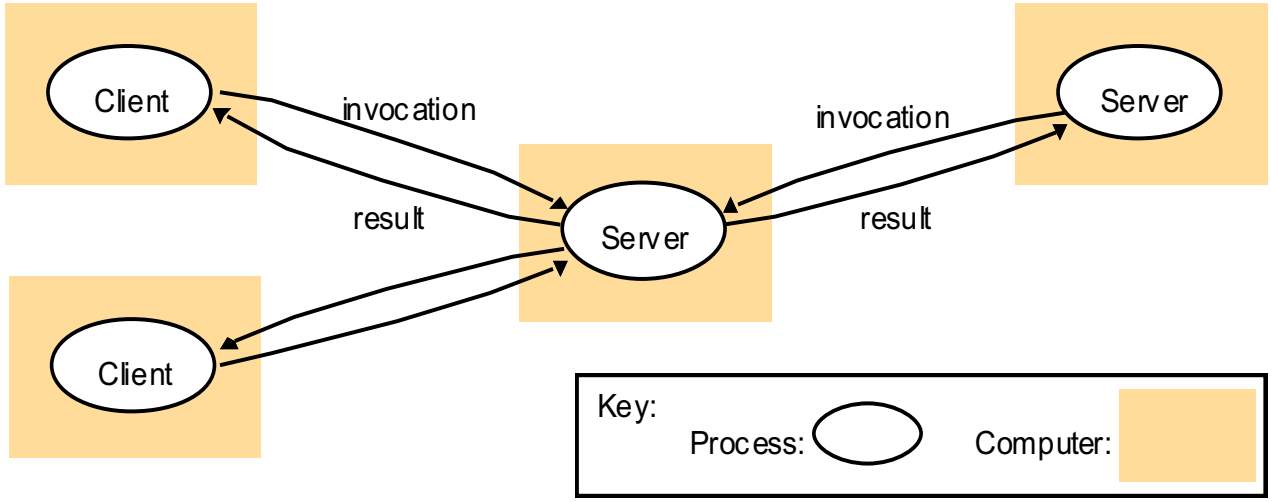
Desafio#1: simplificar a programação de um SD

- Para criar uma aplicação distribuída é necessário um **modelo de programação**

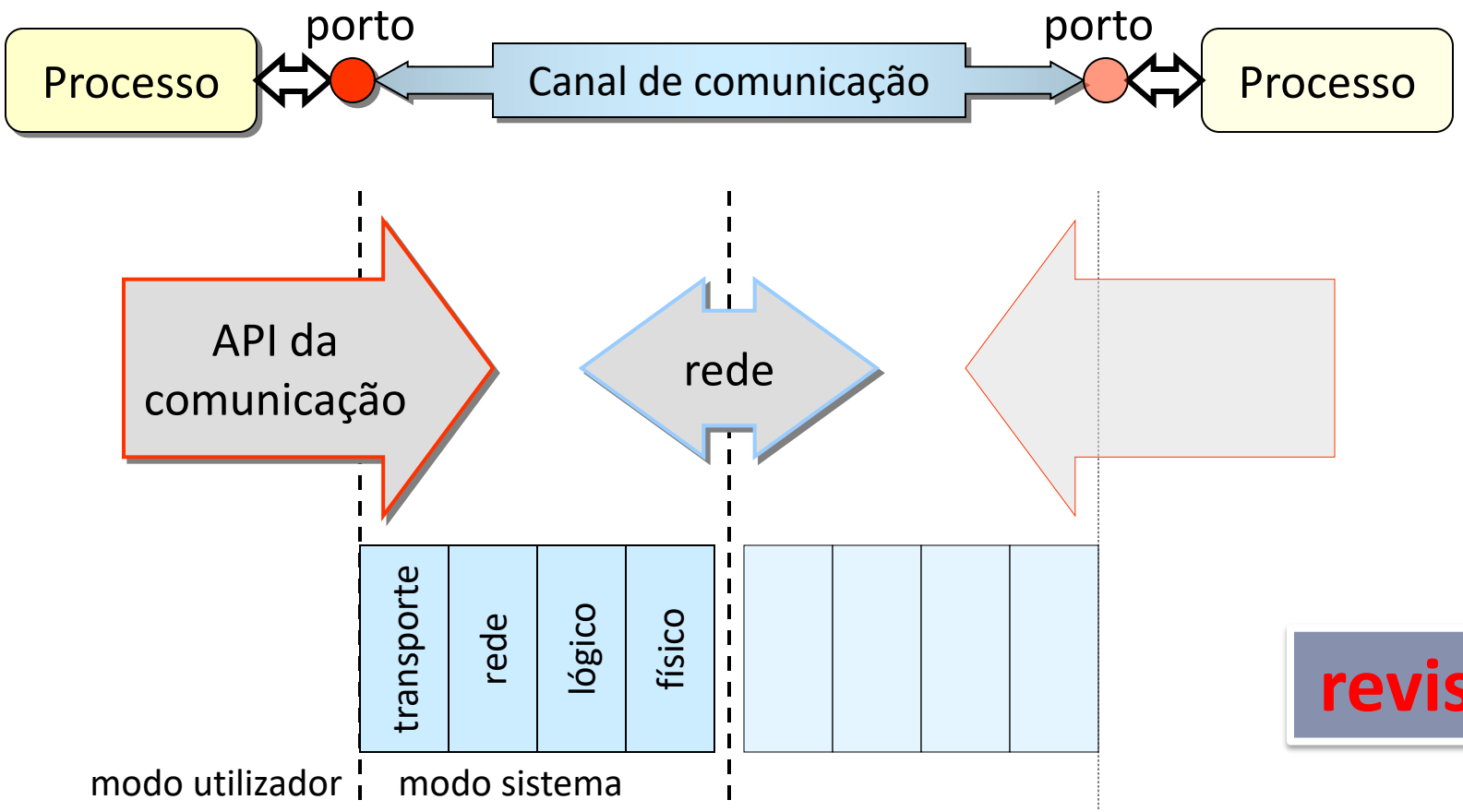
- *Já programaram aplicações distribuídas?*

Arquitetura cliente-servidor

- Servidores mantêm recursos e servem pedidos de operações sobre esses recursos
- Servidores podem ser clientes de outros servidores
- **Simples** e permite distribuir sistemas centralizados muito diretamente
- *O desafio é a **escalabilidade**: limitado pela capacidade do servidor e pela rede que o liga aos clientes*

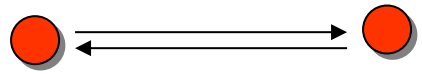


Recordando o modelo computacional do projeto de redes

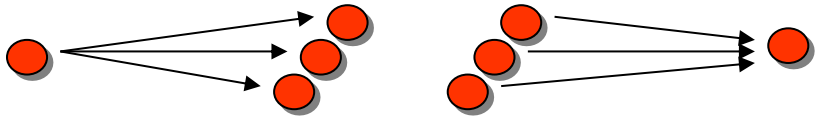


Caracterização do canal de comunicação

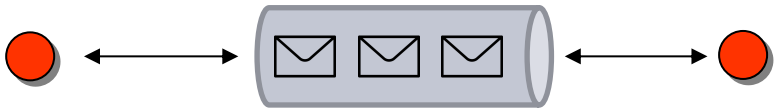
- Com ligação
  - Normalmente serve 2 interlocutores
  - Normalmente fiável, bidireccional e garante sequencialidade



- Sem ligação
  - Pode servir mais de 2 interlocutores
  - Normalmente não fiável: perdas, duplicação, reordenação



- Com capacidade de armazenamento em *fila de mensagens*
  - Normalmente com entrega fiável das mensagens

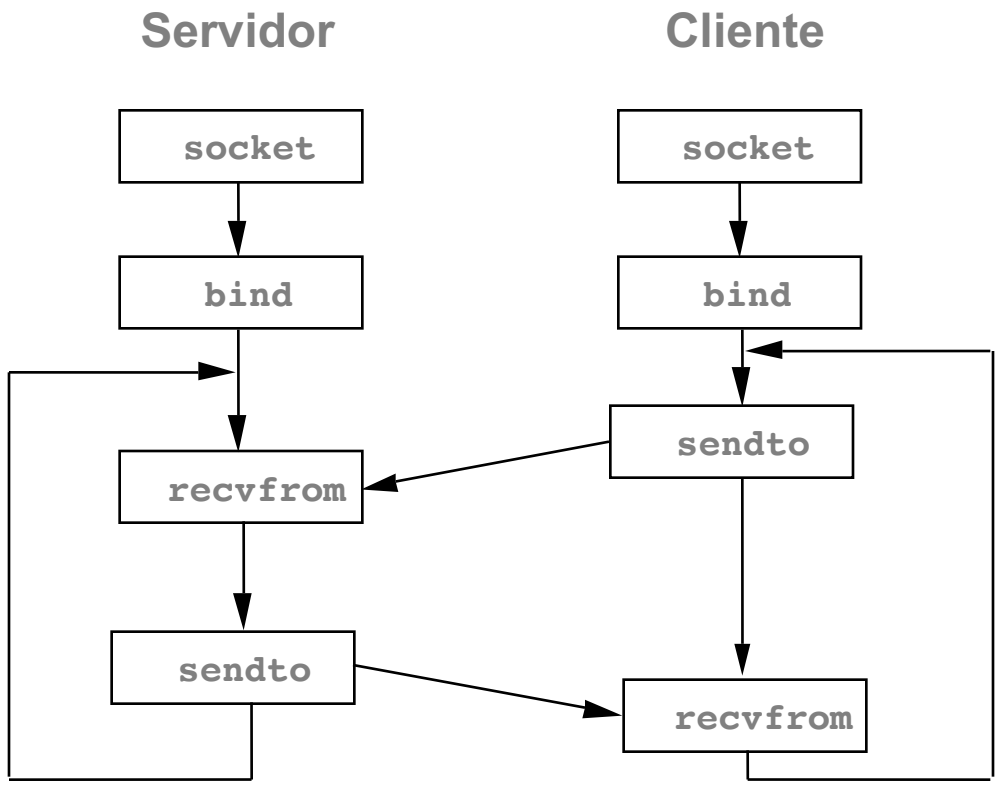


Interface *sockets*

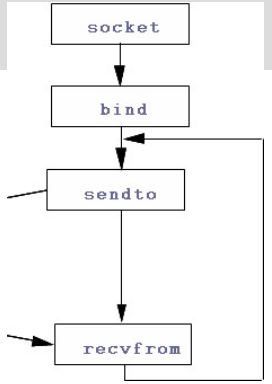
- Interface de programação para comunicação entre processos introduzida no Unix 4.2 BSD e standard POSIX

- Tipos de sockets
  - *Stream*: canal com ligação, bidirecional, fiável, interface tipo sequência de octetos
  - *Datagram*: canal sem ligação, bidirecional, não fiável, interface tipo mensagem
  - *Raw*: permite o acesso direto aos níveis inferiores dos protocolos (ex: IP na família Internet)

Sockets sem ligação



Sockets UDP em Java (Cliente)



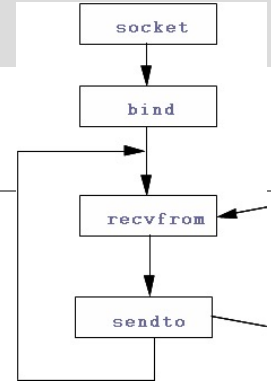

```
import java.net*;
import java.io*;
public class UDPClient{
   public static void main(String args[]){
      // args give message contents and server hostname
      DatagramSocket aSocket = null;
       try {
           aSocket = new DatagramSocket();
           byte [] m = args[0].getBytes();
           InetAddress aHost = InetAddress.getByName(args[1]);
           Int serverPort = 6789;
           DatagramPacket request =
                          new DatagramPacket(m, args[0].length(), aHost, serverPort);
           aSocket.send(request);
           byte[]buffer = new byte[1000];
           DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
           aSocket.receive(reply);
           System.out.println("Reply:" + new String(reply.getData()));
          } catch (SocketException e){System.out.println("Socket:" + e.getMessage());
        } catch (IOException e){System.out.println("IO:" + e.getMessage());
      } finally { if(aSocket ! = null) aSocket.close();}
   }
}
```

Constrói um socket datagram (associado a qualquer porto disponível)

Conversão do nome DNS para endereço IP

Cada mensagem enviada tem que levar junto o identificador do processo destino: IP e porto

Sockets UDP em Java (Servidor)



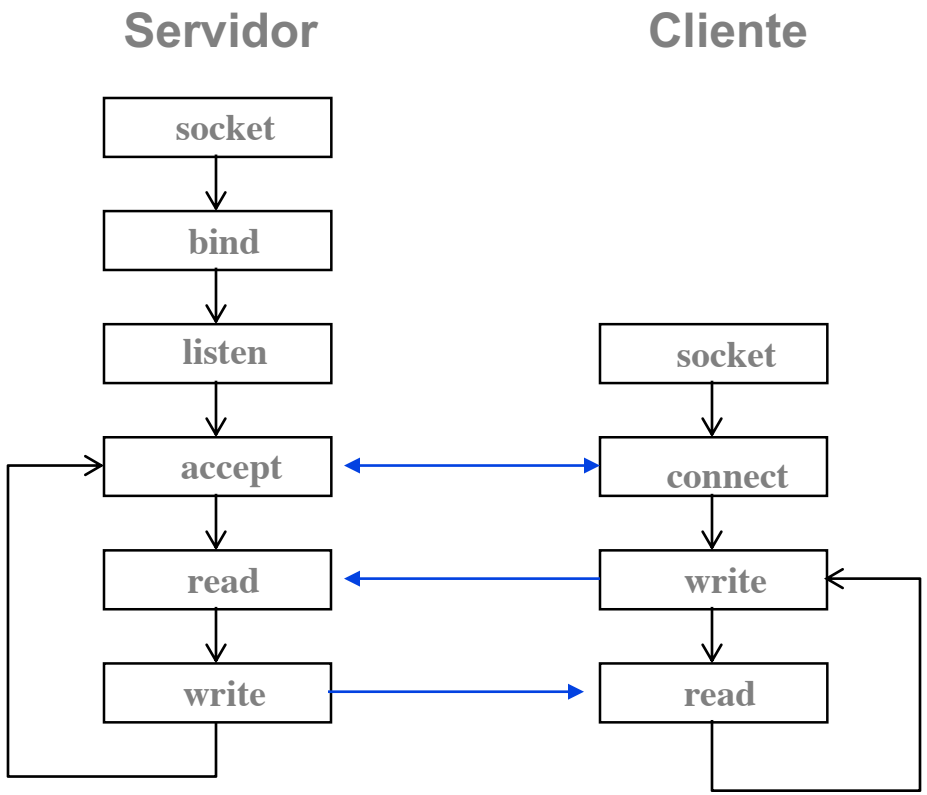

```
import java.net*;
import java.io*;
public class UDPServer{
   public static void main(String args[]){
      DatagramSocket aSocket = null;
       try{
           aSocket = new DatagramSocket(6789);
           byte[] buffer = new byte [1000];
           while(true){
              DatagramPacket request = new DatagramPacket(buffer, buffer.length);
              aSocket.receive(request);
              DatagramPacket reply = new DatagramPacket(request.getData(), request.getLength(),
                  request.getAddress(), request.getPort());
             aSocket.send(reply);
            }
          } catch (SocketException e){System.outprintln("Socket:"+ e.getMessage());
        } catch (IOException e){System.out.println("IO:" + e.getMessage());
      } finally {if(aSocket ! = null) aSocket.close();}
   }
}
```

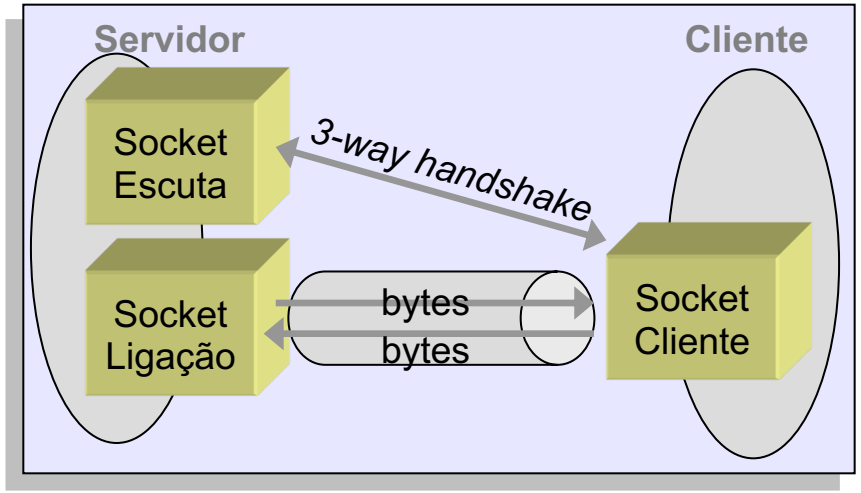
Constrói um socket datagram (associado ao porto 6789)

Recebe mensagem

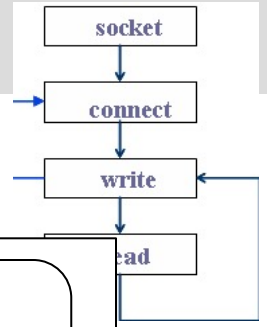
Extrai da mensagem o IP e porto do processo origem para responder

Sockets com ligação





Sockets TCP em Java (Cliente)



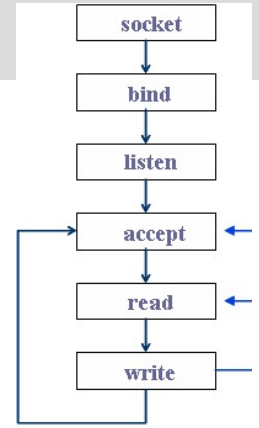

```
import java.net*;
import java.io*;
public class TCPClient{
   public static void main(String args[]){
   // args: message and destin. hostname
      Socket s = null;
      try{
          int server Port = 7896;
          s = new Socket (args[1], serverPort);
          DataInputStream in = new DataInputStream(s.getInputStream());
          DataOutputStream out = newDataOutputStream (s.getOutputStream());
           out.writeUTF(args[0]);
           String data = in.readUTF();
           System.out.prtintln("Received: " + data);
        }catch (UnknownHostException e){ System.out.println("Sock:" + e.getMessage());
       }catch (EOFException e){System.out.println("EOF:"e.getMessage());
      }catch (IOException e){System.out.println("IO:"e.getMessage());
     }finally {if(s!=null) try{s.close();}catch (IOException e}
   }
```

Classe *Socket* – suporta o *socket* cliente. Argumentos: nome DNS do servidor e o porto. Construtor, não só cria o *socket,* como efetua a ligação TCP

Métodos getInputStream() / getOutputStream() – permitem aceder aos dois streams definidos pelo *socket*

writeUTF() / readUTF() – *Universal Transfer Format* / para as cadeias de caracteres

Sockets TCP em Java (Servidor)



Cria socket servidor que fica à escuta no porto "serverPort"

```
import java.net*;
import java.io*;
public class TCPServer{
   public static void main(String args[]){
      try{
            int server Port = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true){
                Socket connectionSocket = listenSocket.accept();
                myConnection c = new myConnection(connectionSocket);
                }
        }catch (IOException e){ System.out.println("Listen:" +e.getMessage());}
     }
   }
```

Bloqueia até cliente estabelecer ligação. Cria novo socket com ligação ao do cliente e onde os dados são recebidos

Classe que cria uma thread para receber mensagens do cliente e responder aos pedidos.

Problema

Programar sistemas distribuídos usando sockets é um processo complexo, difícil e muito propenso a erros.

Problemas da programação com *Sockets*

- É tornada explícita a comunicação pela rede (o envio e receção de mensagens)
- É necessário o *marshalling*/*unmarshalling* (serialização/desserialização) da informação, entre sistemas (potencialmente) heterogéneos
  - Voltamos a este assunto mais à frente
- Consequência: **programação complexa, de baixo nível**

Primeiro objetivo de SD: simplificar programação

- Simplificar a tarefa de programação de aplicações cliente-servidor
  - Torná-la de mais **alto nível**
    - Mais próxima da atividade de programação com linguagens convencionais
  - **Evitar** atividades que consomem tempo e que são normalmente causadoras de erros
  - **Esconder** tanto quanto possível os detalhes de protocolos, endereços dos níveis de rede inferiores

Vamos aumentar o nível de abstração

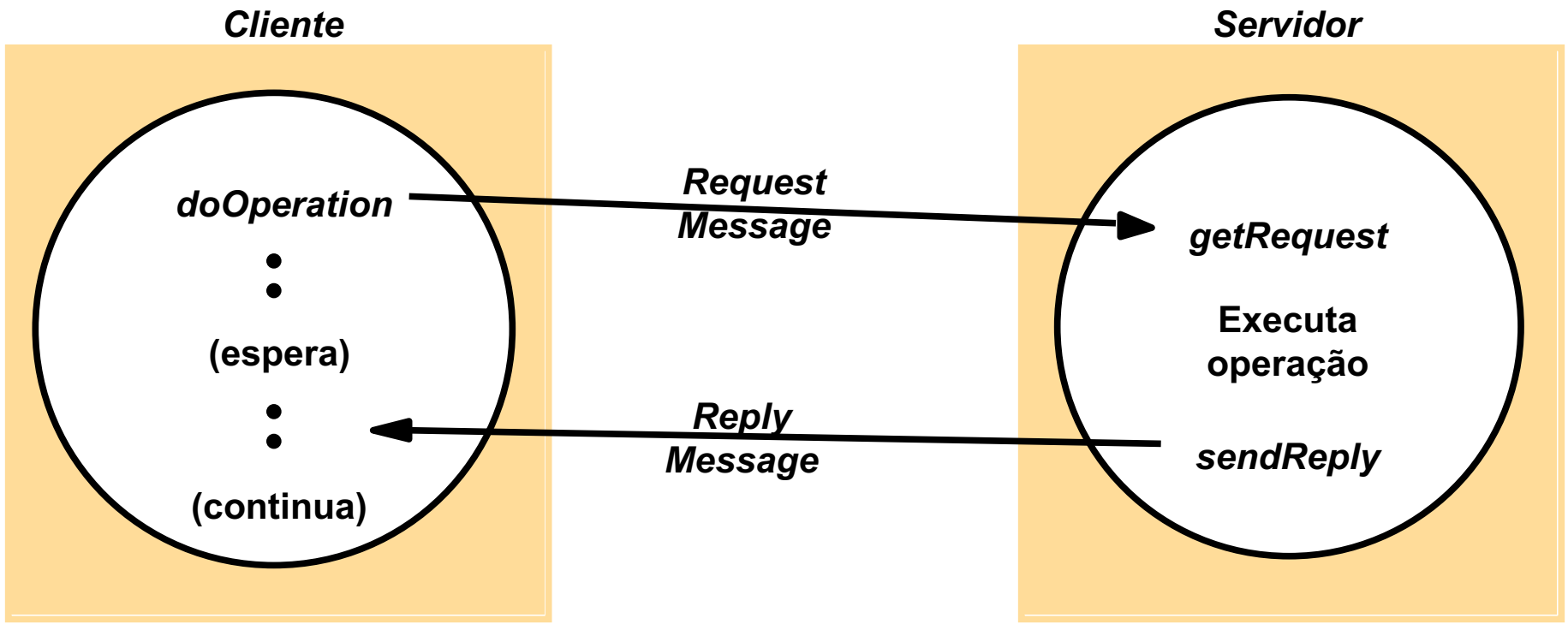
Projeto hipotético para resolvermos hoje

- Implementar um servidor de contagem que mantém um contador e oferece estas operações aos clientes:
  - ***Limpa***: coloca contador a zero
  - ***Incrementa***: incrementa o contador x unidades
  - ***Consulta***: devolve valor atual do contador

Projeto hipotético para resolvermos hoje

- Requisitos adicionais:
  - Rede não é fiável
    - Mensagens podem perder-se e chegar fora de ordem
  - Sistema heterogéneo
    - Servidor e clientes representam inteiros de forma diferente

Protocolo RR (Request, Reply)

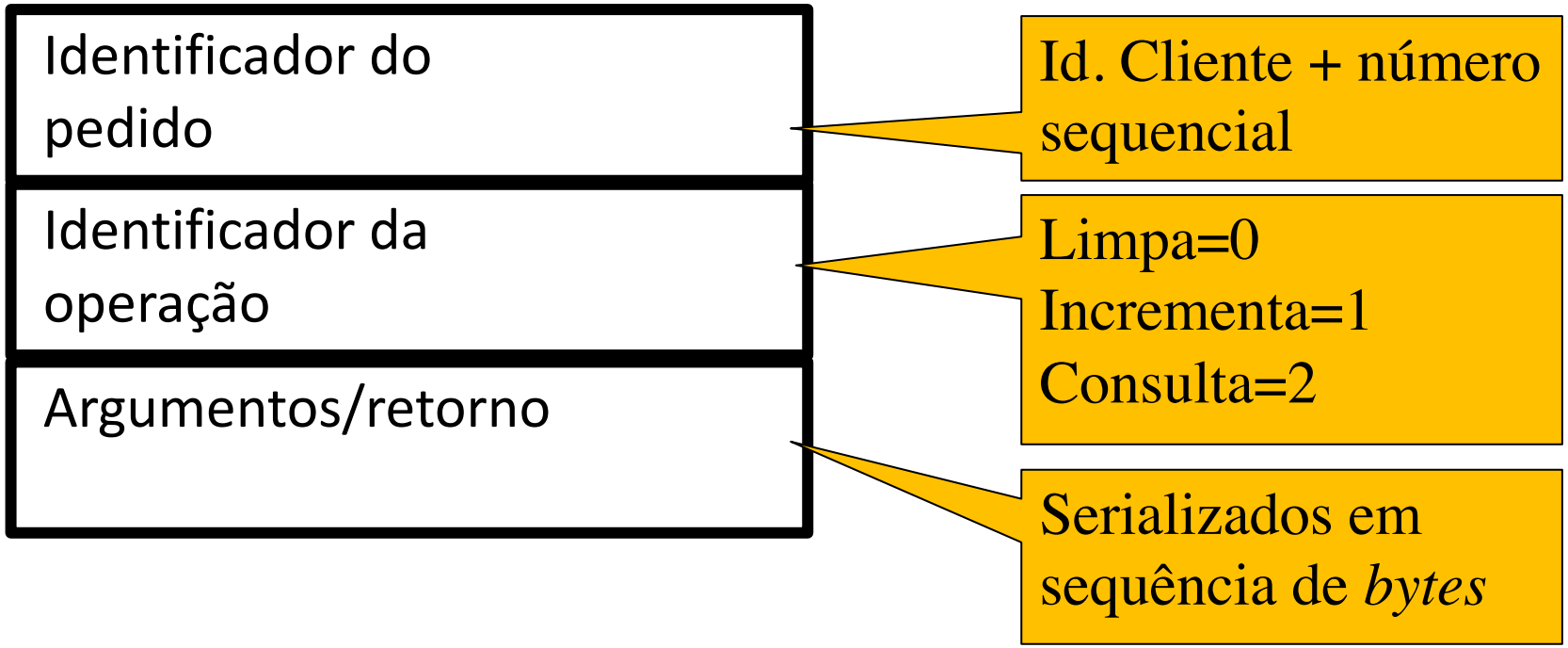


Vantagem: Já não é tornada explícita a comunicação pela rede.

TCP ou UDP?

- Vantagens do **TCP**
  - Oferece canal fiável sobre rede não fiável
- Mas talvez seja demasiado pesado:
  - Para cada invocação remota passamos a precisar de mais 2 pares de mensagens
    - SYN, ACK + FIN, ACK
  - Gestão de fluxo é redundante para as invocações simples do nosso sistema
  - Confirmações (ACKs) nos pedidos são desnecessárias
    - A resposta ao pedido serve de ACK
- Vamos assumir por isso que se usa **UDP**

Conteúdo das mensagens de pedido/resposta

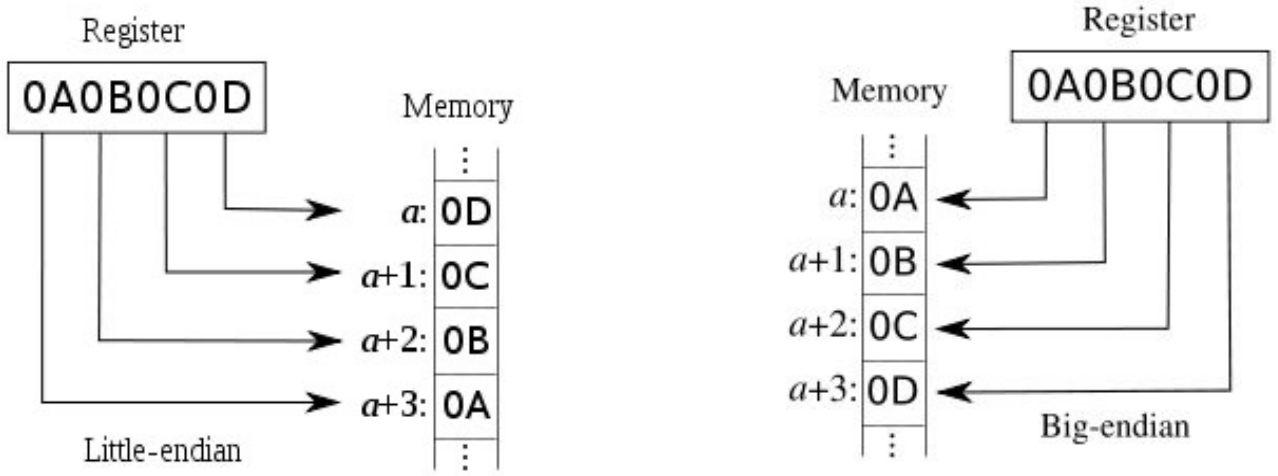


Como serializar os argumentos/retorno?

- É necessário **converter** estruturas de dados em memória para sequência de *bytes* que possam ser transmitidas pela rede
  - Encontrar uma forma de emissor e recetor heterogéneos interpretarem os dados corretamente
- Máquinas **heterogéneas** representam tipos de formas diferentes
  - É necessário traduzir entre representação de tipos do emissor e representação de tipos do recetor
  - Ou usar um formato canónico na rede
- *Marshalling*: serializar + traduzir para formato canónico
  - *Unmarshalling*: operação inversa

Qual o problema da heterogeneidade?

- Nos sistemas distribuídos a **heterogeneidade** é a regra
- Os formatos de dados são diferentes
  - Nos processadores (ex.: *little endian*, *big endian*, apontadores, vírgula flutuante)



  - Nas estruturas de dados geradas pelos compiladores
  - Nos sistemas de armazenamento (ex: strings ASCII vs Unicode)
  - Nos sistemas de codificação

Modelo de faltas

- Usando UDP para enviar mensagens, estas podem:
  - Perder-se
  - Chegar repetidas
  - Chegar fora de ordem
- E os processos podem falhar silenciosamente (por *crash*)
- Como lidar com isto?

Timeout no cliente

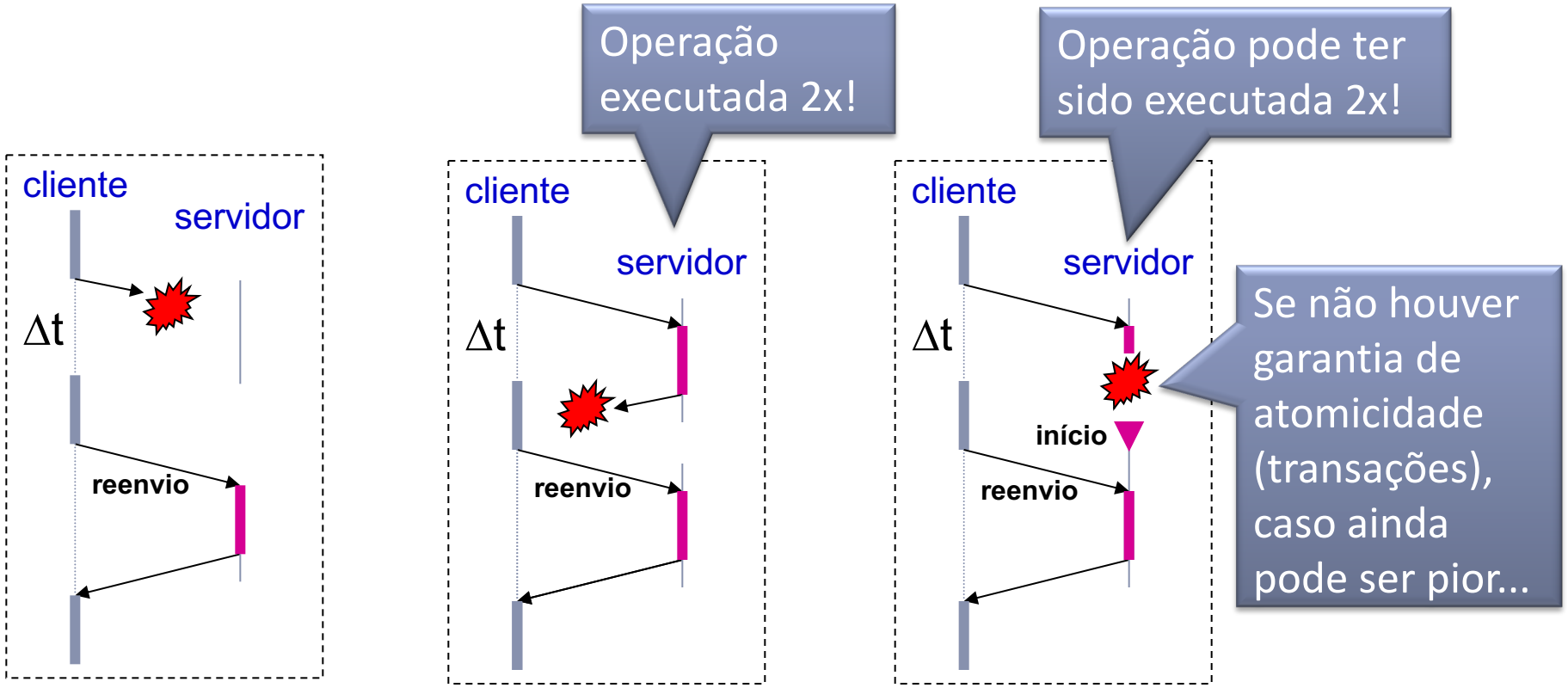
- Situação: cliente enviou pedido mas resposta não chega ao fim do *timeout*
- O que deve o cliente fazer?

- Hipótese 1: Cliente retorna erro

- Hipótese 2: Cliente reenvia pedido
  - Repete reenvio até receber resposta ou até número razoável de reenvios

Timeout no cliente com reenvio

- Quando a resposta chega após reenvio, o que pode ter acontecido?



Problema: execuções repetidas do mesmo pedido

- Perde-se tempo desnecessário
- Efeitos inesperados se operação não for **idempotente**

Função *limpa* é idempotente?
Função *incrementa* é idempotente?

Operação que, se executada repetidamente, produz o mesmo estado no servidor e resultado devolvido ao cliente do que se só executada 1 vez

Execuções repetidas do mesmo pedido: como evitar?

- Servidor deve ser capaz de verificar se *id.pedido* já foi recebido antes
- Se é a primeira vez, executa!
- Se é pedido repetido?
  - Deve guardar história de respostas de pedidos executados e retornar a resposta correspondente
  - Necessário guardar estado: e.g., tabela com (*id.pedido*, resposta)

Quantos pedidos manter por cliente?

Como escalar para grande número de clientes?

Problema: mensagens maiores que um datagrama UDP

- Podemos usar variante multi-pacote dos protocolos anteriores…
  - Implica implementar protocolo complicado
- Ou usar TCP!
  - Boa opção quando tamanho dos pedidos/respostas pode ser arbitrariamente grande
    - Exemplo: **solução usada no protocolo request-reply HTTP**
  - Nesse caso, implementação é mais simples pois TCP já assegura fiabilidade da comunicação
  - Como evitar o custo de estabelecimento de ligação?
    - Múltiplos pedidos por ligação, para amortizar o custo
    - Comum nas versões HTTP modernas

Vantagens e problemas do modelo request-reply

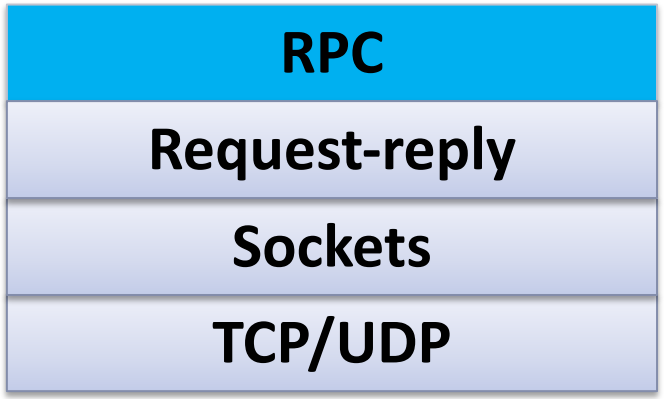
- **Vantagem:** já não é tornada explícita a comunicação pela rede (o envio e receção de mensagens)
- **Problema:** é necessário o *marshalling/unmarshalling* (serialização/desserialização) da informação, entre sistemas (potencialmente) heterogéneos
  - Estruturas de dados no emissor -> stream de bytes da mensagem -> estruturas de dados no destinatário
  - Necessário definir um formato para representação da informação
  - Difícil e *error-prone*

- **Continuamos com desafios de baixo nível complicados…**

Vamos aumentar um pouco mais o nível de abstração

Vamos aumentar um pouco mais o nível de abstração

Chamadas de Procedimentos Remotos (RPC)



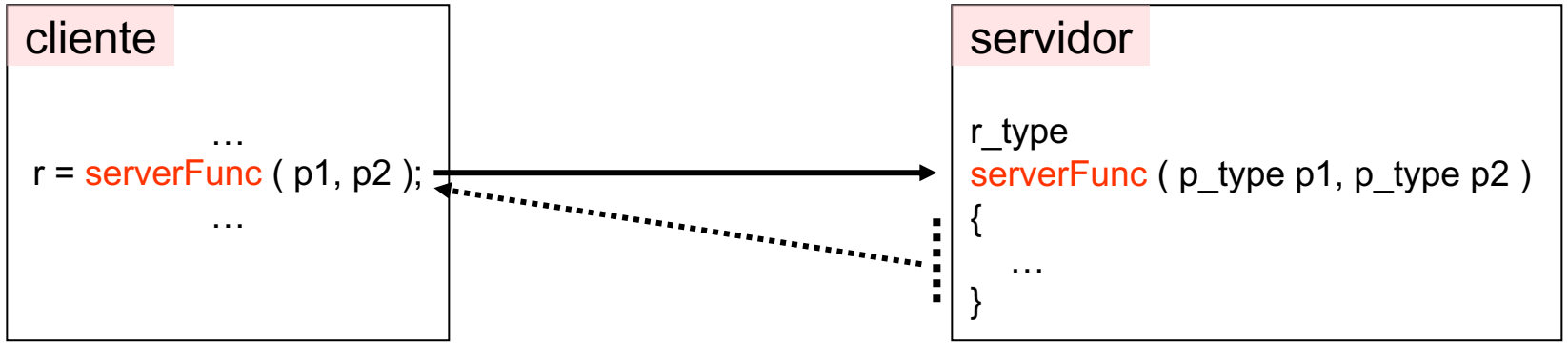
RPC - Remote Procedure Call

- Modelo de programação da comunicação num sistema cliente-servidor

- Objetivo:
  - Estruturar a programação distribuída com base na **chamada pelos clientes de procedimentos que se executam remotamente** no servidor
  - Modelo **mais genérico** do que request-reply, e mais próximo do modelo de programação convencional

RPC: visão do programador

- O programador chama uma função (procedimento) aparentemente local
- A função é executada remotamente no servidor
  - Acedendo a dados mantidos no servidor



RPC: visão do programador (II)

- Programador preocupa-se apenas em programar a lógica do negócio:
  - Código de cada função remota
  - Código do cliente (incluindo chamadas a funções remotas)

- Desafios da distribuição são (quase) **escondidos**

RPC: benefícios

Adequa-se ao fluxo de execução das aplicações

- Chamada síncrona de funções

Simplifica tarefas fastidiosas e delicadas

- Construção e análise de mensagens
- Heterogeneidade de representações de dados

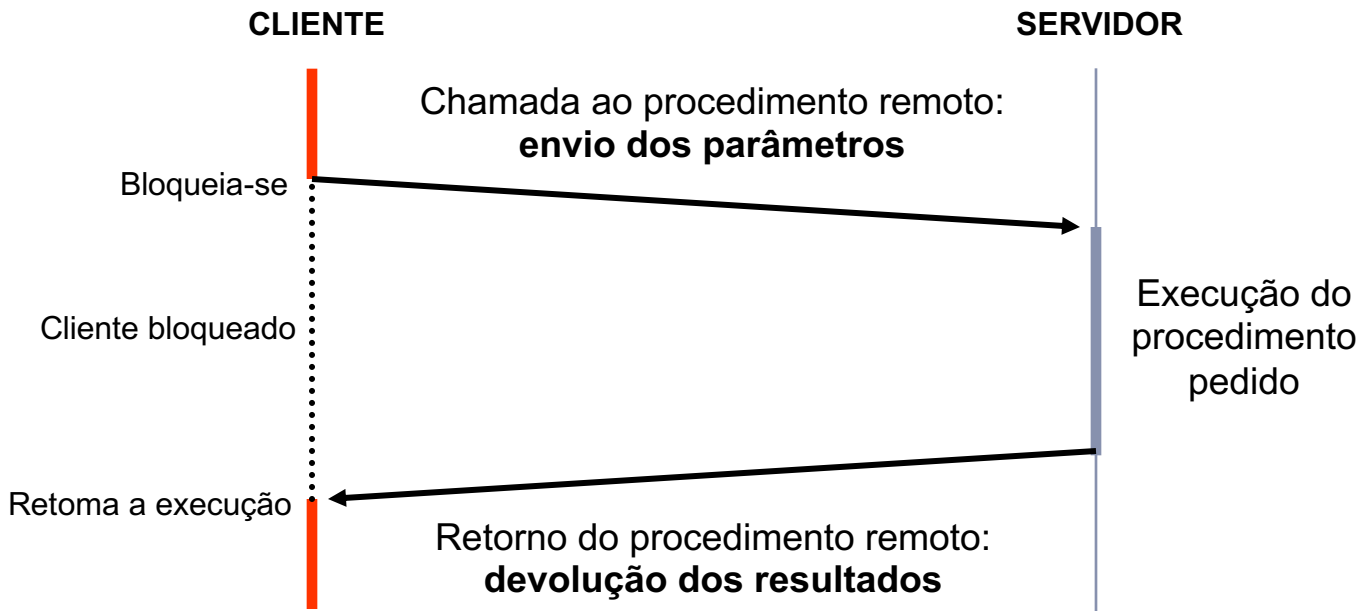
Esconde diversos detalhes do transporte

- Endereçamento do servidor
- Envio e receção de mensagens
- Tratamento de erros

Simplifica a divulgação de serviços (servidores)

- A interface dos serviços é fácil de documentar e apresentar
- A interface é independente dos protocolos de transporte

Fluxo de execução de uma chamada remota



Desafios

específico
- Definir estrutura das mensagens

genérico
- Localizar o porto do servidor

genérico
- Estabelecer canal de comunicação
  - Criar sockets, estabelecer ligação (caso seja TCP), etc.
- Para cada pedido/resposta:

específico
  - Criar mensagem de pedido/resposta

específico
  - Converter e serializar parâmetros

genérico
  - Enviar, reenviar, filtrar duplicados

O que é **genérico**?
O que é **específico** de cada projeto?

Dificuldades

O que o RPC nos oferece

específico
- Definir estrutura das mensagens

genérico
- Localizar o porto do servidor

genérico
- Estabelecer canal de comunicação
  - Criar sockets, estabelecer ligação (caso seja TCP), etc.
- Para cada pedido/resposta:

específico
  - Criar mensagem de pedido/resposta

específico
  - Converter e serializar parâmetros

genérico
  - Enviar, reenviar, filtrar duplicados

Aspetos genéricos:
Resolvidos por biblioteca de *run-time*

Aspetos específicos:
Programador define a interface remota
Compilador gera o código à medida

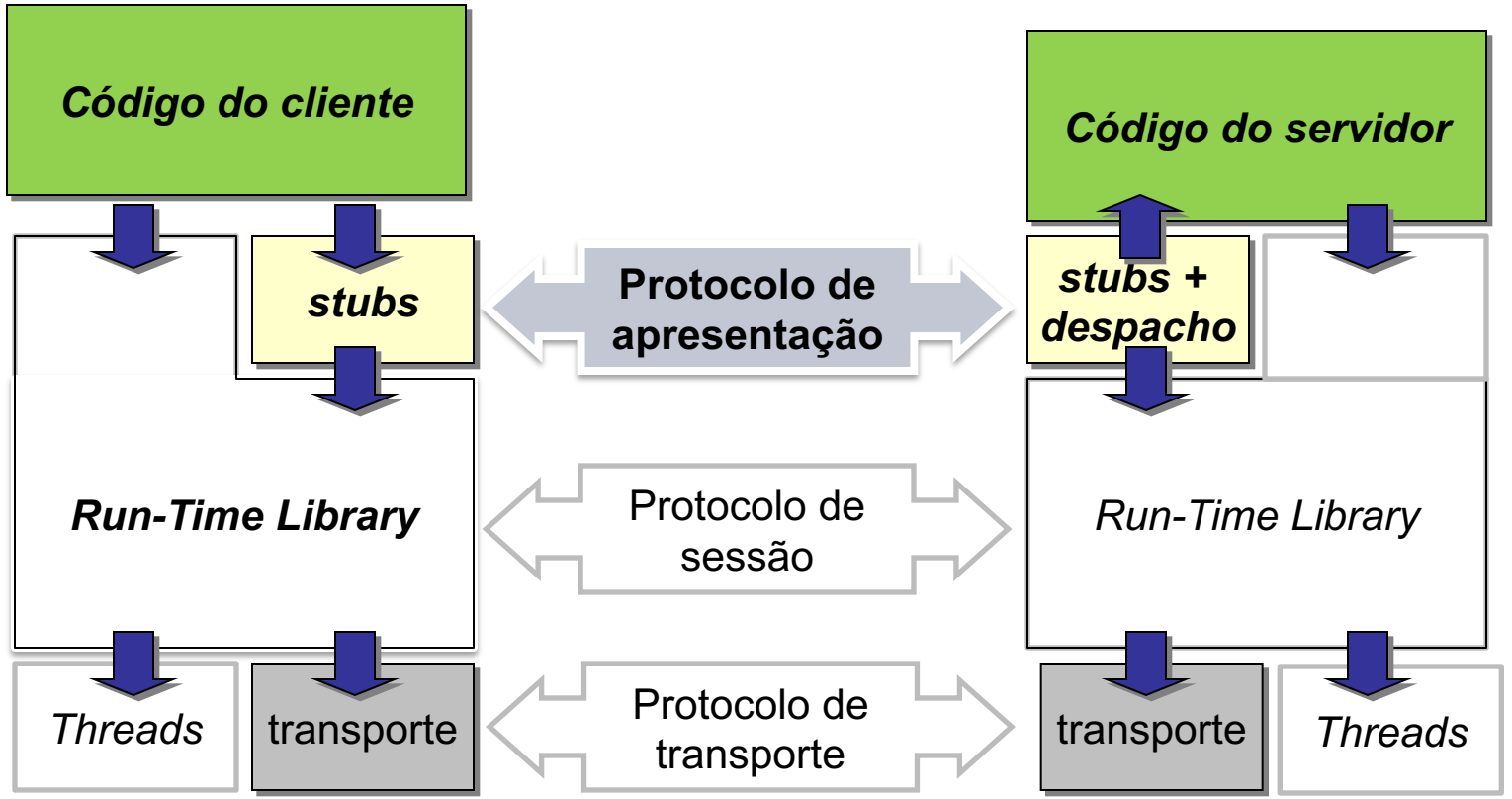
RPC

- Vamos apresentar os aspetos gerais do paradigma de RPC
- Começamos com um exemplo clássico: Sun RPC
  - Desenvolvido pela Sun Microsystems por volta de 1985 para suportar o sistema de ficheiros distribuído NFS
  - Especificação de domínio público
  - Implementação simples e muito divulgada em grande número de plataformas
  - Usa linguagem C

Estrutura do RPC

- Linguagem de descrição de interfaces remotas
- *Stubs* para adaptar dados de cada procedimento
- Biblioteca de *run-time* para o suporte genérico
- Gestor de nomes para localizar servidores

Tudo junto dá...



Estrutura do RPC

Linguagem de descrição de interfaces remotas

Stubs para adaptar dados de cada procedimento

Biblioteca de *run-time* para o suporte genérico

Gestor de nomes para localizar servidores

RPC IDL

- Linguagem própria para descrição de interfaces
  - Linguagem declarativa
  - Não descreve implementação
  - Permite que procedimentos escritos numa linguagem possam ser invocados por outra

- Permite definir:
  - Tipos de dados
  - Protótipos de funções
    - Incluindo parâmetros de entrada e de saída
  - Interfaces remotas
    - Conjuntos de funções

IDL: Pode ser simplesmente um ".h"?

- Quais os parâmetros de entrada/saída da seguinte função?

```
int transfere(int origem, int destino, int valor,
  int *saldo, char *descr);
```

IDL: Pode ser simplesmente um ".h"?

Não…

- Como serializar parâmetros como estes?
  - Endereçamento puro de memória (void *)
  - Ponteiros para um vetor
    - Por exemplo, vetor de tamanho fixo ou string terminada em '\0'
  - Passagem de variáveis por referência (&var)
  - Booleano (passado como *int*)
  - Estruturas dinâmicas com ponteiros

Como o Sun RPC resolve estas ambiguidades

- Apenas um parâmetro de entrada e um de saída o que resolve a ambiguidade da mensagem de envio e de resposta
  - Se houver necessidade de mais parâmetros será necessário encapsulá-los numa estrutura
- Novo tipo *String* resolve o vetor de caracteres (char* é ambíguo)
- *Boolean* elimina a ambiguidade nos valores lógicos
- *Pointers* existem e o *stub* copia a estrutura referenciada para a mensagem
  - Passagem de **parâmetro por cópia**

- Um compilador (**rpcgen**) gera automaticamente o programa principal do servidor

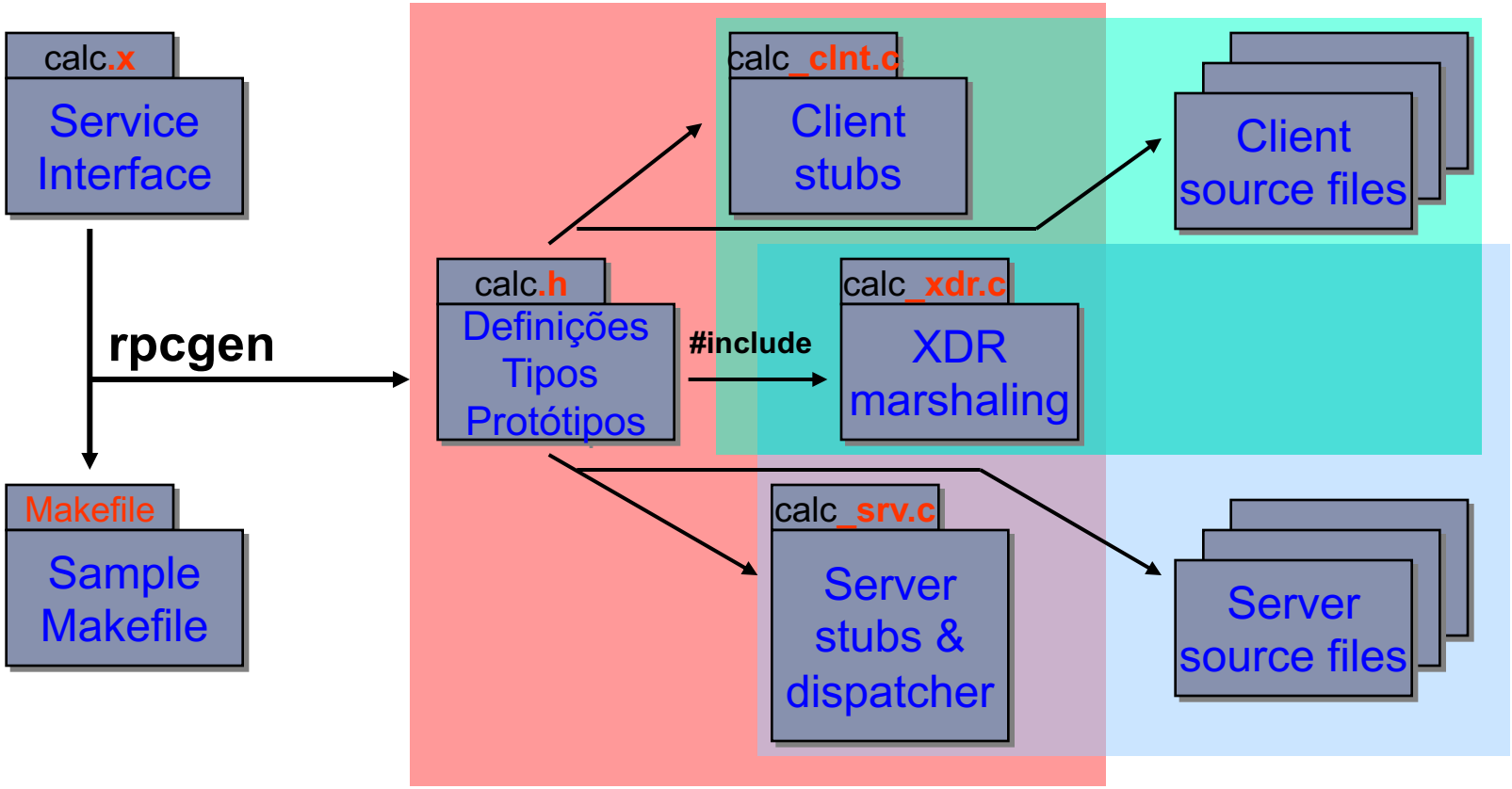
Exemplo: IDL Sun RPC

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
        int length;
        char buffer[MAX];
};
struct writeargs {
        FileIdentifier f;
        FilePointer position;
        Data data;
};
```

```
struct readargs {
        FileIdentifier f;
        FilePointer position;
        Length length;
};

program FILEREADWRITE {
  version VERSION {
        void WRITE(writeargs)=1;        1
        Data READ(readargs)=2;          2
  }=2;
} = 9999;
```

Diagrama de ficheiros



Outro exemplo: Java RMI

RMI vs RPC: semelhanças

- O uso de **interfaces** para definir os métodos invocáveis remotamente

- Existe um protocolo de invocação remota que oferece as **semânticas habituais** de RPC
  - Pelo-menos-uma-vez, no-máximo-uma-vez, etc.

- Temos um **nível de transparência semelhante**
  - Invocações locais e remotas com sintaxe próxima, mas programador é exposto a alguns aspetos da distribuição (e.g., exceções remotas)

RMI vs RPC: diferenças (I)

- Em RMI o programador tem ao seu dispor o **poder completo do paradigma OO**
  - Objetos, classes, herança, polimorfismo, etc.

- As linguagens com objetos já são **fortemente tipificadas** e têm a **noção de interface** através de classes abstratas
  - **Não é necessário usar IDL distinta**
  - Contudo é fundamental considerar que **indicações suplementares** dar ao compilador para distinguir um objeto local de um remoto

RMI vs RPC: diferenças (II)

- **Passagem por referência** é agora permitida
  - Cada objeto (local e remoto) tem um identificador único, e assim os objetos podem ser referenciado também remotamente
  - Referência remotas podem ser passadas por argumento ou retorno quando um método é invocado remotamente

- Num servidor existem vários objetos remotos e respetivas interfaces remotas
  - Em RPC, 1 servidor apenas oferecia 1 interface remota

Interface remota

- Conjunto de métodos de uma classe que podem ser **invocados remotamente** nos objetos dessa classe
  - Uma classe pode também definir métodos que podem ser acedidos apenas localmente

- A especificação da interface é semelhante a uma classe abstrata ou interface na linguagem Java

Exemplo em Java RMI: Interfaces remotas

```
import java.rmi.*;


public interface Account extends Remote {
      float debit(float amount) throws RemoteException, InsufficientFundsException;
      float credit(float amount) throws RemoteException;
}


public interface AccountList extends Remote {
      Account getAccount(int id) throws RemoteException;
      void addAccount(int id, Account a) throws RemoteException;
}
```

O super-tipo **Remote** indica uma interface que tem métodos invocáveis remotamente

Os métodos remotos lançam uma exceção especifica: **RemoteException**

Objetos remotos podem ser usados como argumentos (desde que *serializable*)...

...e como resultados

Mais um exemplo: Web Services

Até agora

- Sun RPC para sistemas **baseados em C**
- Java RMI para sistemas **baseados em Java**

Motivação dos Web Services (I)

- Protocolo simples para garantir a **interoperação** entre plataformas de múltiplos fabricantes/multi-linguagem
- Tratar todo o tipo de **heterogeneidade de dados e informação**
  - Com XML
- Permitir a **transferência** de todo o tipo de informação
  - Estruturas de dados
  - Documentos estruturados
  - Dados multimédia
- Usar **URL e URI** como referências remotas

Motivação dos Web Services (II)

- Permitir utilizar **RPC ou MOM** (*Message Oriented Middleware*)
  - Comunicação síncrona ou assíncrona
- Usar **protocolos de transporte amplamente conhecidos**
  - HTTP para passar através de *firewalls*
  - SMTP (email)
  - MQ (filas de mensagens persistentes *store-and-forward*)
  - Outros

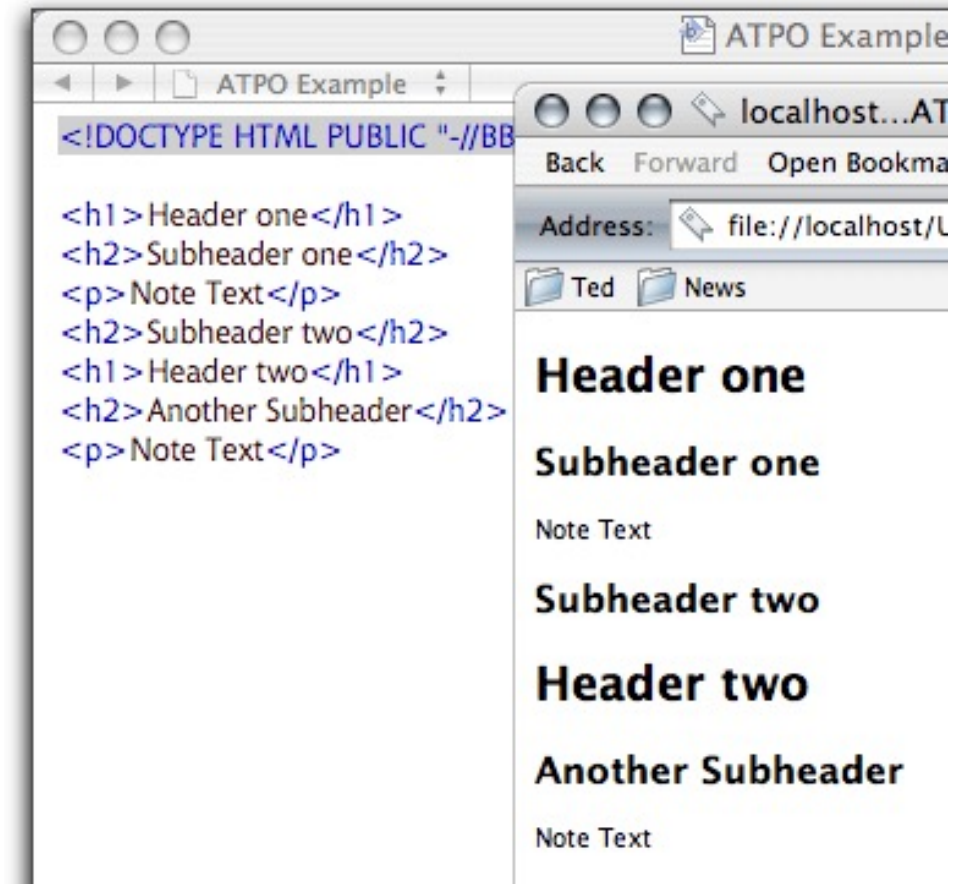
Em 2 palavras: heterogeneidade e generalidade

Porque se chama **Web** Service?

- Porque usa as tecnologias de ***World Wide Web***

- *Uniform Resource Identifiers* (URIs)
  - Identificam documentos e outros recursos da WWW
- *HyperText Transfer Protocol* (HTTP)
  - Protocolo de interação cliente-servidor baseado em TCP/IP
- *HyperText Markup Language* (HTML)
  - Linguagem para especificar conteúdos e apresentação das páginas apresentadas nos browsers

HyperText Markup Language - HTML



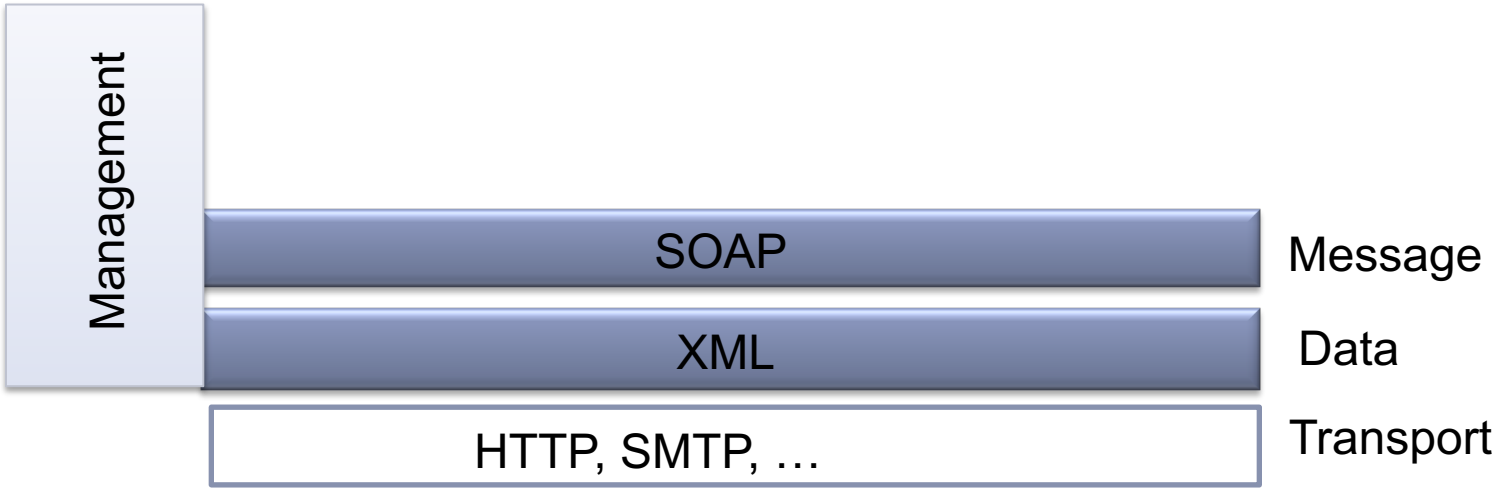
- Essencialmente orientado a **apresentação** da informação
  - E não à **descrição dos dados**
  - Pouco útil para os Web Services

eXtensible Markup Language - XML

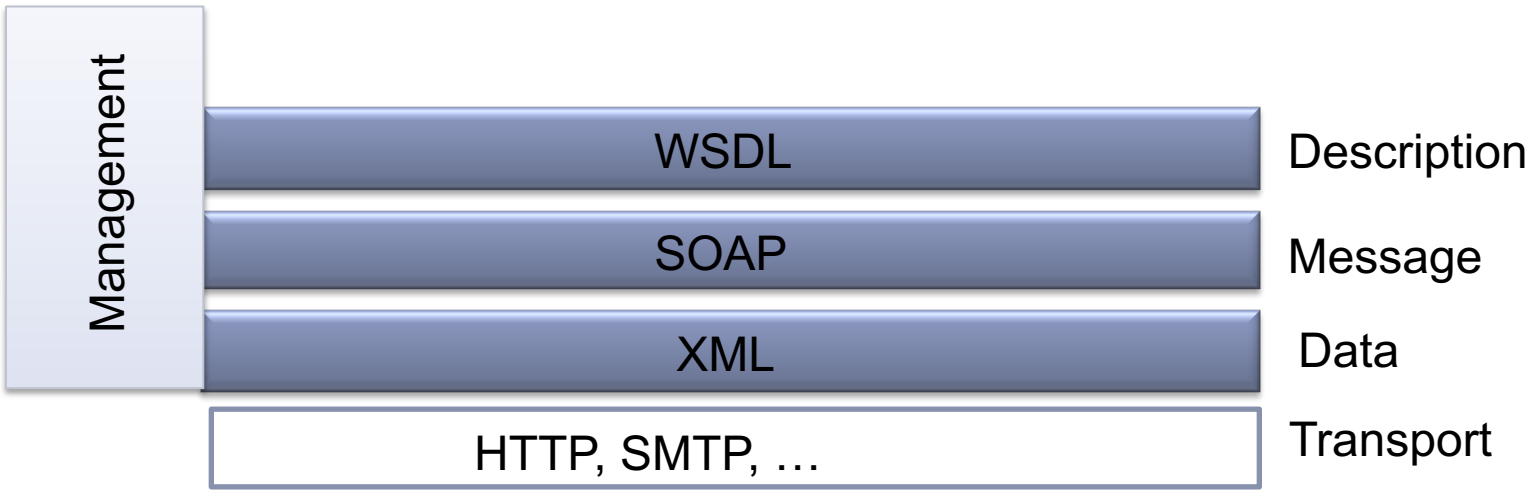
```
<note>
  <from>Mike</from>
  <to>Jo</to>
  <heading>Reminder</heading>
  <body>Meet for tea</body>
</note>
```

- Linguagem de etiquetas
  - Tal como o HTML
- Focado na **descrição** do conteúdo da informação
  - E não no seu **aspeto**

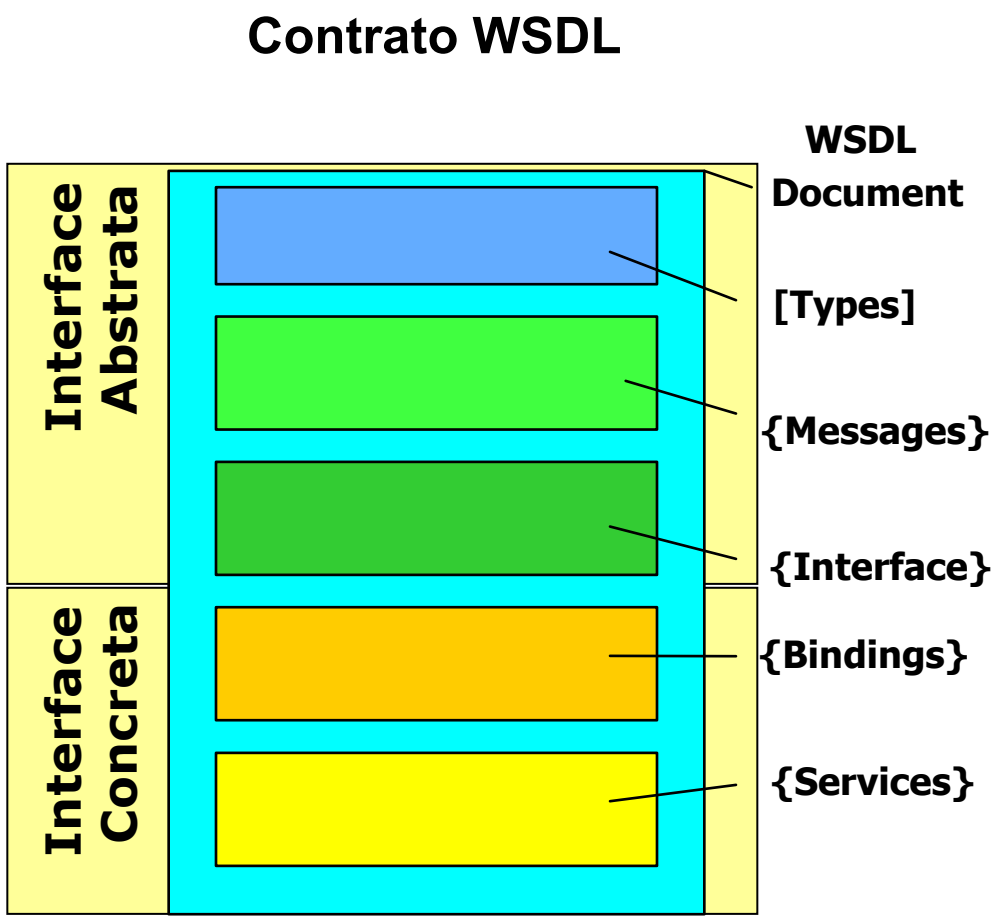
Web Services standards base



Web Services standards base



Web Services Description Language



- **Acordo** cliente-servidor relativamente ao serviço
  - Geralmente serve para gerar os stubs
- Mais **flexível** que outros IDLs
  - Para permitir vários tipos de interação
    - Request-reply
    - Troca de documentos

Estrutura do RPC

Linguagem de descrição de interfaces remotas

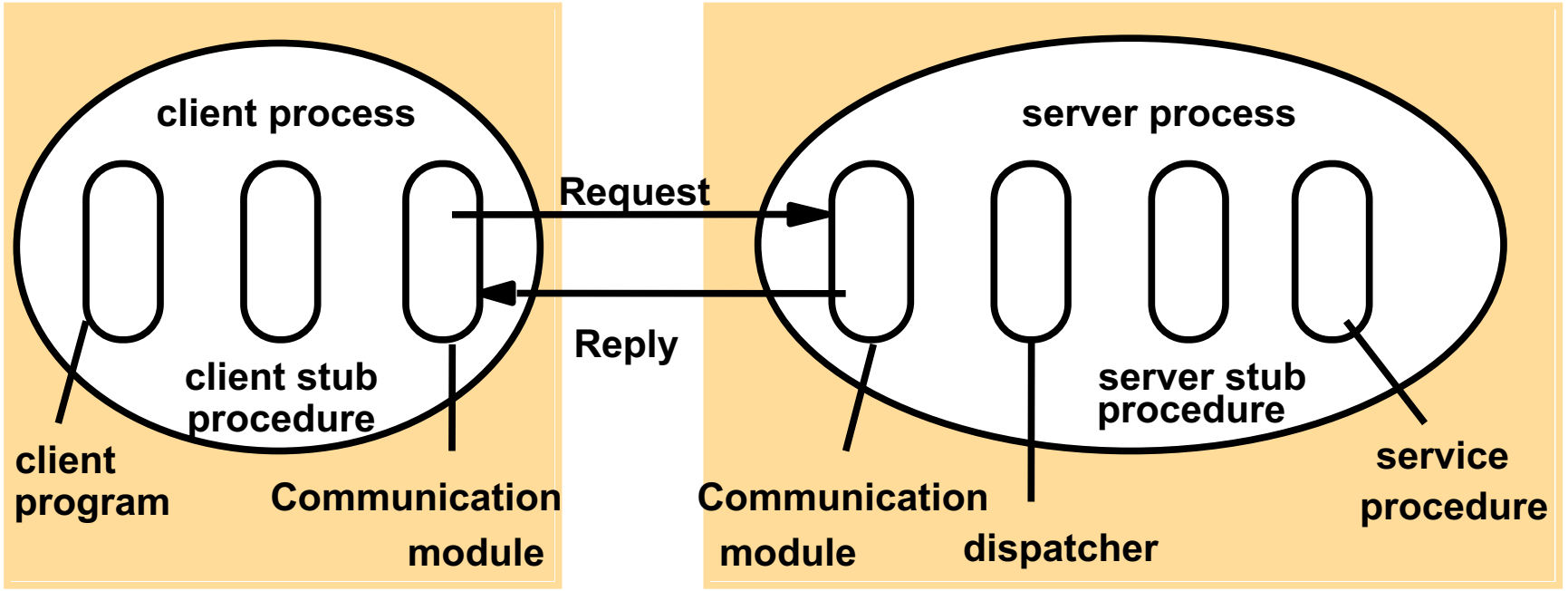
Stubs para adaptar dados de cada procedimento

Biblioteca de *run-time* para o suporte genérico

Gestor de nomes para localizar servidores

Rotinas de adaptação (*stubs*)

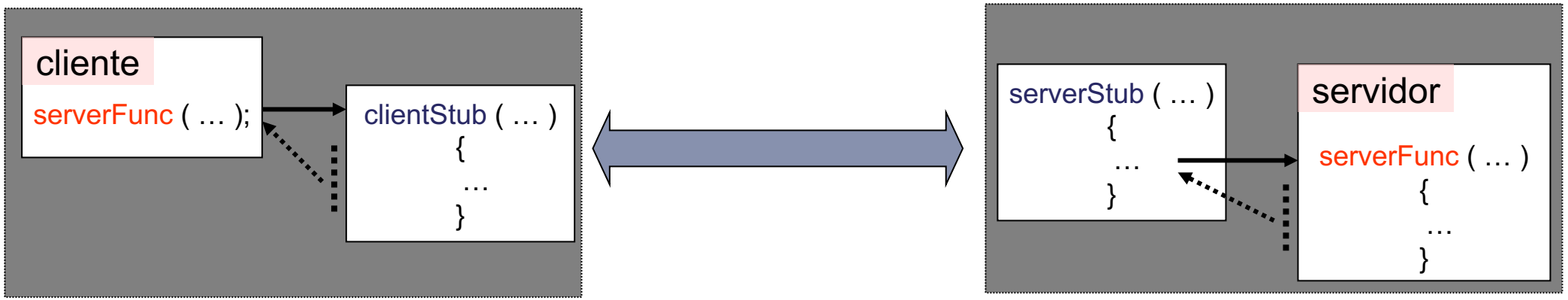
- Cada função remota tem um *stub*
  - Do lado do cliente e do lado do servidor
- Elemento chave para oferecer a ilusão de uma chamada local



Rotinas de adaptação (*stubs*)

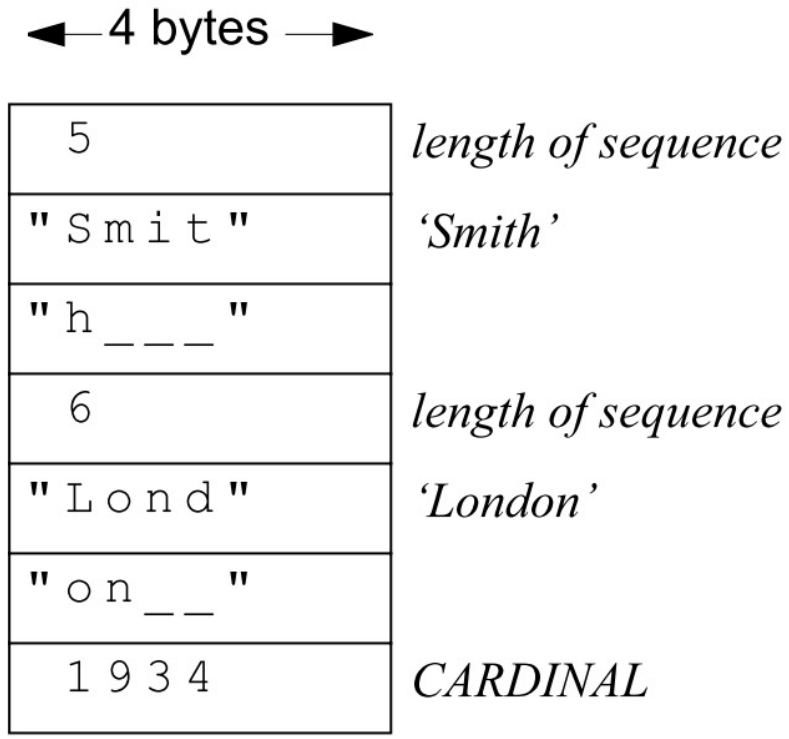
- *Stubs* Cliente
  - **Conversão** de parâmetros
  - Criação e envio do pedido
  - Recepção e análise da resposta
  - **Conversão** de retorno

- *Stubs* Servidor
  - Recepção e análise do pedido
  - **Conversão** de parâmetros
  - Chamada da função local
  - **Conversão** de retorno
  - Criação e envio da resposta



Sun XDR (External Data Representation)

- Exemplo de *marshalling* de parâmetros em mensagem Sun XDR:
  - ‘Smith’, ‘London’, 1934



Função de despacho do servidor (*dispatch*)

- Espera por mensagens de clientes num porto de transporte
- Envia mensagens recebidas para o *stub* respetivo
  - Ou seja, analisa o código de operação da mensagem
- Recebe mensagem retornada pelo *stub* e envia-a para o cliente

Estrutura do RPC

Linguagem de descrição de interfaces remotas

Stubs para adaptar dados de cada procedimento

Biblioteca de *run-time* para o suporte genérico

Gestor de nomes para localizar servidores

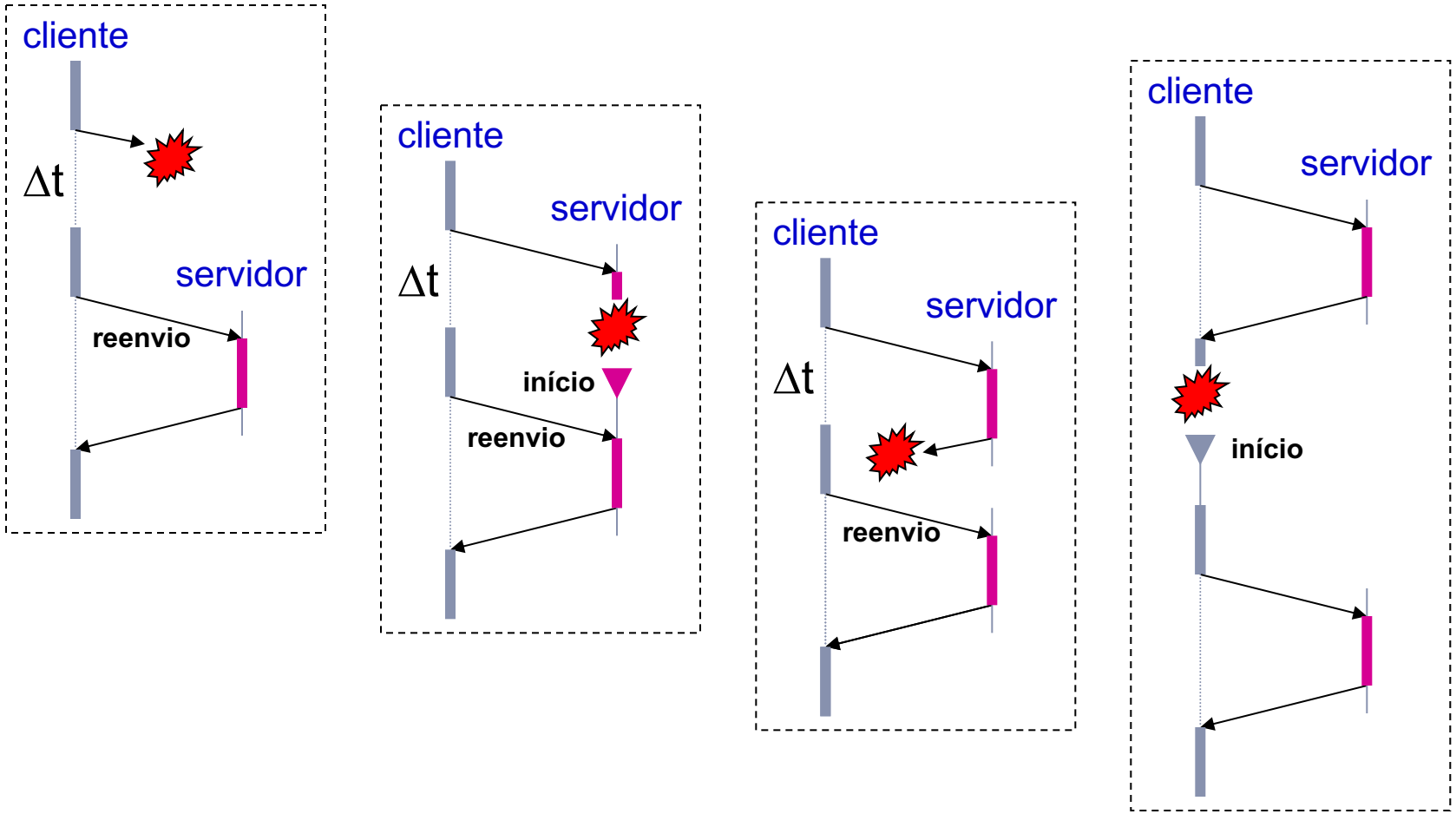
Biblioteca de *run-time*

- Suporta as operações genéricas do RPC
- Por exemplo:
  - Localizar o porto do servidor, registar o servidor
  - Inicializar portos de comunicação
  - Estabelecer ligação entre cliente e servidor
  - Construir mensagens
  - Converter e serializar tipos primitivos de parâmetros
  - Autenticação de cliente e servidor
  - Enviar e receber mensagens
    - Incluindo fragmentação, reenvio, filtragem de duplicados, etc.
    - Definindo a **semântica de execução**

Semânticas de execução

- A semântica de execução determina o modelo de recuperação de faltas
  - Semântica ideal ≡ procedimento local
- Modelo de faltas
  - Perda, duplicação ou reordenação de mensagens
  - Faltas no servidor e no cliente
  - Possibilidade de servidor e cliente reiniciarem após a faltas

Perante estas faltas, como garantir uma dada semântica de execução?



Semântica de execução

- A semântica de execução do RPC é sempre considerada na **ótica do cliente**
- Se a chamada retornar no cliente, **o que é que se pode inferir da execução**, considerando que existe um determinado modelo de faltas?
- O modelo de faltas especifica quais as faltas que podem ocorrer

Semânticas de execução

Talvez (*maybe*)

Pelo-menos-uma-vez (*at-least-once*)

No-máximo-uma-vez (*at-most-once*)

Exatamente-uma-vez (*exactly-once*)

Semânticas de execução

Semântica **talvez**

- O *stub* cliente não recebe uma resposta num prazo limite
- O *timeout* expira e a chamada retorna com erro
- Neste caso o cliente **não sabe se o pedido foi executado ou não**
- Se receber resposta, sabe que foi processado **pelo menos uma vez**

Protocolo

- Protocolo não pretende tolerar nenhuma falta pelo que **nada faz** para recuperar de uma situação de erro (não há retransmissões).

Semânticas de execução

Semântica **pelo-menos-uma-vez**

- O *stub* cliente não recebe uma resposta num prazo limite
- O *timeout* expira e **o *stub* cliente repete o pedido até obter uma resposta**
- Se receber uma resposta o cliente **tem a garantia** que o pedido foi executado **pelo menos uma vez**
- Para evitar que o cliente fique permanentemente bloqueado em caso de falha do servidor existe um segundo *timeout* mais amplo

Para serviços com funções **idempotentes**

Semânticas de execução

Semântica **no-máximo-uma-vez**

- O *stub* cliente não recebe uma resposta num prazo limite
- O *timeout* expira e o *stub* cliente repete o pedido
- O servidor **não executa pedidos repetidos**
- Se receber uma resposta, o cliente tem a **garantia** que o pedido foi executado no máximo uma vez

O protocolo de controlo tem que:

- Identificar os pedidos para **detetar repetições** no servidor
- **Manter estado no servidor** acerca dos pedidos em curso ou que já foram atendidos

Resumo de técnicas para cada semântica

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

E no caso da semântica **exatamente-uma-vez**?

Semânticas de execução

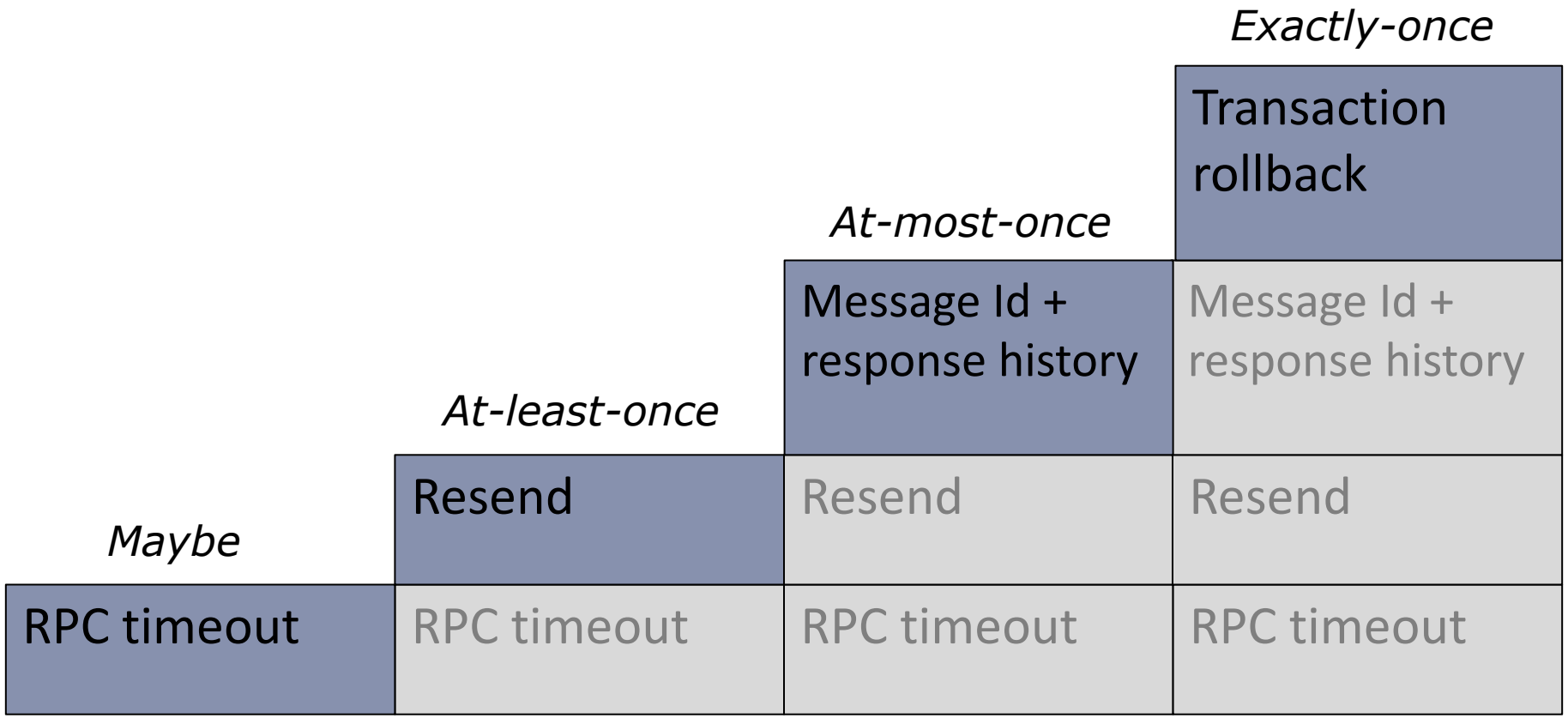
Semântica exatamente-uma-vez

- O *stub* cliente não recebe uma resposta num prazo limite
- O *timeout* expira e o *stub* cliente repete o pedido
- O servidor não executa pedidos repetidos
- **Se o servidor falhar existe a garantia de fazer *rollback* ao estado do servidor de modo a não ficar com pedidos "a meio"**

Protocolo

- Servidor e cliente com funcionamento transacional

RPC: semânticas de invocação e mecanismos necessários



Estrutura do RPC

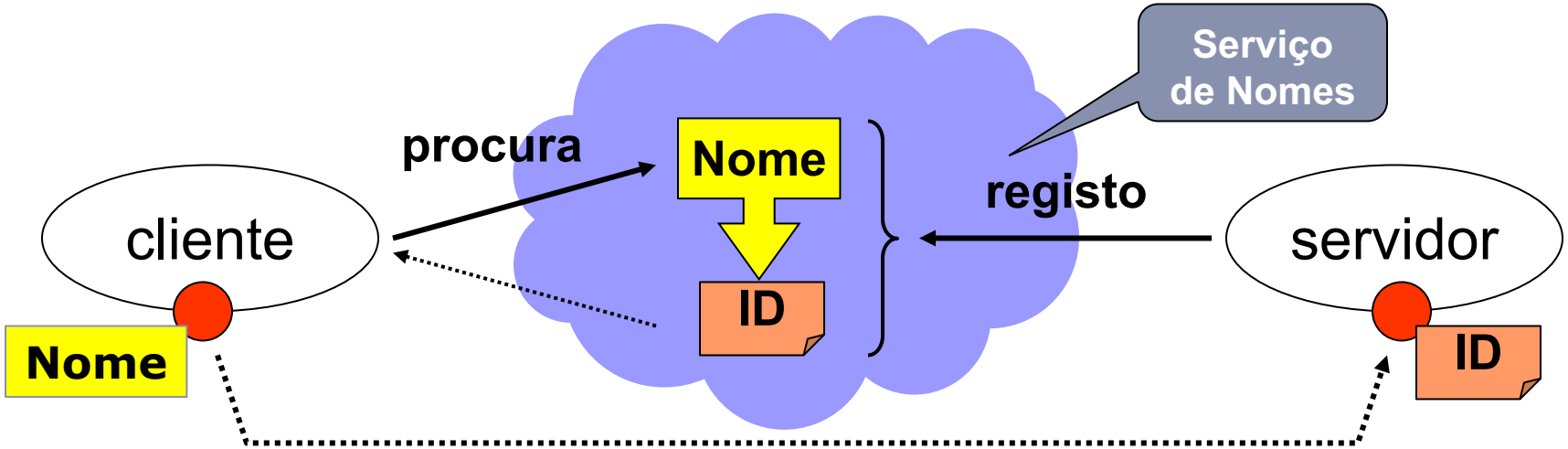
Linguagem de descrição de interfaces remotas

Stubs para adaptar dados de cada procedimento

Biblioteca de *run-time* para o suporte genérico

Gestor de nomes para localizar servidores

Num sistema cliente-servidor, como permitir que cliente descubra o servidor?



Este problema coloca-se em qualquer sistema distribuído!

Exemplos de Serviços de Nomes

- Serviços de **Nomes** – guardam informação **essencial**
  ->Recebem nome, devolvem endereço
  - DNS (Domain Name System)
  - rpcbind (SUN RPC)
  - RMI registry (Java RMI)

- Serviços de **Diretório** – guardam informação **mais rica**
  ->Recebem pesquisas por atributo, devolvem conjuntos de objetos
  - NIS (Network Information System)
  - DCE CDS (Cell Directory Service), GDS (Global Directory Service)
  - X.500
    - Active Directory da Microsoft
    - *Lightweight Directory Access Protocol* (LDAP)
  - UDDI (Web Services)

DNS (*Domain Name Service*)

- Arquitectura para registo e **resolução de nomes** de máquinas da Internet
  - Inicialmente proposta em 1983

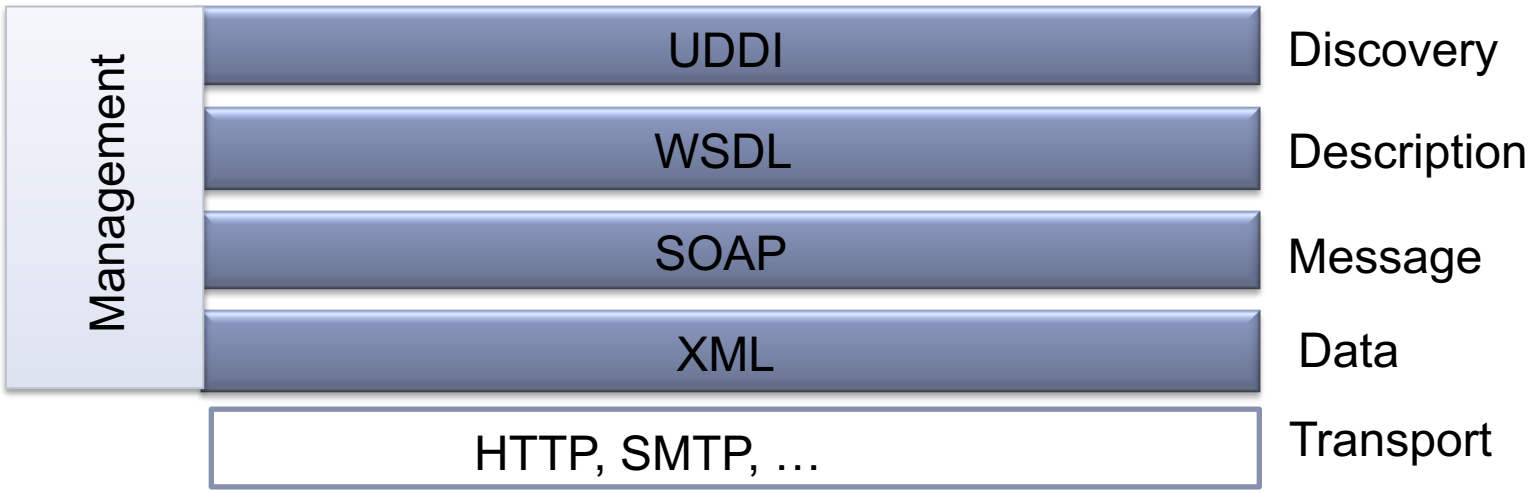
- Exemplo de concretização:

  UNIX BIND (*Berkeley Internet Name Domain*)

UC REDES

De regresso aos Web Services...

Web Services standards base

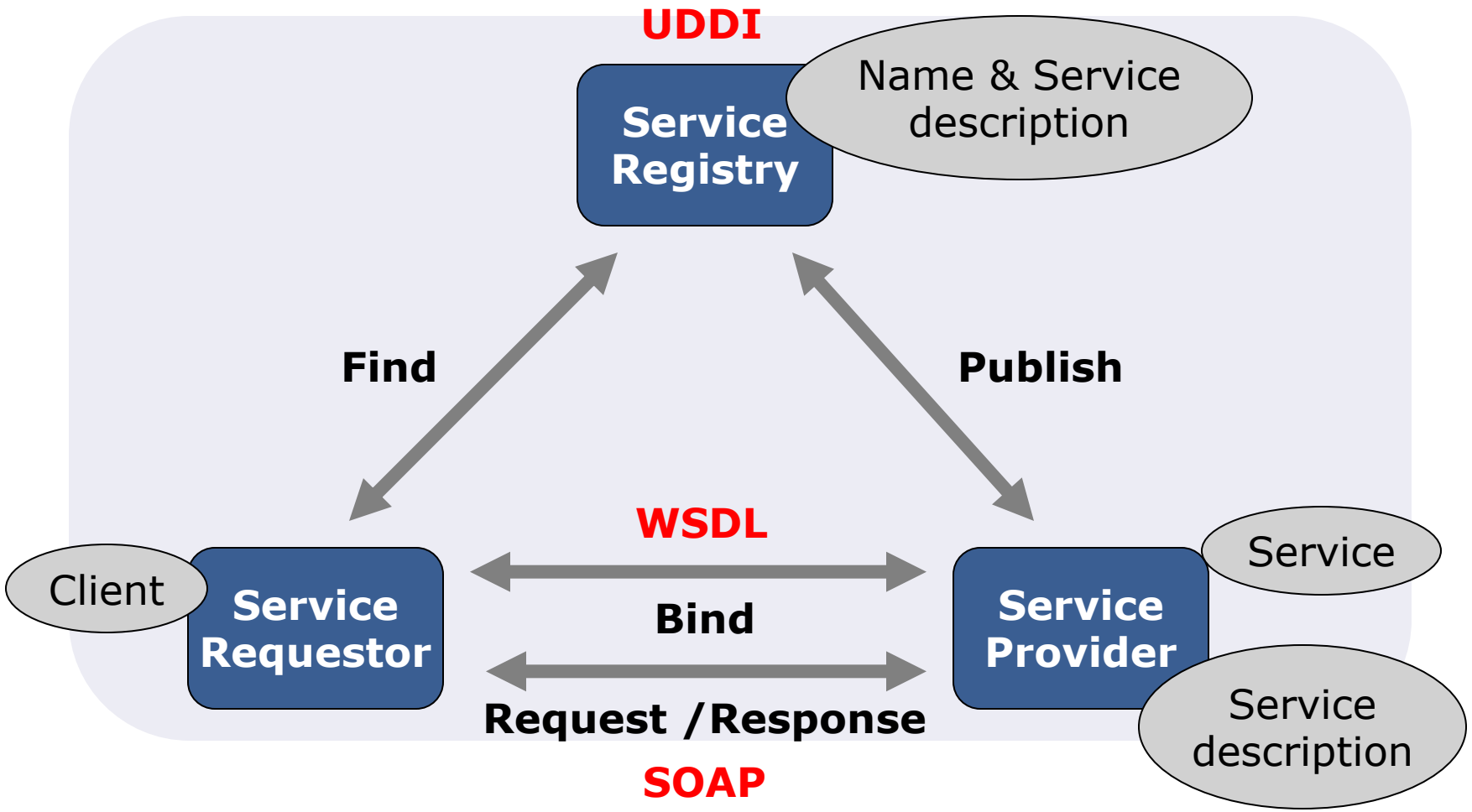


Universal Data Discovery and Integration

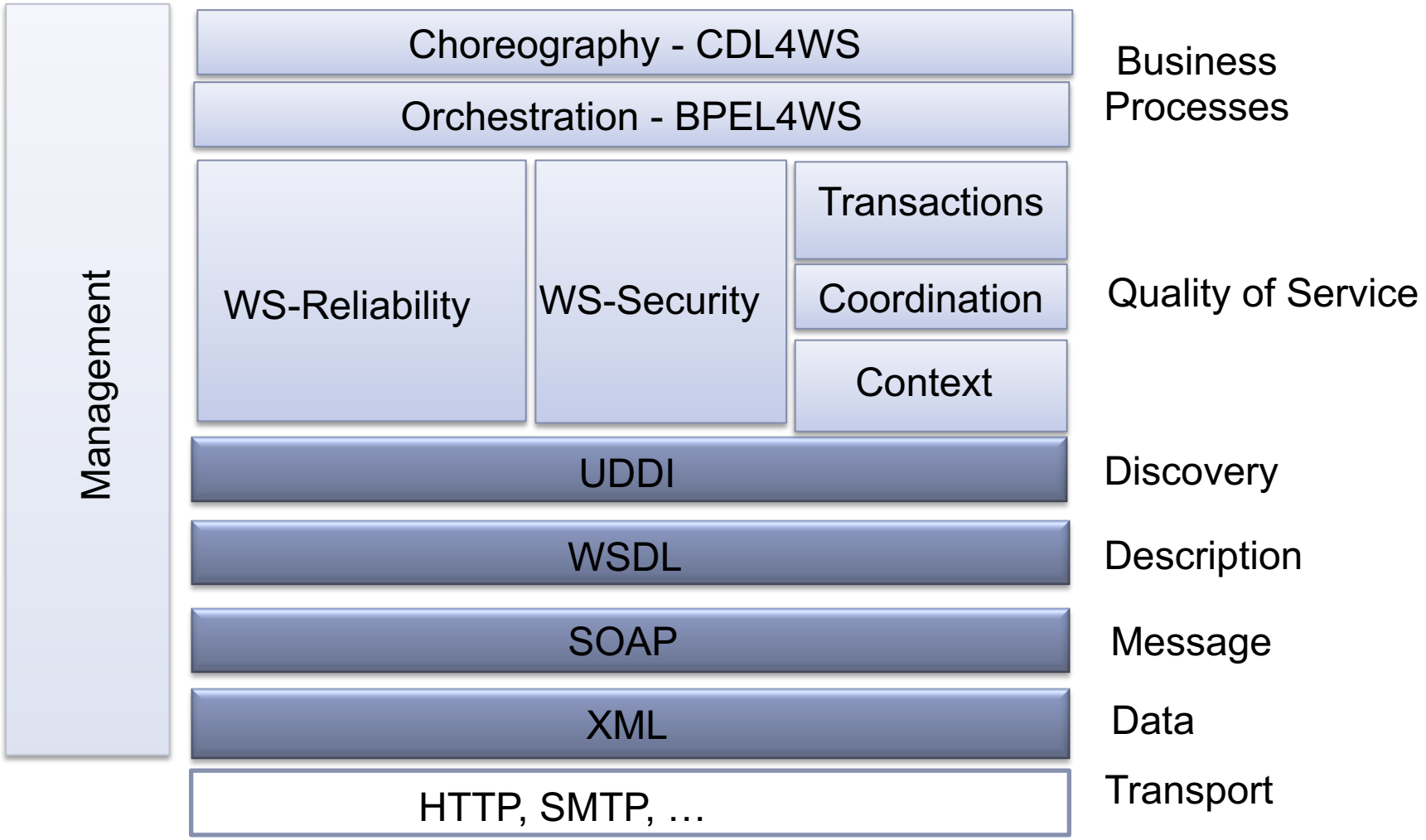
- Serviço de **diretório** para registo e pesquisa dos serviços disponíveis

- Cliente pode procurar **descrição de serviço WDSL** por nome ou atributo
  - Ou aceder diretamente ao URL

Web Service: interação



Web Services standards completos (WS-*)



Web Services: é precisa toda esta complexidade?

RESTful services

Implementação de serviços como alternativa ao SOAP

Desvantagens do SOAP

- Obriga ao uso de XML
- **Desempenho e escalabilidade limitados**
  - Representações em XML são longas, *marshalling/unmarshalling* é pesado
  - **Não permite *caching* de dados do lado do cliente**
- Relativamente difícil de implementar
  - Integração nas linguagens nem sempre é direta

REST standards base

REST na prática

- Clientes que pretendam agir sobre um recurso, recebem **cópia por inteiro** dos dados
  - **Permite *caching* no cliente => melhor escalabilidade**

- Objetos podem ser representados recorrendo a tecnologias **alternativas ao XML**
  - Permite representações mais eficientes
    - JavaScript Object Notation (JSON)

Exemplo de comparação entre XML e JSON

XML:

```
<employees>
    <employee>
        <firstName>John</firstName>
        <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName>
        <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName>
        <lastName>Jones</lastName>
    </employee>
</employees>
```

12% payload data

JSON:

```
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Peter", "lastName":"Jones" }
]}
```

23% payload data

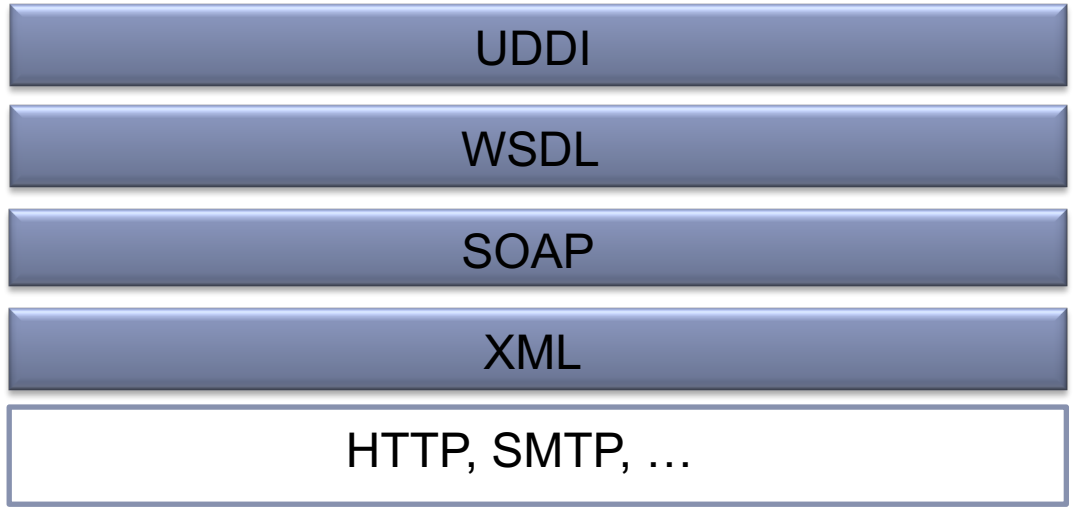
Vantagens JSON vs. XML

- Mais **curto**

- Notação mais próxima das linguagens de programação estilo C
  - Melhor desempenho a serializar/deserializar
  - Melhor integração na linguagem => mais fácil para o programador

SOAP vs REST standards

(SOAP) Web Services



RESTful services

Analogia: SOAP vs REST

SOAP

REST

RPC: resumo

Ideia chave

- Fazer uma invocação remota deve ser tão **simples para o programador** como fazer uma invocação local

Desafios para garantir transparência

- Passagem de parâmetros
  - Dados têm que ser **serializados**
- Execução do procedimento remoto
  - A rede falha, os nós falham
  - **Tolerância a faltas** e notificação de faltas
- Desempenho
  - Depende em grande medida da infraestrutura de comunicação entre cliente e servidor
  - Geralmente mais **lento** do que uma invocação local

Infraestrutura de suporte ao RPC

- No **desenvolvimento**:
  - Uma linguagem de especificação de interfaces
    - *Interface Description Language*, IDL
  - Compilador de IDL
    - Gerador de *stubs*

- Na **execução**:
  - Biblioteca de suporte à execução do RPC (*RPC Run-Time Support*)
    - Registo de servidores
    - *Binding* – protocolo de ligação do cliente ao servidor
    - Protocolo de controlo da execução de RPCs
    - Controlo global da interação cliente-servidor
  - Serviço de Nomes

Bibliografia recomendada

- Secções 4.3 e Cap. 5

- Adicional
  - capítulo 3 como revisão de redes
  - capítulo 13 para serviços de nomes
  - Secções 9.1 a 9.3 para web services