



# FSD

## 1. Introdução ao Sistemas Distribuídos

### Características dos Sistemas Distribuídos

Definição (I)

Definição (II)

Definição (III)

Principais características distintivas dos sistemas distribuídos:

### Principais Desafios dos Sistemas Distribuídos

Heterogeneidade

Sistemas abertos

Segurança

Escalabilidade

Tolerância a falhas

Concorrência

Transparência

Qualidade de serviço

### Exercícios

## 2. Modelos de Sistema

### Modelos

Modelos Físicos

Modelos de Arquitetura

Paradigmas de Comunicação

Comunicação direta vs indireta

Paradigmas de comunicação direta

Paradigmas de comunicação indireta

**Modelo cliente-servidor**

Modelo Peer-to-Peer (P2P)

### Exercícios

Modelos Fundamentais

Modelo de interação

Modelos de falhas

Modelo de segurança

## Exercícios

### 3 - Comunicação Remota entre Processos

#### Sockets

Como endereçar processos

Portas dos Sockets

Endereço dos Sockets

Sincronização no envio e receção de mensagens

Socket Buffers

Sockets para UDP

Sockets para TCP

Sockets em Java

#### Representação externa de dados

Exemplo

CDR Corba

Serialização de objetos em java

#### Resumo

#### Exercícios

### 4.1 - Middleware - Invocação remota entre processos

#### Invocação remota

#### Protocolos pedido-resposta

Protocolos de invocação remota

**TCP vs UDP**

#### Middleware para sistemas distribuídos

Middleware de invocação remota

Chamada de Procedimentos Remotos (RPC)

#### Exercícios

Invocação de métodos remotos (RMI)

#### Exercícios

#### Web services

RPC Web Services

REST Web Services

### 5 - Segurança

#### **Safety vs Security:**

#### **Modelo de ameaças para um sistema distribuído (Unidade temática 2)**

#### Política de Segurança

#### Tipologias de ameaças

#### Ameaças de código móvel

#### Requisitos de segurança

#### Segurança

#### Mecanismos de Segurança

Criptografia

Assinatura Digital

Certificados digitais

#### Transport Layer Security (TLS) Protocol

**Estabelecimento da comunicação (handshake):**

**Fase de comunicação:**

#### HTTP Secure (HTTPS)

Comunicação HTTPS:

#### Exercícios:

# 1. Introdução ao Sistemas Distribuídos

## Características dos Sistemas Distribuídos

O mais importante Não é:

- Decorar definições de sistemas distribuído
- Discutir a fronteira que distingue sistemas distribuídos de sistemas que não distribuídos

O mais importante é reconhecer as propriedades que caracterizam os sistemas distribuídos e perceber as suas implicações (características e desafios fundamentais)

### Definição (I)

“We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.”

“Definimos um sistema distribuído como aquele em que o hardware ou componentes de software localizados em computadores em rede comunicam e coordenam as suas ações apenas passando mensagens.”

### Definição (II)

Um sistema distribuído é um coleção de máquinas autónomas ligadas em rede, e em que cada máquina executa determinados componentes que interagem entre si com vista à prestação de uma funcionalidade computacional que apareça ao utilizador como sendo um sistema integrado.

### Definição (III)

Um Sistema Distribuído é um sistema em que os diversos componentes, situados em diversas máquinas ligadas à rede, comunicam e coordenam as suas ações apenas através da troca de mensagens. Desta definição resultam as seguintes características fundamentais dos sistemas distribuídos:

- concorrência
- ausência de relógio global
- falhas independentes dos componentes

### Principais características distintivas dos sistemas distribuídos:

- Concorrência
- Ausência de relógio global
- Falhas independentes dos componentes

### Principais Desafios dos Sistemas Distribuídos

- Heterogeneidade
- Segurança
- Sistemas abertos
- Escalabilidade

- Tolerância a falhas
- Transparência
- Concorrência
- Qualidade de serviço

## Heterogeneidade

Diverso, não uniforme

Máquinas autónomas ligados em rede → Heterogeneidade

Um sistema distribuído é composto por máquinas autónomas, ou seja, não podemos assumir que são todas iguais, ou sequer que são todas geridas pela mesma entidade

### De onde é que vem a heterogeneidade dos sistemas distribuídos?

- Diferentes sistemas operativos
- Diferentes tipos de computador
- Diferentes tipos de rede
- Componentes implementados em diferentes linguagens de programação
- Implementações autónomas dos componentes

### Quais são as implicações da heterogeneidade para o desenvolvimento de sistemas distribuídos?

- Minimizar pressupostos sobre o ambiente de execução dos componentes do sistema para que estes possam funcionar, se possível, em qualquer ambiente de execução
- Mitigar problemas originados por pressupostos que cada componente pode ter sobre o funcionamento dos outros componentes
  - Diferentes representações de dados, necessidade de versões dos componentes para cada SO,  
....

### Técnicas habituais para lidar com desafios da heterogeneidade dos sistemas

- *Middleware*
  - Tipos de dados comuns
  - Protocolo de comunicação
- Máquina virtuais, e.g. *javascript* ou *java*

## Sistemas abertos

Um sistema aberto permite que novos componentes possam ser desenvolvidos e acrescentados ao sistema por muitas entidades



*Um sistema aberto não tem de ser open source!*

*Um sistema aberto não é um sistema em que qualquer pessoa pode entrar sem precisar de conta!*

### Vantagens de sistemas que podem ser facilmente estendidos por qualquer entidade

- Redução de custos
- Efeito de rede gerado por cada nova entidade que implementa componentes ou os utiliza
- Concorrência entre fabricantes e inovação rápida

- Maior escrutínio do funcionamento do sistema a nível de falhas e de segurança

### Técnicas habituais para criar sistemas abertos

- Protocolos de interação entre componentes são públicos
- Qualquer entidade pode desenvolver a sua própria implementação de componentes do sistema
- Qualquer entidade pode associar novos componentes ao sistema

**Exemplos: DNS, WWW e Facebook**

## Segurança

Num sistema distribuído existem riscos acrescidos em termos de segurança.

Comunicação entre componentes é feita por troca de mensagens.

Mensagens são transmitidas através de um canal de comunicação (a rede) que se tem de assumir que não é seguro.

Comunicação através de um canal não seguro abre a porta para diversos tipos de risco:

- Alguém pode ver as mensagens trocadas entre A e B
- Alguém pode alterar as mensagens trocadas entre A e B
- Alguém se pode fazer passar por A ou B



**Quais os serviços básicos de segurança necessários, então, numa comunicação sobre um canal não seguro?**

- |  |   |
|--|---|
| <ul style="list-style-type: none"> <li>• Confidencialidade</li> <li>• Autenticação</li> <li>• Garantia de disponibilidade (evitar <i>denial-of-service</i>)</li> </ul> | <ul style="list-style-type: none"> <li>• Integridade</li> <li>• Não-repúdio</li> <li>• Segurança de código móvel</li> </ul> |
|--|---|

### Técnicas para lidar com os desafios de segurança

- Diversas tecnologias de criptografia
- Autoridades de certificação
- Protocolos que suportam serviços de segurança
- Políticas de segurança
- Atualização e segurança do software

## Escalabilidade

Um sistema escalável é um sistema que consegue manter um funcionamento efetivo mesmo quando confrontado com aumentos muito grandes no número de recursos e de utilizadores

Os sistemas distribuídos têm à partida um forte potencial para serem altamente escaláveis, mas para isso precisam de lidar com os desafios resultantes do aumento de escala do sistema.

**Quais são ao certo os efeitos que um aumento muito significativo de escala pode ter num sistema distribuído?**

- Efeitos na robustez do sistema
- Efeitos no desempenho do sistema

- Efeitos administrativos
- Efeitos no utilizador
- Efeitos de Heterogeneidade

### Técnicas de escalabilidade em sistemas distribuídos

- Distribuição de responsabilidades entre componentes
- Replicação de componentes para distribuição de carga
- Redução (nas formas de interação)
- Gestão distribuída
- Protocolos abertos

### Tolerância a falhas

Um sistema distribuído é composto por múltiplos componentes, sendo expectável que, ocasionalmente, possam ocorrer falhas em algum desses componentes.

Como num sistema distribuído os componentes estão em máquinas autónomas, diz-se que as **falhas dos componentes são independentes**, ou seja, a falha de um componente não está relacionada com potenciais falhas de outros componentes que, em princípio, continuarão a funcionar

**Falta:** uma falha de um componente significa para o sistema uma falta (desse componente)

**Falha parcial:** Quando ocorre uma falta de um dos seus componentes, o sistema entra em falha parcial se continuar operacional mas com a sua funcionalidade afetada

**Degradação graciosa:** Manter as funcionalidades possíveis quando confrontado com faltas que impeçam uma operação normal.

**Falha do sistema:** Quando perante a falta de um ou mais componentes, o sistema deixa de funcionar como um todo (ficou inoperacional ainda que alguns dos seus componentes possam continuar operacionais)

**Tolerância a faltas:** Um sistema diz-se tolerante a faltas se, perante a falta de um dos seus componentes, puder manter um funcionamento efetivo, ainda que parcial

**Ponto único de falha** (ou ponto crítico): Componente que em caso de falha implica a falha do todo o sistema distribuído, mesmo que os restantes componentes continuem todos operacionais.

Os sistemas distribuídos têm à partida um forte potencial para serem altamente tolerantes a faltas, desde que concebidos para lidar adequadamente com essas faltas.

**Técnicas habituais para lidar com a falta de componentes:**

- Replicação
- Desenhar o sistema sem pontos únicos de falha
- Redundância

### Concorrência

Num sistema distribuído diferentes clientes podem querer aceder concorrentemente aos recursos partilhados

Servidores têm de estar preparados para lidar concorrentemente com múltiplos pedidos que podem ir chegando a qualquer momento, possivelmente de vários clientes

Processos que proporcionam acesso a recursos partilhados têm de estar preparados para garantir a consistência dos dados quando estes são acedidos concorrentemente

## Técnicas habituais para suporte à concorrência

- Servidores *multi-threaded* ou multi-processo
- Recursos partilhados com funções sincronizadas
- Princípios gerais de programação concorrente

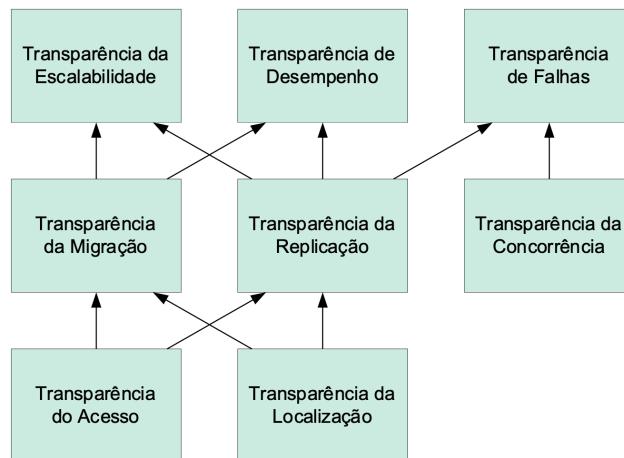
## Transparência

Um desafio dos sistemas distribuídos é o custo em termos de complexidade inerente à própria distribuição do sistema

A transparência de um sistema distribuído traduz-se na capacidade de oferecer as suas funcionalidades como um todo integrado, escondendo do utilizador e, principalmente, dos programadores de aplicações os detalhes da distribuição e separação dos seus componentes.

Permite beneficiar do potencial da distribuição sem necessidade de lidar com os detalhes e complexidade que lhe são inerentes

Acesso	Recursos locais e remotos podem ser acedidos com as mesmas operações
Localização	Recursos podem ser acedidos sem se conhecer a sua localização
Concorrência	Diferentes entidades podem usar recursos comuns sem interferirem
Replicação	Diversas réplicas de um recurso são usadas sem que as entidades que as usem se apercebam do número e localização das réplicas
Faltas	A ocorrência de faltas é disfarçada de modo que as operações possam ser realizadas na mesma
Mobilidade	Recursos podem ser movidos sem afectar o funcionamento
Desempenho	A configuração pode ser mudada para melhorar o desempenho
Escala	O sistema pode ser expandido sem implicar mudanças



**Transparência é tornar invisível o que não seja relevante para a execução de uma determinada tarefa**

*Transparência não é o sistema estar numa caixa transparente e portanto todos os seus detalhes estarem sempre visíveis. É o oposto disso, são os detalhes que não precisam de ser visíveis porque são transparentes!*

**Técnicas habituais:** Organização por camadas e APIs de acesso.

## Qualidade de serviço

Para além dos desafios funcionais, um sistema distribuído também precisa de responder adequadamente a nível de:

- Fiabilidade
- Desempenho
- Adaptabilidade
- Disponibilidade de recursos

**Técnicas habituais para gestão de qualidade de serviço**

- Reserva de recursos
- Comunicações com suporte para QoS
- Service Level Agreements (SLA)

## Exercícios

1. How might the clocks in two computers that are linked by a local network be synchronized without reference to an external time source? What factors limit the accuracy of the procedure you have described? How could the clocks in a large number of computers connected by the Internet be synchronized? Discuss the accuracy of that procedure.

Exemplo do protocolo Cristian (há mais propostas de soluções para este problema)

Medir round trip time entre máquinas A e B ( $t$ )

A envia  $T_A$  para B

B acerta relógio  $T_B = T_A + t$

### Problemas

Contenção das redes locais

atrasos no processamento das mensagens pelos Sos

Atrasos nos routers da Internet

2. A server program written in one language (for example C++) provides the implementation of a BLOB object that is intended to be accessed by clients that may be written in a different language (for example Java). The client and server computers may have different hardware, but all of them are attached to an internet. Describe the problems due to each of the five aspects of heterogeneity that need to be solved to make it possible for a client object to invoke a method on the server object.

Vamos focar então nos 5 aspectos sobre heterogeneidade:

Redes de comunicação;

assumindo que as máquinas estão ligadas a uma intranet (ou à Internet), a heterogeneidade das redes (wi-fi, ethernet, 3G) é resolvida pelos protocolos da Internet

Hardware (máquinas)

representação standard de dados

Sistemas operativos

tradução das operações de envio e receção (pelos compiladores ou máquinas virtuais) para a primitivas dos SOs

Linguagens de programação

- representação standard de dados
- Programadores diferentes
- Cumprir os standards acordados
3. An open distributed system allows new resource sharing services such as the BLOB object of exercise 2 to be added and accessed by a variety of client programs. Discuss in the context of this example, to what extent the needs of openness differ from those of heterogeneity.
- Além de ser necessário definir os standards (ver exercício anterior), é necessário que esses standards coincidam com os standards do sistema aberto (previamente definidos e publicados)
- Os interfaces do serviço BLOB terão de ser publicados
- Se o sistema aberto já concluir objetos do tipo BLOB, a implementação do objeto BLOB terá de cumprir os interfaces previamente publicados
- A publicação de standards permite que partes do sistema sejam desenvolvidas por diferentes vendedores.
4. The INFO service manages a potentially very large set of resources, each of which can be accessed by users throughout the Internet by means of a key (a string name). Discuss an approach to the design of the names of the resources that achieves the minimum loss of performance as the number of resources in the service increases. Suggest how the INFO service can be implemented so as to avoid performance bottlenecks when the number of users becomes very large.
- Esquema de nomes hierárquicos. Exemplo A.B.C
- Para conseguir responder a um aumento muito grande de utilizadores, poder-se-ia repartir os recursos por diferentes servidores em que cada servidor é responsável por recurso que iniciam em A, B, etc. cada um dos níveis poderá ainda dividir a procura, usando a segunda letra.

---

## 2. Modelos de Sistema

### Modelos

Sistemas distribuídos podem ser muito variados e diversos mas há muitas propriedades comuns que originam desafios semelhantes

Modelos proporcionam uma forma simplificada e abstrata, mas também consistente, de representar aspectos comuns do funcionamento dos sistemas distribuídos.

#### Modelos físicos

- Representam dispositivos concretos e as redes que os ligam

#### Modelos arquiteturais

- Descrevem o sistema na perspetiva das tarefas desempenhadas pelos seus componentes e as interações entre esses componentes

#### Modelos fundamentais

- Descrevem abstrações para problemas fundamentais de sistemas distribuídos

- Facilitam o desenvolvimento de soluções adequadas para problemas complexos

## Modelos Físicos

Representam dispositivos concretos e as redes que os ligam

Foco é no hardware, mas abstraindo dos detalhes tecnológicos da rede ou dos computadores envolvidos

Modelos fortemente marcados pelas tendências tecnológicas

*Generations of distributed systems - Gerações de sistemas distribuídos:*

Sistemas distribuídos	Early	Internet-scale	Contemporary
Escala	Pequena	Grande	Ultra-grande
Heterogeneidade	Limitada (configurações tipicamente e relativamente homogêneas)	Significativo em termos de plataformas, linguagens e middleware	Dimensões adicionais introduzidas, incluindo estilos radicalmente diferentes de arquiteturas
Openness	Não é uma prioridade	Prioridade significativa com variedade de standards introduzidos	Grande desafio de pesquisa com os standards existentes ainda incapazes de abraçar um sistema complexo
Quality of service	Na sua infância	Prioridade significativa com a range de serviços introduzidos	Grande desafio de pesquisa com os standards existentes ainda incapazes de abraçar um sistema complexo

## Modelos de Arquitetura

Num sistema distribuído a funcionalidade do sistema está repartida por várias entidades

- Quais são as entidades que comunicam entre si para formar o sistema distribuído?
- Qual o paradigma de comunicação em que se baseia essa comunicação?
- Que papéis e responsabilidades é que cada componente tem na arquitetura?

A arquitetura de sistema define a sua estrutura em termos de distribuição de responsabilidade pelos componentes e as suas interligações.

## Paradigmas de Comunicação

### Paradigma de comunicação direta

- Entidades comunicam diretamente entre si
- *Strongly coupled* (tempo/espaço partilhados)
- Paradigmas mais comuns: protocolos pedido-resposta, invocação de procedimentos remotos, invocação de métodos remotos

### Paradigma de comunicação indireta

- Entidades comunicam através de intermediário
- *Loosely coupled* (tempo/espaço separados)
- Paradigmas mais comuns: Publish-subscribe, Message-Oriented middleware (MOM), Tuple Spaces

## Comunicação direta vs indireta

Num paradigma de comunicação direta, os intervenientes na comunicação trocam explicitamente mensagens entre si

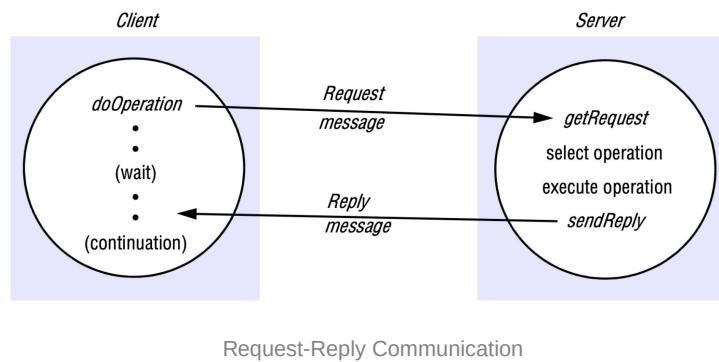
- Entidades têm de saber com quem estão a trocar mensagens (*space coupled*)
- Entidades têm de estar ambas ativas no momento da troca de mensagens (*time coupled*)

Abordagem alternativa é utilizar paradigmas de comunicação indireta em que a comunicação é feita com base numa entidade que intermedia a comunicação entre as entidades envolvidas numa colaboração

## Paradigmas de comunicação direta

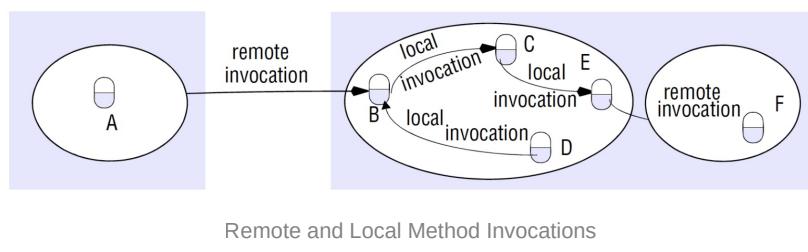
### Protocolo pedido-resposta

Tipicamente envolve a troca de duas mensagens entre dois elementos do sistema: pedido de execução de operação e resultado de operação



### Invocação de métodos remotos

Abstração de invocação remota aplicada a métodos de objetos



## Paradigmas de comunicação indireta

Propriedades comuns quando comunicação é feita através de um intermediário:

- **[loose coupling]** Maior independência entre componentes, diminui pressupostos, permite maior heterogeneidade nas implementações
- **[Space uncoupling]** Emissor pode enviar mensagem mesmo sem saber quem é que a vai receber (ou se alguém a vai receber). Mensagem vai sempre para entidade intermediária que a faz chegar aos receptores adequados.
- **[Time uncoupling]** Emissor pode enviar mensagem mesmo que quem vai receber não esteja ativo naquele momento. Mensagem é guardada pela entidade intermediária e pode ser recuperada mais

tarde.

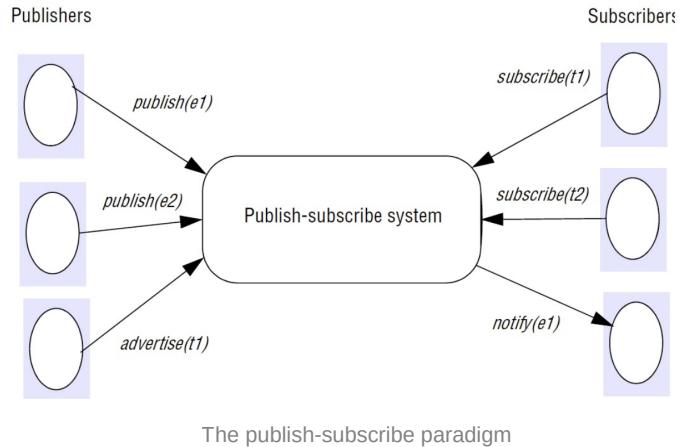
### Aplicações de comunicação indireta

Modelos do comunicação indireta são especialmente adequados nas situações que:

- Seja expectável haver mudanças frequentes na composição, configuração ou funcionamento do sistema
- Exista forte mobilidade de componentes, que em função disso vão aparecendo e desaparecendo
- Exista informação para disseminar por um conjunto alargado e potencialmente muito variável de destinatários
- Exista necessidade de uma autoridade intermédia

### Produtor-Subscritor

- Subscritores registam interesse em receber determinadas mensagens
- Produtores enviam mensagens estruturadas para o sistema
- Mensagens são enviadas para subscritores interessados

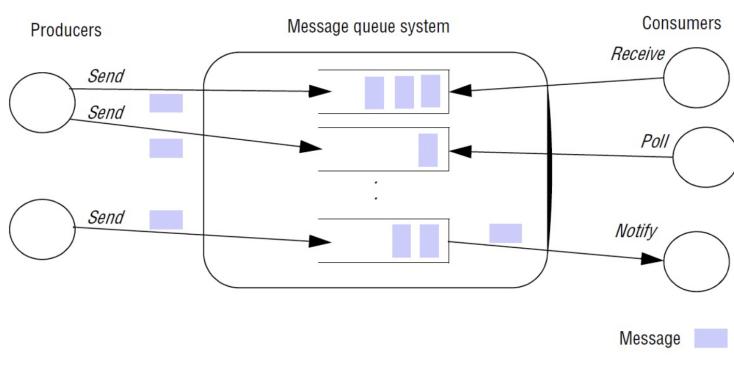


Principais áreas de aplicação

- Lives feeds de dados com vários consumidores
- Sistemas colaborativos baseados na partilha de eventos
- Sistemas de sensores e monitorização

### Message Queues

- Produtores enviam para uma queue
- Receive bloqueia à espera que cheguem mensagens
- Poll devolve mensagem se existir, ou aviso de que não há mensagens
- Notify avisa que existe mensagem para ler



## *Message-Oriented middleware (MOM)*

Principal área de aplicação é na interoperabilidade entre sistemas organizacionais

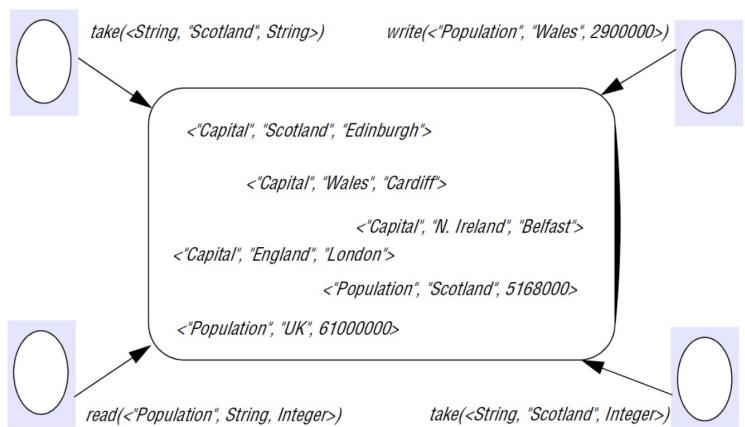
Mensagens usam formatos normalizados para transportar os dados de modo a serem independentes de vários sistemas

MOM oferecem abstração, interoperabilidade, fiabilidade e persistência

Exemplos: IBM WebSphere MQ, Amazon SQS, RabbitMQ, JMS

## Tuple spaces

- Processo coloca tuplos num espaço de tuplos (write)
- Outros processos podem ler ou remover esses tuplos (read)
- Tuplos não tem destino mas têm propriedades que permitem aplicar múltiplos filtros na sua seleção (take)



The tuple space abstraction

Tuplos são sequências de propriedades

Tuplos podem ser pesquisados em função das suas propriedades

Espaço de tuplos oferece persistência

Normalmente é um modelo centralizado que conceptualmente pode ser visto como um paradigma de memória partilhada

Exemplos: JavaSpaces, IBM's TSpaces

## **Modelos de arquitetura:**

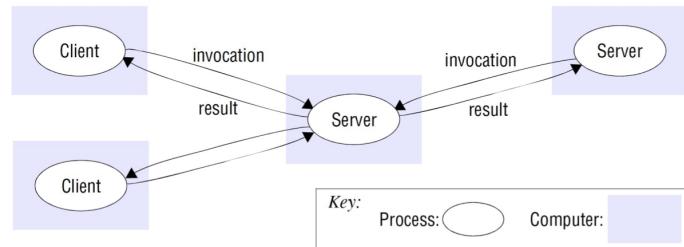
Que papéis e responsabilidades é que cada componente tem na arquitetura?

- Modelo cliente-servidor
- Modelo peer-to-peer

## **Modelo cliente-servidor**

- servidores: processos que oferecem serviços

- clientes: processos que requerem serviços



Clients invoke individual servers

Os servidores oferecem uma interface bem definida e que é partilhada por todos os clientes

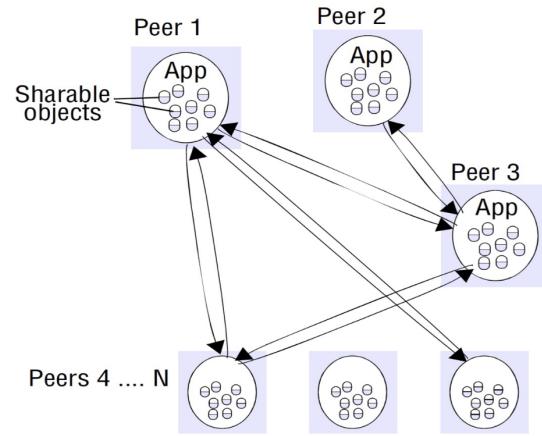
Existe uma especificação de um protocolo de comunicação que define exatamente que tipo de pedidos é que o servidor aceita e quais as respostas que deve gerar

Os termos cliente e servidor correspondem aos papéis definidos à partida para cada um dos elementos do sistema

Um processo servidor, pode ele próprio funcionar ocasionalmente como cliente de outros serviços. Por exemplo, um servidor web pode consumir informação de outros serviços para gerar as suas páginas

### **Modelo Peer-to-Peer (P2P)**

- Semelhante ao cliente-servidor em quase tudo
- A grande diferença é que não existem à partida papéis diferenciados para os componentes
- Todos os elementos desempenham um papel semelhante e interagem de forma cooperativa sem distinção entre clientes e servidores
- Qualquer componente pode ter a iniciativa de pedir serviços a qualquer outro



### **Vantagens do sistema peer-to-peer**

Escalabilidade do sistema não é limitada pela capacidade de servidores.

Cada novo peer acrescenta capacidade

Maior redundância pode permitir maior resiliência

### **Desvantagens**

Maior dificuldade de controlo

Dificuldade de encontrar peers obriga quase sempre a modelos híbridos que combinam cliente-servidor com peer-to-peer

## Exercícios

1. What problems do you foresee in the direct coupling between communicating entities that is implicit in remote invocation approaches? Consequently, what advantages do you anticipate from a level of decoupling as offered by space and time uncoupling?

O cliente está intrinsecamente ligado ao servidor e vice-versa, o que se reflete em termos de lidar com falhas.

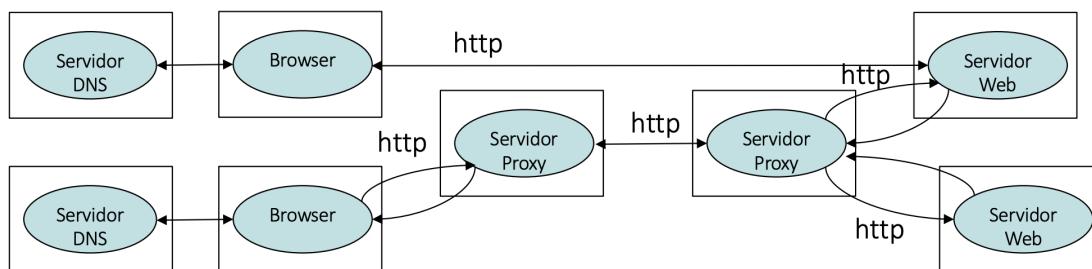
O servidor e o cliente têm de estar ativos simultaneamente.

A substituição por um servidor redundante pode ser complexa

Space decoupling permite mais liberdade em lidar com alterações, por exemplo quando um novo componente começa a atender pedidos.

Time decoupling permite que as entidades comuniquem mesmo que não estejam simultaneamente ativas.

2. Describe and illustrate the client-server architecture of one or more major Internet applications (for example, the Web, email or netnews).



Os Browsers são clientes de servidores de DNS (DNS) e dos servidores Web (HTTP).

Algumas redes de organizações (intranets) são configuradas com servidores proxy. Os proxy podem funcionar no lado cliente e reduzem latência e tráfego de rede, funcionando como caches; ou do lado do servidor, funcionando como um ponto de segurança, podendo reduzir a carga no servidor.

## Modelos Fundamentais

Modelos simplificados do sistema que incluem apenas os conceitos necessários para entender um determinado tipo de comportamento do sistema, e.g. segurança, interação, falhas.

Conjunto de abstrações que nos permitem abordar problemas semelhantes com soluções semelhantes

Apresentam explicitamente os pressupostos fundamentais do sistema de modo a que as soluções e propriedades definidas para o modelo sejam aplicáveis a qualquer sistema no qual esses pressupostos sejam válidos

Generalizam o que pode ou não ser feito, tendo em conta os pressupostos identificados, e definindo propriedades ou algoritmos que permitam atingir determinados comportamentos.

- **Modelo de interação** lida com as dificuldades em estabelecer limites de tempo na troca de mensagens
- **Modelo de falhas** faz uma especificação dos vários tipos de falhas potenciais e define os requisitos de comunicação fiável

- **Modelo de segurança** especifica os vários tipos de ameaças à comunicação e define o conceito de canal seguro.

## **Modelo de interação**

A velocidade com que um processo progride na sua execução e o tempo de transmissão de uma mensagem entre processos não podem ser geralmente previstos.

**Existem fundamentalmente dois fatores que afetam os processos num sistema distribuído:**

- O desempenho dos canais de comunicação entre processos
- Não é possível manter uma noção global de tempo

**Duas variantes de modelos de interação:**

- **Sistemas distribuídos síncronos**, define pressupostos sobre tempo, como por exemplo limites para o tempo entre o envio e receção de mensagens numa rede
- **Sistemas distribuídos assíncronos**, não define pressupostos sobre tempo

Sistemas distribuídos baseados na Internet têm de assumir um modelo assíncrono.

Não são definidos limites para:

- velocidade ed execução de processos
- atrasos na transmissão de mensagens
- taxas de *clock drift*

## **Principais pressupostos do modelo de interação (assíncrono)**

- Modelo de interação reflete o facto de que a interação entre processos vai ser afetada pelos atrasos nas mensagens, podendo originar problemas de sincronização
- Mesmo quando não se considera a possibilidade de falhas de comunicação, vão existir problemas de sincronização que envolvam coordenação temporal
- Tempo não é um proxy perfeito para a ordenação de eventos a nível do sistema como um todo

Normalmente, o que é verdadeiramente importante num sistema não é o tempo absoluto em que as coisas aconteceram, mas sim a ordem em que aconteceram.

- Tempo real vs Tempo Lógico
- Relação temporal vs Relação causal ( $A \rightarrow B$ )
- Relógios lógicos como solução para ordenação de eventos sem utilização de tempo

**Relação causal** ( $A \rightarrow B$ ) significa que A ocorreu antes de B:

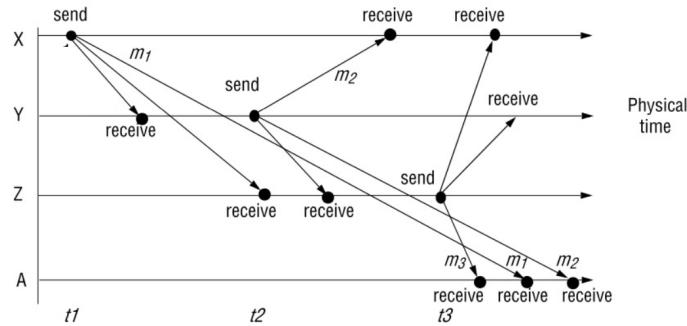
1. A e B são eventos no mesmo processo e A foi observado antes de B.
2. A é um evento de envio de uma mensagem num processo e B é o evento correspondente à chegada dessa mensagem a outro processo.
3. Se  $A \rightarrow B$  e  $B \rightarrow C$ , então  $A \rightarrow C$ , ou seja, a relação de causalidade é transitiva

## **Exemplo mensagens email**

Considerem a seguinte troca de mensagens entre um grupo

de utilizadores de email, X, Y, Z e A, de uma *mailing list*. E considerem que:

1. O utilizador X envia uma mensagem com o assunto Meeting.
2. Os utilizadores Y e X respondem enviando uma mensagem com o assunto *Re: Meeting*.



Inbox:		
Item	From	Subject
23	Z	Re: Meeting
24	X	Meeting
25	Y	Re: Meeting

Real-time ordering of events

### Algoritmos de ordenação causal

Relógios lógicos [Lamport]

- Cada processo mantém um contador que é incrementado a cada evento
- Ao enviar uma mensagem o valor do contador é enviado também
- Ao receber uma mensagem (e antes de incrementar o valor) o contador local é alterado de modo a corresponder ao máximo de (contador local, contador na mensagem)

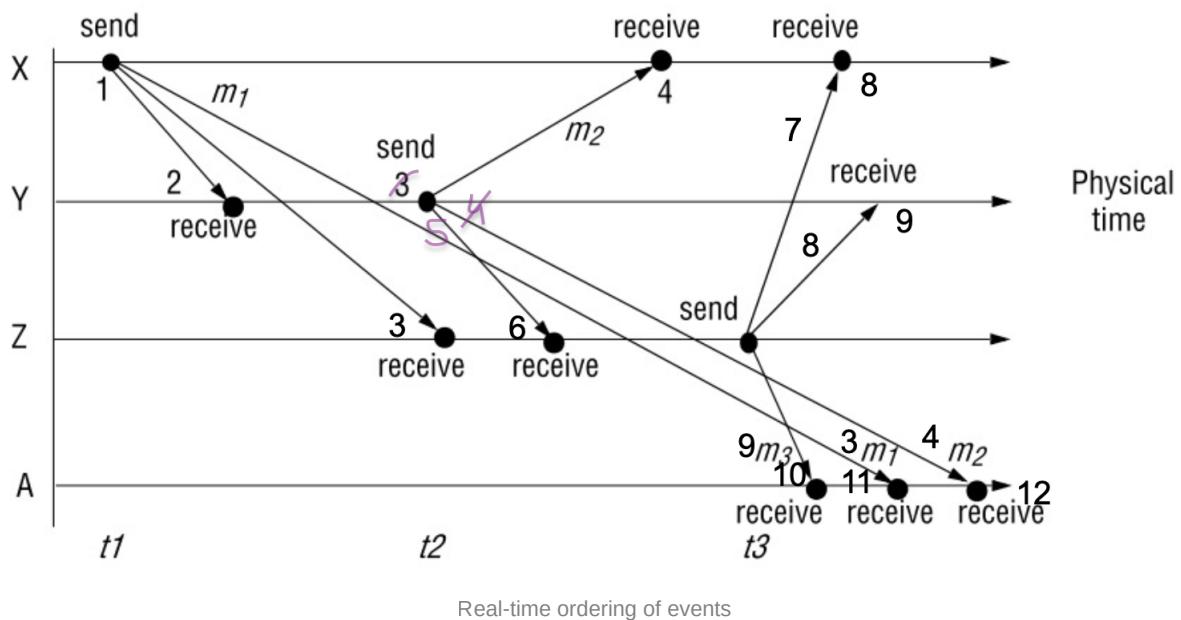
Totally ordered logical clocks

- Inclui identificadores dos vários processos e regras de ordenação em caso de não causalidade

Vector clocks

- Implica troca de vetor de contadores dos vários processos

### Exemplo de relógios lógicos



### Não existe agora:

Apesar dos ambientes computacionais serem normalmente descritos como virtuais, eles têm de funcionar no mundo físico, cujos desafios não podem ignorar

Porque é que não podemos usar o tempo como um referencial comum em Sistemas Distribuídos?

### Modelos de falhas

**Em sistemas reais, podem ocorrer falhas a vários níveis**

- Como modelar essas falhas?
- Como lidar com essas falhas?

O modelo de falhas define as formas como essas falhas podem ocorrer de modo a proporcionar um entendimento mais claro dos efeitos dessas falhas

Num sistema distribuído, as falhas podem ocorrer a nível dos processos e dos canais de comunicação.

Num sistema distribuído assíncrono podemos considerar 2 tipos de falhas:

- **Falhas por omissão:** Um processo ou canal de comunicação não faz algo que era suposto fazer
  - Exemplo: Canal de comunicação deixa de entregar no destino uma mensagem que lhe foi entregue: '*dropping messages*'
- **Falhas arbitrárias** (ou bizantinas): Um processo ou canal envia valores incorretos ou respostas erradas a pedidos
  - Exemplo: Canal de comunicação corrompe os dados de uma mensagem
  - Exemplo: Servidor envia uma resposta incorreta

### Deteção de falhas

Falhas por omissão são facilmente detetáveis mas pode ser difícil identificar o ponto de falha

Não se recebeu uma resposta a um pedido mas pode existir muitas causas para isso, a nível dos processos, dos sistemas operacionais ou dos canais de comunicação

Falhas arbitrárias são muito difceis de detetar.

Fez-se um pedido, recebeu-se uma resposta. Como distinguir uma resposta errada de uma resposta genuína?

Omission and arbitrary failures:

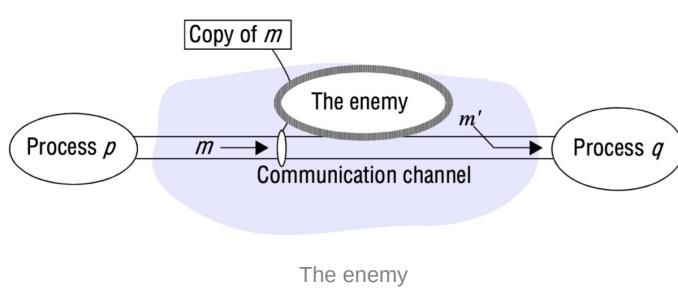
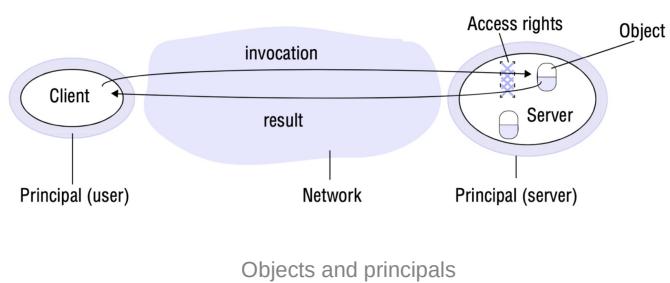
Class of failure	Affects	Description
Fail-stop	Process	O processo é interrompido e permanece interrompido. Outros processos podem detetar este estado
Crash	Process	O processo para e permanece parado. Outros processos podem não ser capazes de detectar este estado
Omission	Channel	Uma mensagem inserida num buffer de mensagem de saída nunca chega ao buffer de mensagem de entrada do final do pedido
Send-omission	Process	Um processo completa uma operação de send, mas a mensagem não é colocada no seu buffer de mensagens de saída
Receive-omission	Process	Uma mensagem é colocada no buffer de mensagem de entrada de um processo, mas esse processo não a recebe
Arbitrary (Byzantine)	Process or channel	Processo / channel exibe comportamento arbitrário: pode enviar / transmitir mensagens arbitrárias em temporizador arbitrário ou cometer omissões; um processo pode parar ou dar um step incorreto

## Modelo de segurança

Devido à sua inerente distribuição, os sistemas distribuídos estão expostos a um número potencialmente muito grande de riscos de segurança, por agentes internos ou externos

O modelo de segurança define as possíveis formas desses ataques e proporciona uma base para a análise das ameaças e para a conceção de um sistema capaz de resistir a essas mesmas ameaças

- Principal- Um entidade que pode ser identificada e cuja identidade pode ser verificada através de um processo de autenticação
- Pessoas (Bob e Alice) ou aplicações



- Pode fazer-se passar por um dos processos que legítimamente fazem parte do sistema
- Um ataque pode ser realizado a partir de um computador que faz legitimamente parte do sistema

- O Inimigo representa ameaça para os processos e para os canais de comunicação. Pode enviar qualquer mensagem para qualquer processo e ler, copiar ou alterar qualquer mensagem entre um par de processos

ou de um computador ligado de forma não autorizada

### Técnicas para segurança

- Autenticação
- Criptografia
- Canais seguros



## Exercícios

- Consider a simple server that carries out client requests without accessing other servers. Explain why it is generally not possible to set a limit on the time taken by such a server to respond to a client request. What would need to be done to make the server able to execute requests within a bounded time? Is this a practical option?

A taxa de chegada de pedidos de clientes a um servidor é imprevisível.

Os recursos podem não ser suficientes para limitar a execução do pedido a um determinado limite de tempo

Para poder executar pedidos num tempo limitado, é necessário limitar o número de pedidos; aumentar a capacidade de processamento; ou replicar o serviço.

Estas soluções podem ter custos elevados, tanto ao nível financeiro como de esforço de programação, e no caso da replicação, é necessário manter as cópias consistentes

- For each of the factors that contribute to the time taken to transmit a message between two processes over a communication channel, state what measures would be needed to set a bound on its contribution to the total time. Why are these measures not provided in current general-purpose distributed systems?

Fatores que influenciam a comunicação:

o tempo gasto pelas primitivas de envio de mensagem e receção de mensagem do sistema operativo para completar o envio e receção de uma mensagem  
tempo gasto no acesso à rede de comunicação

Medidas

mais recursos; reserva de recursos de rede  
mais processamento

Problemas

heterogeneidade de redes e servidores

- Consider two communication services for use in asynchronous distributed systems. In service A, messages may be lost, duplicated or delayed and checksums apply only to headers. In service B, messages may be lost, delayed or delivered too fast for the recipient to handle them, but those that are delivered arrive with the correct contents. Describe the classes of failure exhibited by each

service. Classify their failures according to their effects on the properties of validity and integrity. Can service B be described as a reliable communication service?

#### Falhas serviço A

Omissão (falha de mensagens)

Arbitrarias (mensagens corrompidas ou mensagens duplicadas)

#### Falhas serviço B

Omissão (falha de mensagens ; dropped messages)

A robustez do canal de comunicação é medida pelas propriedades de Validade (qualquer mensagem no buffer de saída será entregue) e Integridade (a mensagem recebida corresponde à mensagem enviada e uma mensagem não é entregue duas vezes).

#### Serviço A

Validade e Integridade não se verificam

#### Serviço B

Integridade é verificada; Validade não se verifica

---

## 3 - Comunicação Remota entre Processos

### Sockets

Abstração baseado no paradigma de troca de mensagens

Um *socket* representa um ponto terminal na comunicação entre dois processos

Fornece um API para o TCP e outro para o UDP

Criado originalmente como API do BSD UNIX

Disponível em quase todos os tipos de sistemas operativos

Ideia inspirada nos *sockets* dos sistemas de comutação telefónica com operador humano

### Como endereçar processos

Para processos poderem comunicar entre si, têm de poder identificar qual o processo a que uma mensagem se destina

Utilizar um nome (ou número) de processo seria uma forma de endereçamento explícita mas pouco flexível

- Uma alteração na identificação de um processo implica a alteração de todas as referências para esse processo

*Sockets* utilizam um conceito de endereçamento implícito

- Mensagens são endereçadas para o endereço de um *objeto*, designado porta (ou porto num tradução mais literal do termo em Inglês: *port*)
- Os processos referenciam-se implicitamente:
  - send (A, message) – envia uma mensagem para porta A

- Receive (A, message) – retira uma mensagem da porta A

## Portas dos Sockets

Múltiplos processos podem enviar mensagens para a mesma porta

Apenas pode existir um processo associado a cada porta

- Porta onde mensagem chega determina processo que a vai receber

Cada porta é identificada por um número entre 1 e 65,535

As portas 1 a 1023 (*well-known ports*) são reservadas a serviços comuns

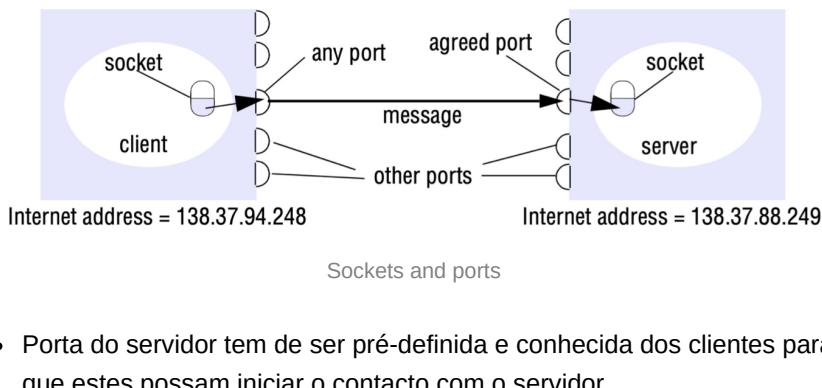
- 25 SMTP, 110 – POP, 80 – HTTP, 20 – FTP

Cada porta está associada a um protocolo (TCP ou UDP)

- Mesmo número pode estar a ser usado simultaneamente por protocolos TCP e UDP (Porta 20 do UDP é diferente de Porta 20 do TCP)

## Endereço dos Sockets

- Uma ligação por sockets envolve 2 endereços IP e duas portas, uma em cada um dos processos
- Porta do cliente que inicia contacto não é relevante



- Porta do servidor tem de ser pré-definida e conhecida dos clientes para que estes possam iniciar o contacto com o servidor

## Sincronização no envio e receção de mensagens

Processos em comunicação precisam de sincronizar as suas trocas de informação

- *Blocking send* - emissor bloqueia até a mensagem ser recebida com sucesso pelo receptor (primitiva síncrona)
- *Nonblocking send* - emissor continua a sua execução logo após o envio da mensagem (primitiva assíncrona)
- *Blocking receive* - o processo receptor bloqueia até a chegada de uma mensagem (primitiva síncrona)
- *Nonblocking receive* - retorna imediatamente ao processo receptor caso existam mensagens ou não (primitiva assíncrona)

Normalmente é utilizada o *nonblocking send* e o *blocking receive*.

## Socket Buffers

Cada porta (ou *mailbox*) pode suportar uma fila de mensagens

Este processo de *buffering* facilita sincronização

- Capacidade nula (0 mensagens) - O processo emissor bloqueia até a mensagem ser recebida com sucesso pelo recetor
- Capacidade limitada (tamanho finito de n mensagens) - O processo emissor terá de bloquear se a fila está cheia.
- Capacidade ilimitada (tamanho infinito) - O processo emissor nunca bloqueia.

## Sockets para UDP

API para o protocolo de transporte UDP

- Serviço não fiável de entrega de datagramas

Cada mensagem é uma entidade autónoma que carrega consigo o destino onde deve ser entregue

Normalmente um *non-blocking send* e um *blocking receive*

Aplicações têm de ter os seus próprios mecanismos de deteção de perda de *datagramas*

## Sockets para TCP

API para o protocolo de transporte TCP

- Serviço fiável
- Orientado à ligação
- connect , send , receive , close

Normalmente: *non-blocking send* e *blocking receive*

Fornece uma paradigma de *stream*

Abstração da *stream* esconde os seguintes aspetos da comunicação

- O tamanho e número das mensagens
- Mensagens perdidas, duplicadas, trocadas
- Controlo de fluxo
- Destino das mensagens

## Sockets em Java

Na linguagem Java também existe um API para acesso ao serviço de comunicações baseado no paradigma dos sockets

Superta sockets TCP e UDP

Package java.net

### **Principais Classes do package *java.net*:**

InetAddress  
ServerSocket  
Socket  
DatagramPacket  
DatagramSocket

MulticastSocket

URL

HttpURLConnection

**Classes auxiliares:**

InetAddress

    InetAddress getByName(), getLocalHost()

    String getHostName(), getHostAddress()

Java class UR

    URL u = new URL("http://www.dsi.uminho.pt");

    String getProtocol(), getHost(), getFile(), getRef()

    int getPort()

**Envio de Datagramas UDP:**

Classe DatagramPacket

    Representa um datagrama

    DatagramPacket packet = new DatagramPacket(message, msglen, address, port);

Classe DatagramSocket

    Representa um socket UDP

    DatagramSocket socket = new DatagramSocket();

    DatagramSocket socket = new DatagramSocket(porta);

    Permite enviar datagramas

        socket.send(packet);

    Permite receber datagramas

        socket.receive(packet);

**UDP client sends message and gets a reply:**

```

import java.net.*;
import java.io.*;
public class UDPCClient{
    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();}
    }
}

```

### UDP server receives requests and sends them back to clients

```

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.out.println("Socket: " + e.getMessage());
        }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}

```

### Classe Socket:

Utilizada para ligações TCP (cliente)

Criar um socket e estabelecer uma ligação

```
Socket s = new Socket("www.dsi.uminho.pt",80);
```

### Métodos

InetAddress getInetAddress(), getLocalAddress()

int getPort(), getLocalPort()

InputStream getInputStream()

```
OutputStream getOutputStream()
```

#### **Classe ServerSocket:**

Utilizado por processos servidores em ligações TCP

Criar um socket servidor e associá-lo a uma porta

```
ServerSocket ss = new ServerSocket(porta);
```

Aguardar o estabelecimento de ligações (bloqueia)

```
Socket accept() throws IOException
```

Quando existe um pedido de ligação é criado um socket que depois irá ser usado para a comunicação (getInputStream() e getOutputStream())

Múltiplos clientes podem-se ligar à mesma porta e se o servidor for multi-threaded pode tratá-los em simultâneo.

```
Ligação = IP_cliente+Porta_cliente+IP servidor+Porta_Servidor
```

#### **Cliente TCP em Java:**

Criar um socket para estabelecer uma ligação para a máquina e porto pretendidos

Associar ao socket streams de entrada e/ou saída

Ler e escrever para o socket através das streams

Fehchar o socket

#### **Acesso via sockets ao serviço TCP (cliente):**

```
import java.net.*;
import java.io.*;
public class TCPClient {

    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);
            String data = in.readUTF();
            [...]
        }
    }
}
```

#### **Servidor TCP em Java:**

Criar um ServerSocket

Aguardar pelo aparecimento de ligações

Criar um socket para cada nova ligação

Obter a referência das streams de entrada e/ou saída

Ler e escrever para o socket através das streams

Fechar o socket

**TCP server makes a connection for each client and then echoes the client's request:**

```
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e)
        {System.out.println("Listen:"+e.getMessage());}
    }
}

class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream( clientSocket.getInputStream());
            out =new DataOutputStream( clientSocket.getOutputStream());
            this.start();
        } catch(IOException e)
        {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
        try {                                // an echo server
            String data = in.readUTF();
            out.writeUTF(data);
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());}
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
        } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
    }
}
```

## Representação externa de dados

Os processos, durante a sua execução, utilizam diversas estruturas de dados que são suportadas pelas linguagens de programação

*Objectos, Strings, arrays, etc...*

Indicação de um tipo na instanciação das variáveis define a sua representação interna

Um sistema distribuído implica transmissão de dados, mas Sockets só permitem a transmissão de mensagens de sequências de bytes

Para os dados poderem ser transferidos pela rede, precisam de ser transformados em sequências de bytes e depois reconstruídos ao chegar ao destino

A representação externa de dados especifica a forma como os processos representam em bytes os dados que têm de enviar nas mensagens que vão trocar entre si

Diferentes computadores ou linguagens de programação podem usar representações internas de dados diferentes

- Número de bytes usado para representar a informação
- Formato de representação, e.g. *strings*, *floating point*
- Códigos de caracteres
- Ordenação dos bytes: *big-endian*, *little-endian*

Estes diferentes formatos não podem ser usados diretamente para comunicação entre processos

## Exemplo

Um processo pretende enviar para outro uma mensagem com o ano de nascimento de uma pessoa e o seu nome

Qual a sequência de bytes adequada para representar esses dois valores?

Como é que se garante que o receptor, sabendo que vai receber um ano e um nome, consiga interpretar corretamente o conteúdo dos bytes que vai receber para poder reconstruir a informação enviada?

```
private int year=1934;  
private String name="Smith";
```

### Serialização / Marshalling / Coding

- Transformar um conjunto de estruturas internas de dados numa sequência de bytes adequada para transmissão pela rede

### Desserialização / Unmarshalling / Decoding

- reconstruir as estruturas de dados a partir de uma sequência de bytes transmitida por uma mensagem

Estes processos podem ser feitos com código próprio, com bibliotecas ou de forma transparente por software de *middleware*.

Processo é em tudo idêntico ao processo de guardar estruturas de dados em ficheiros para posteriormente as recuperar

A representação externa de dados é um formato comum de serialização de dados

Permite a transferência de dados entre sistemas com representações internas muito diversas

Entidades que vão comunicar devem acordar previamente num formato externo a utilizar por ambas

Normalmente os formatos não incluem informação sobre o tipo de dados; este terá sido publicado na interface do serviço

Exemplos:

- *eXternal Data Representation* (XDR)
- XML documents e schemas

- Corba Common Data Representation (CDR)
- JAVA Serialization formats
- JSON

## CDR Corba

CORBA CDR é um formato de representação externa de dados definido para o *middleware* (objetos distribuídos) CORBA 2.0 [OMG 2004a].

CDR consegue representar todos os tipos de dados que podem ser usados como argumentos e resultados de invocações remotas.

O CDR consiste em 15 tipos primitivos, de entre os quais: *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit) e um conjunto de tipos compostos.

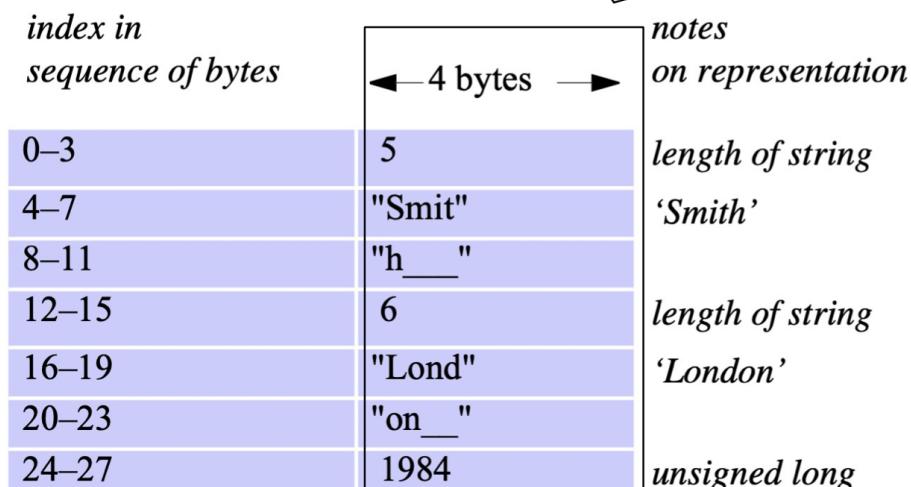
### CORBA CDR for constructed types:

Type	Representation
sequence	comprimento (unsigned long - longo sem sinal) seguido por elementos em ordem
string	comprimento (unsigned long - longo sem sinal) seguido por caracteres em ordem (também pode ter caracteres grandes)
array	elementos da matriz em ordem (nenhum comprimento especificado porque é fixo)
struct	na ordem de declaração dos componentes
enumerated	unsigned long (os valores são especificados pela ordem declarada)
union	digita a tag seguida pelo membro selecionado

### Exemplo:

#### CORBA CDR message

mensagem



The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

## Serialização de objetos em java

Objeto é uma instância de uma classe (com estado)

Objeto que implementa interface *serializable* significa que pode ser *serialized*

- Apenas o estado é serializado
- Assume-se que a classe existe no destino

Para serializar um objeto escreve-se o objeto numa *stream*, associada a um *socket* ou a um ficheiro

Referências podem serem serializadas, mas os objetos referidos tem de ser incluídos e referências tem de ser transformados em *handles* internos

Para serializar um objeto em java, cria-se um objeto (*stream*) da classe *ObjectOutputStream* e escreve-se o objeto na *stream*

#### **Exemplo serialização:**

Exemplo de serialização (*marshalling*) em Java com base nas classes *Stream*

```
FileOutputStream ostream = new FileOutputStream("t.tmp");
ObjectOutputStream p = new ObjectOutputStream(ostream); p.writeInt(12345);
p.writeObject("Today");
p.writeObject(new Date()); p.flush(); ostream.close();
```

Exemplo de serialização inversa (*unmarshalling*) em Java com base nas classes *Stream*

```
FileInputStream istream = new FileInputStream("t.tmp");
ObjectInputStream p = new ObjectInputStream(istream);
int i = p.readInt();
String today = (String)p.readObject();
Date date = (Date)p.readObject();
istream.close();
```

JAVA assume que deve ser possível ler um objeto serializado em relação ao qual não se conhece o tipo.

Formato de serialização inclui indicação de qual a classe que se encontra serializada

Isto permite carregar a classe adequada

<i>Serialized values</i>				<i>Explanation</i>
Person	8-byte version number		h0	<i>class name, version number</i>
3	int year	java.lang.String name	java.lang.String place	<i>number, type and name of instance variables</i>
1984	5 Smith	6 London	h1	<i>values of instance variables</i>

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

*Person p = new Person("Smith", "London", 1984);*

## Resumo

- Processos trabalham com estruturas internas de dados (variáveis, cada um de um determinado tipo de dados)
- Comunicação por mensagens implica envio de dados na forma de sequências de bytes

- Processos diferentes podem ter formas diferentes de representar estruturas de dados equivalentes
- Um tipo de dados que é enviado numa mensagem tem de poder ser corretamente reconstruído no destino
- Um formato externo de representação de dados evita que essas diferenças na representação de dados possam gerar interpretações incorretas das sequências de bytes trocadas entre processos

## Exercícios

1. Why is there no explicit data-typing in CORBA CDR, in opposite to Java serialization, for example? Which are the advantages?

Os tipos de dados estão implícitos na definição do protocolo (isto é particularmente verdade nos sistemas de invocação remota em que é usado o CORBA XDR)

Menos espaço nas mensagens

Menos tempo de transmissão e processamento das mensagens.

## 4.1 - Middleware - Invocação remota entre processos

### Invocação remota

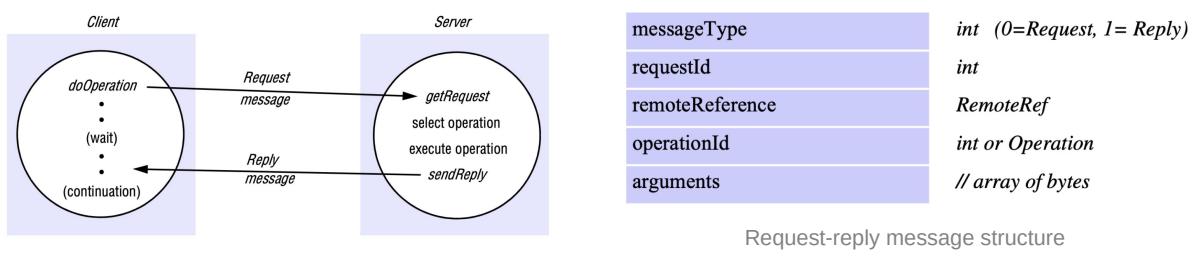
Invocação remota pressupõe troca de mensagens entre processos

Modelo cliente-servidor pressupõe uma troca de mensagens na qual cliente solicita operação ao servidor

Protocolo de comunicação especifica que mensagens devem ser enviadas e que comportamento devem ter as entidades perante as várias situações que podem ocorrer

Dependendo de necessidades concretas deve-se escolher um protocolo com as propriedades adequadas

### Protocolos pedido-resposta



Request-reply communication

#### Principais questões a considerar:

- Identificação das mensagens
- Falhas
  - *Timeouts*

- Mensagens perdidas, trocadas, duplicadas
- Falhas nos processos
- Histórico de mensagens

## Protocolos de invocação remota

Independentemente das muitas especificidades dos muitos protocolos que existem, há 3 modelos fundamentais de protocolo de invocação remota

Name	Messages sent by		
	Client	Server	Client
R	<i>Request</i>		
RR	<i>Request</i>	<i>Reply</i>	
RRA	<i>Request</i>	<i>Reply</i>	<i>Acknowledge reply</i>

RPC exchange protocols

### Protocolo R (*Request*):

- Não se espera resposta do servidor
- Não há necessidade de confirmação de execução pelo servidor
- Cliente envia mensagem e continua
- Não dá garantias da execução da operação se funcionar sobre UDP
- Exemplo: serviços de atualização periódica (*heartbeat, sensor update, ..*)

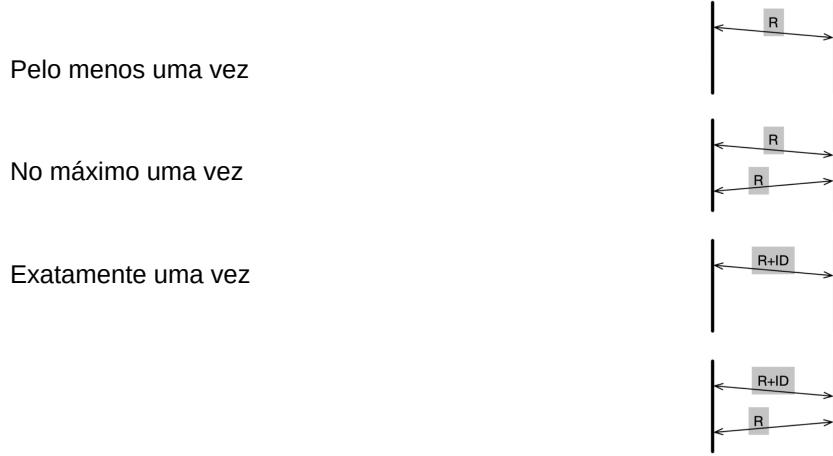
### Protocolo RR (*Request/Reply*)

- É suposto servidor responder ao pedido
- Resposta do servidor serve como confirmação da receção do pedido (*implicit acknowledgement*)
- *Timeout* determina falha no pedido e originam repetição do pedido para o servidor
- Garantia de execução da operação, uma ou mais vezes, se recurso a filtro de pedidos duplicados e histórico de mensagens de resposta

### Protocolo RRA (*request/reply/acknowledged-reply* )

- Cliente confirma explicitamente a correta receção da resposta do servidor
- Como mensagem ACK se pode perder, servidor tem de aplicar ordenação nos pedidos para poder identificar ACKs fora de ordem ou usar ACK implícitos
- Não interfere com as garantias de execução de operações mas permite ao servidor confirmar que pedido foi recebido e portanto pode ser fechado, libertando recursos que lhe estejam associados.
- Por exemplo, um pedido pode originar cálculos complexos e demorados que seria pouco eficiente estar a repetir se por acaso a mensagem de resposta se perdesse.

Não é permitido nada



## TCP vs UDP

Tamanho das mensagens

- mensagens com muitos dados podem implicar o envio de mais do que um pacote/datagrama UDP

Transmissão orientada à ligação

- O cliente recebe uma resposta ou Informação de quebra de ligação
  - semântica “no máximo uma vez”

## Middleware para sistemas distribuídos

**Motivação:**

- Os sistemas operativos (ou máquina virtuais) oferecem aos processos um serviço de comunicações que é disponibilizado através de sockets (TCP ou UDP)
- Serviço é muito flexível mas de baixo nível
- Obriga os processos a terem de lidar com diversas tarefas genéricas e repetitivas associadas à comunicação
- Tarefas genéricas a considerar na utilização da API sockets
  - Definir um protocolo de comunicação que determina a sequência e formato das mensagens que vão ser trocadas
  - Definir previamente um porto e um endereço do servidor de modo a poder estabelecer uma ligação com o mesmo
  - Transformar os dados a enviar numa sequências de bytes e vice-versa
  - Definir um formato de representação dos dados comum a todos os processos
  - Lidar com erros resultantes da comunicação

Globalmente, as funcionalidades genéricas envolvidas na invocação de um pedido a um servidor remoto representam um processo complexo demais para ser desenvolvido para cada nova aplicação

Estar a desenvolver repetidamente a mesma funcionalidade genérica em cada novo sistema é um desperdício de esforço que leva ao aumento dos custos de desenvolvimento

Ter qualquer programador a desenvolver o suporte correspondente a estas funcionalidades genéricas significa um risco muito elevado de criar implementações pouco adequadas por não lidarem da forma mais correta com todos os desafios inerentes à comunicação remota

O desenvolvimento de baixo nível tende a aumentar a complexidade e as dependências do sistema, criando maiores desafios de manutenção de código e maior propensão a erros de programação

## **Middleware de invocação remota**

### **Solução desejável:**

Baseada em paradigmas de invocação de mais alto nível que tornam mais fácil o desenvolvimento de aplicações distribuídos e fazem com que seja mais viável o desenvolvimento de aplicações distribuídas complexas

Integrada nos modelos de programação existentes e com os quais os programadores já estão familiarizados

Solução genérica para muitos tipos de aplicações distribuídas

Capaz de esconder os detalhes da heterogeneidade das várias entidades envolvidas na comunicação

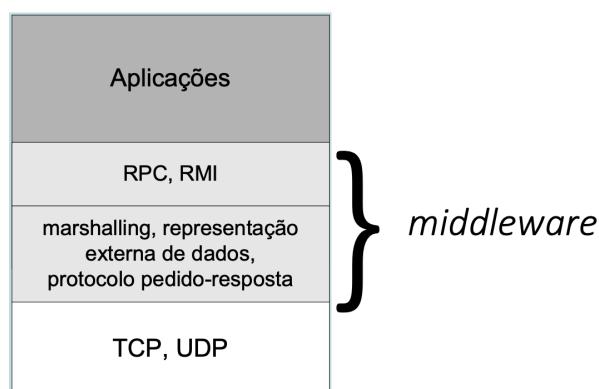
Sistematiza tarefas que de outra forma precisariam de ser repetidas para cada aplicação desenvolvida

- Marshaling/unmarshaling
- Definição de uma representação externa
- Definição de um protocolo pedido-resposta

Proporciona paradigmas de programação distribuída de mais alto nível do que o envio de mensagens

Permite incorporar o processamento distribuído como uma extensão natural dos mecanismos das linguagens de programação

- Invocação de procedimentos remotos
- Invocação de métodos em objetos remotos



### **Vantagens:**

- Facilita o desenvolvimento de aplicações distribuídas
- Desenvolvimento mais rápido
- Menos propício a erros
- Complexidade das aplicações reduzida

### Desvantagens:

- Menos flexível
- Menos espaço para optimizações de desempenho
- Soluções genéricas tendem a ter maiores custos de processamento e dados
- Existência de mais componentes ativas aumenta consumo de recursos na máquina

### Invocação de Procedimentos Remotos ou *Remote Procedure Call* (RPC)

Exemplos: Sun RPC, Web Services

### Invocação de Métodos Remotos ou *Remote Method Invocation* (RMI)

Exemplos: Java RMI, Corba

## Chamada de Procedimentos Remotos (RPC)

Uma extensão aos métodos convencionais de invocação de procedimentos

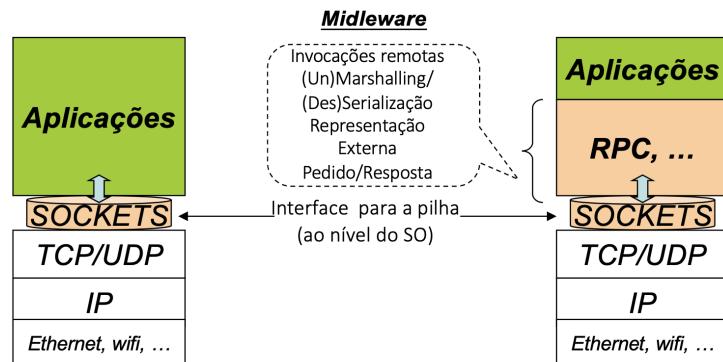
Permite que uma aplicação cliente invoque procedimentos num espaço de endereçamento distinto, na mesma máquina ou em máquinas remotas;

Ideal para aplicações baseadas no modelo cliente-servidor

O servidor oferece uma interface de serviço (*service interface*)

Objetivo principal: facilitar a programação distribuída

- RPC como uma abstração construída sobre a pilha protocolar, e que usa a interface disponibilizada pelo sistema operativo e esconde os detalhes e a complexidade do programador



Como tornar a invocação remota parte da linguagem de programação de uma forma transparente?

Como trocar mensagens entre máquinas distintas que podem representar os dados de forma diferente?

Como localizar o serviço pretendido numa enorme coleção de servidores e/ou de serviços?

- Transparência da localização, assumindo que o cliente não tem de saber onde pode o serviço residir

Como lidar com os erros de uma forma mais ou menos elegante?

- Servidor desligado, Rede em baixo, Servidor ocupado, Pedidos duplicados, etc...

### Interfaces no RPC:

Cada servidor disponibiliza um conjunto de procedimentos ou interface de serviço (*service interface*) para serem invocados pelos seus clientes

Uma *Interface Definition Languages* (IDL) disponibiliza a notação para definir interfaces em que os parâmetros de uma operação são descritos como de entrada ou saída assim como os seus tipos

Exemplos: SUN XDR, CORBA IDL, WSDL Web services

#### **Exemplo - SUN XDR IDL:**

```
/ * servidorFicheiros.x especificação da interface */ <definição dos códigos de erro>
#define FILE_SERVER_NAME "miegsi.dsi.uminho.pt"
struct Data {
    int tam;
    char buffer[1024];
};

struct writeargs {
    char name[255];
    int position;
    Data data;
};

struct readargs {
    char name[255];
    int position;
    int length;
};

program FILESERVER{
    version VERSION{.....
        int WRITE(writeargs) = 1;
        Data READ(readargs) = 2;
    } = 2;
} = 9999;
```

Transparência:

- de acesso
- de localização

#### **Passagem de parâmetros nas invocações convencionais:**

Os parâmetros podem ser passados usando Passagem-por valor e Passagem-por-referência

#### **Passagem de parâmetros nas invocações RPC:**

Só é possível passagem-por valor

Apontadores (endereços) não têm qualquer significado num espaço de endereçamento distinto

Também se designa de passagem por cópia (copy-in/copy-out)

#### **Transparência de acesso:**

- Uma invocação de procedimento remoto é mais vulnerável a falhas do que uma invocação de procedimentos locais
- É sempre possível que o resultado não seja recebido e, em caso de falha, não é possível distinguir, como vimos, falhas de rede e falhas de processos.
- É necessário que os clientes que fazem a invocação remota estejam preparados para recuperar destas situações.

#### **Transparência de localização**

Um problema que se coloca é como determinar a localização e a identidade do procedimento remoto a invocar, normalmente corresponde a um *socket* (*host+porta*)

Esta operação de associação (*Binding*) pode ser feita de forma estática ou dinâmica:

- *binding* estático associa o endereço e porta do servidor ao cliente em tempo de compilação (indesejável):

Cliente e servidor são compilados separadamente e em tempos diferentes

Servidor pode mudar de máquina

- *binding* dinâmico é mais desejável:

Permite que os servidores registem os serviços que exportam

Permite que os clientes procurem o serviço pelo nome

Solução: Serviço de registo RPC

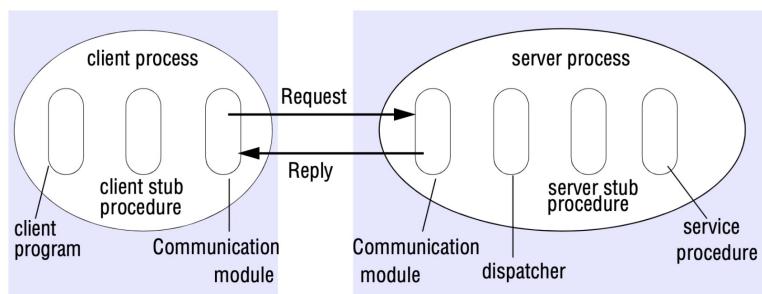
### **Serviço de registo RPC:**

O registo é um serviço que pode ele próprio ser implementado usando RPC:

- PROCEDURE Register (serviceName:String; serverPort:Port; version:integer)
  - regista nome do serviço e a versão na sua tabela
- PROCEDURE Withdraw (serviceName:String; serverPort:Port; version:integer)
  - remove serviço da tabela
- PROCEDURE LookUp (serviceName:String; version:integer): Port
  - procura serviço na tabela e devolve endereço (ou endereços) associados

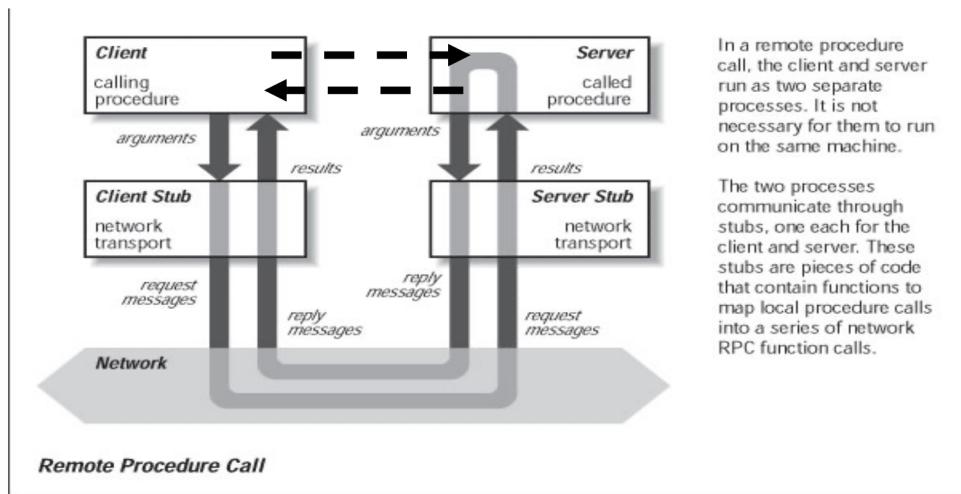
### **Implementação:**

Componentes de software necessárias para implementar um sistema de RPC



Role of client and server stub procedures in RPC

### **Remote Procedure Call (RPC):**



### RPC em pseudo-código:

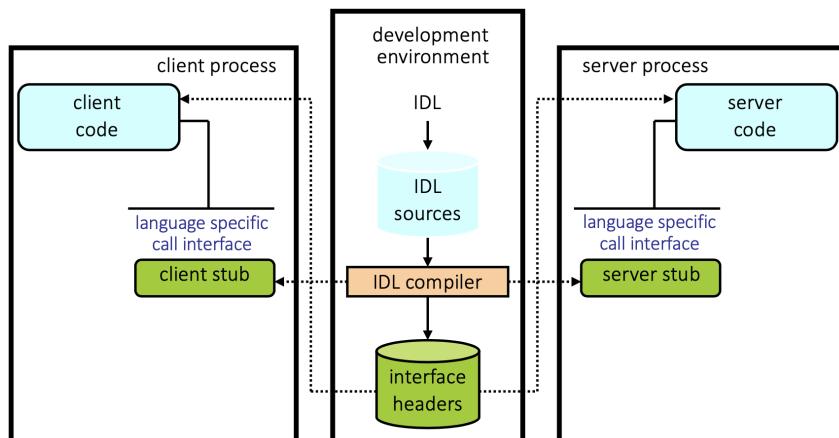
```
//your client code
result = function(parameters)

//client side stub
function(parameters) {
    address a = bind("function");
    socket s = connect(a);
    send(s,"function");
    send(s,parameters);
    receive(s,result); //blocking
    return result;
}

//rpc server main loop
void rpc_server() {
    register("function",address);
    while (true) {
        socket s = accept(); //blocking
        receive(s,id);
        if (id == "function")
            dispatch_function(s);
        close(s);
    }
}

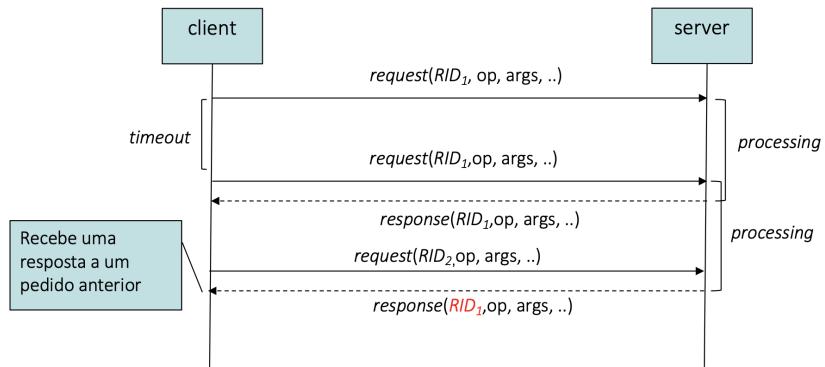
//server side stub
void dispatch_function(socket s) {
    receive(s,parameters);
    result = function(parameters);
    send(s,result);
}
```

### Ambiente de desenvolvimento:



## Exercícios

1. Describe a scenario in which a client could receive a reply from an earlier call.



1. **Discuss whether the following operations are *idempotent*:**

- i) pressing a lift (elevator) request button;
- ii) writing data to a file;
- iii) appending data to a file

Is it a necessary condition for idempotence that the operation should not be associated with any state?

A operação write(ficheiroId, 1024, buffer) não é idempotente, porque se baseia no valor de uma posição de escrita que faz parte do estado do servidor

A operação write(ficheiroId, POSICAO, 1024, buffer) é idempotente porque tem toda a informação para ser executada e não se baseia no estado do servidor

A operação de append de informação a um ficheiro não é idempotente porque depende do estado anterior

## Invocação de métodos remotos (RMI)

### RMI (*Remote Method Invocation*):

Estende o modelo de programação baseado em objetos de modo a suportar a invocação de métodos em objetos de outros processos.

- Oferece um paradigma de programação baseado em interfaces
- Esconde a complexidade da comunicação remota entre processos ao programador
- É construído em cima de um paradigma pedido-resposta
- Oferece transparência na invocação de métodos locais/remotos
- Oferece todos os conceitos de um paradigma orientado a objetos no desenvolvimento de sistemas distribuídos
- Todos os objetos (locais ou remotos) possuem uma referência e podem ser passados como parâmetros de métodos

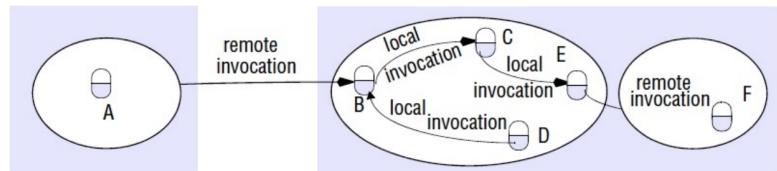
Um sistema distribuído é formado por objetos distribuídos por diferentes processos/máquinas

Normalmente estes sistemas distribuídos adotam um modelo arquitetural cliente- servidor

Os processos servidores

gerem objetos remotos

Os processos clientes invocam os métodos dos objetos fazendo uma invocação remota



Remote and local method invocations

### RMI - Referência Remota:

Invocar um objeto implica ter uma referência para esse objeto

Quando o objeto é instanciado no mesmo espaço de endereçamento, essa referência existe na forma de uma variável gerada quando se instancia o objeto (normalmente num método construtor)

Quando o objeto é remoto, não existe uma referência local e portanto precisamos de outro tipo de referência

Quando o objeto é remoto, não é possível usar um construtor de classe para criar o objeto, temos de obter uma referência para um objeto já existente

**Referências remotas** são referências para objetos remotos e permitem invocação de métodos remotos

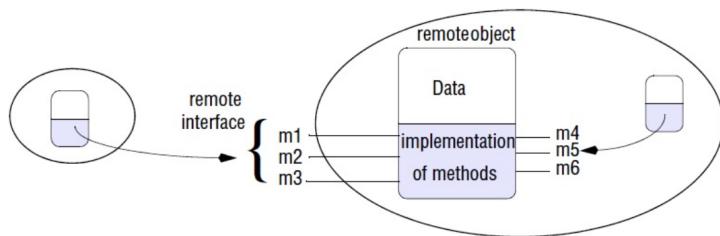
Não são referências para posições de memória como objetos locais

São referências para um serviço disponível na rede (identificam, e.g., a localização do serviço)

### RMI - Interface Remota:

Uma interface remota declara a assinatura dos métodos remotos

A classe de um objeto remoto implementa os métodos da sua interface remota



A remote object and its remote interface

Objetos em processos diferentes podem invocar apenas os métodos declarados na interface remota do objeto

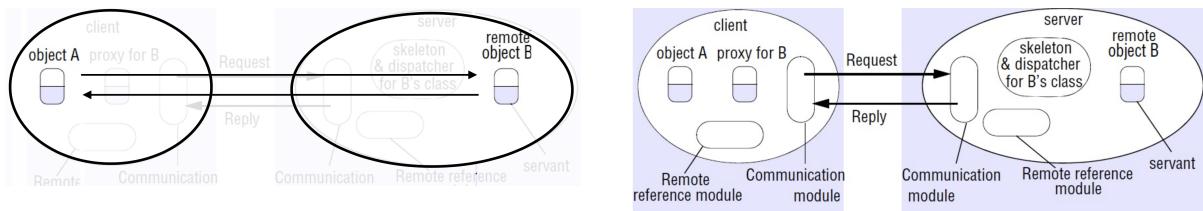
Métodos que não são remotos podem ser invocados por outros objetos no mesmo processo

Argumentos de métodos remotos não podem incluir referências locais a variáveis, apenas valores

Argumentos de métodos remotos podem incluir referências remotas

### Implementação de RMI:

Middleware RMI suporta um processo de invocação remota que cria a ilusão de que temos uma referência para um objeto remoto e podemos invocar diretamente os seus métodos



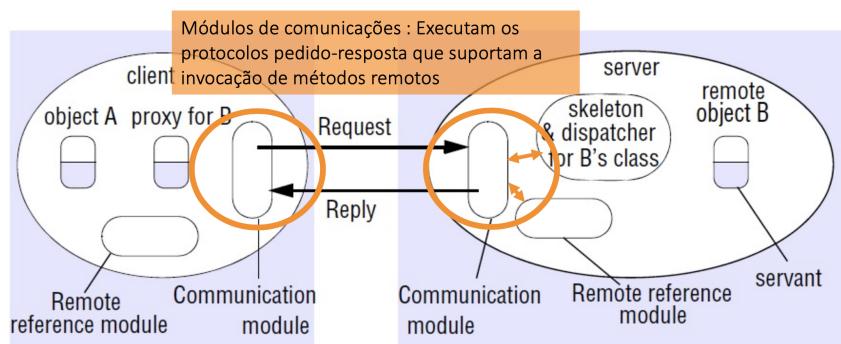
...mas invocação de métodos em objetos de outras JVMs não é realmente possível!

### Módulo de Comunicações:

Os módulos de comunicações em ambos os intervenientes cooperam na execução do protocolo pedido-resposta que suporta a invocação do método remoto

Definem o tipo de protocolo que determina a semântica de invocação, e.g. *no máximo uma vez*

No lado do servidor, o modulo de comunicação passa a informação ao *dispatcher* do objeto remoto que se pretende invocar, após ter obtido a sua referência a partir do módulo de referências remotas



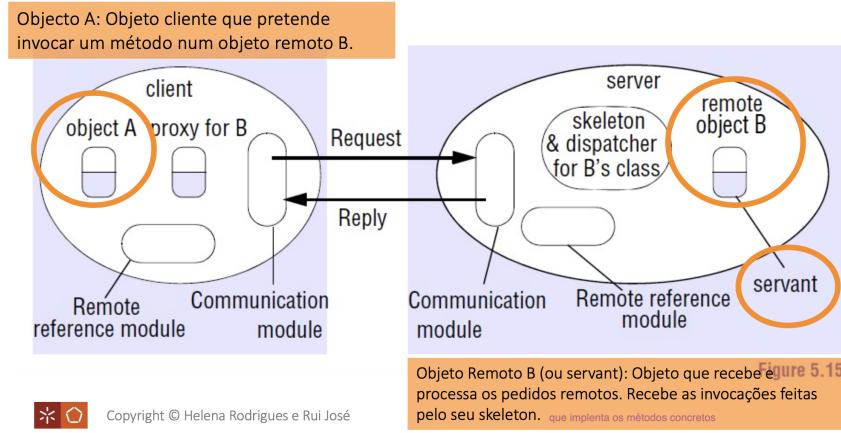
### Módulo de referências remotas:

O módulo de referências remotas é responsável por gerir pela tradução entre referências locais (para objetos proxy ou *skeleton*) e referências remotas e por criar referências remotas

Gere uma tabela de *objetos remotos* que mapeia entre referências de objetos locais (desse processo) e referências de objetos remotos.

Permite ao módulo de comunicações saber para onde encaminhar uma invocação remota ou entregar uma resposta

### Implementação de RMI:



### Proxy:

Representa no lado cliente o objeto remoto

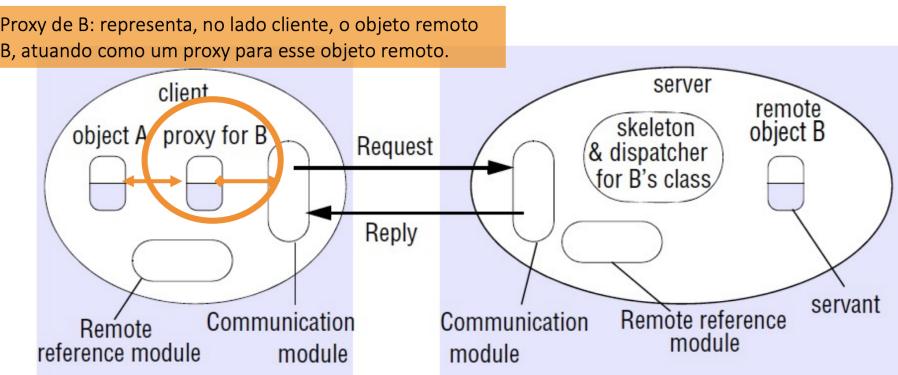
O seu papel é tornar a invocação remota transparente ao cliente

As invocações remotas são na verdade invocações locais realizadas pelo cliente no proxy que implementa a mesma interface que o objeto remoto.

Em vez de executar a invocação, encaminha o pedido numa mensagem para o objeto remoto (através do módulo de comunicações).

Esconde do cliente toda a complexidade associada às tarefas de serialização dos parâmetros, interação pedido/resposta e desserialização dos resultados

Gerado automaticamente pelo *Middleware* com base numa especificação da interface remota do objeto remoto



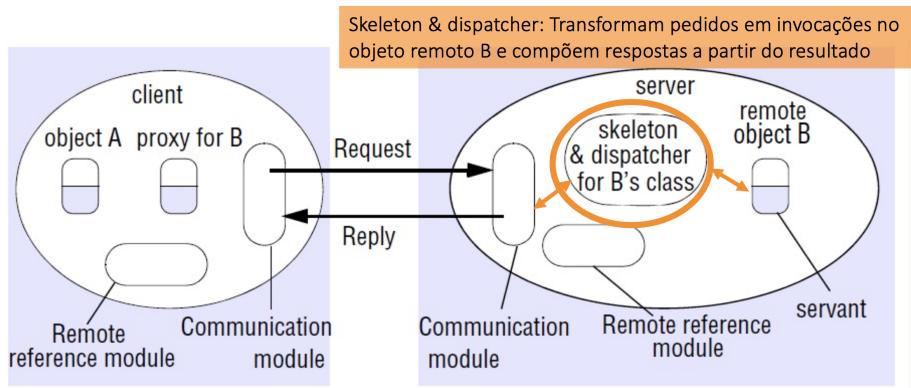
### Skeleton & Dispatcher

Existe um objeto *Dispatcher* e *Skeleton* para cada classe de um objeto remoto

O *dispatcher* recebe a mensagem pedido a partir do módulo de comunicações. Usa o *operationID* (que vêm na mensagem) para selecionar o método apropriado no *Skeleton*.

O *Skeleton* implementa os métodos da interface remota do objeto correspondente. Quando chega um pedido de invocação de um cliente é necessário:

- Interpretar os dados e construir as respetivas estruturas de dados
- Invocar o método na verdadeira implementação do objeto “remoto”.
- Serializar o resultado e devolvê-lo numa mensagem para o cliente



### Implementação de RMI:

#### Geração automática de código:

As classes para o *Proxy*, *Dispatcher* e *Skeleton* são geradas automaticamente por um Compilador de Interface

Um programa servidor contém as classes para *dispatchers* e *skeletons* e as classes para todos os objetos remotos (*servants*) que disponibiliza no sistema distribuído

Contém ainda uma secção de inicialização responsável por criar e inicializar pelo menos um objeto remoto hospedado no servidor

Um programa cliente terá de conter todas as classes de *proxies* de objetos remotos que vai invocar

#### Serviço de Nomes

Os programas cliente genericamente necessitam de um meio para obter uma referência remota pelo menos para um objeto hospedado num servidor

Um Serviço de nomes é um componente de um sistema distribuído que associa serviços (neste caso referências para objetos remotos) a nomes

Os programas cliente obtêm as referências para objetos remotos a partir dos seus nomes

Objetos que invocam métodos de outros objetos não precisam de distinguir objetos locais de remotos (Transparência de acesso)

- Referências locais e remotas são usadas (quase) da mesma forma

### RMI – Ativação de objetos:

Os objetos remotos podem estar no estado “passivo” ou estado “ativo”

- Estado ativo – o objeto está disponível para invocação
- Estado passivo – o objeto não está correntemente disponível para invocação, mas pode ser ativado

Objetos no estado passivo consistem em duas partes:

- A implementação dos seus métodos
- O seu estado na forma serializado

O processo de ativação consiste em criar uma nova instância do objeto a partir do seu estado serializado

Algumas implementações oferecem processos ativadores

### **Modelo de *threading* em RMI:**

A especificação do RMI não faz garantias sobre o modelo de *threading* ao servir invocações remotas (depende de uma implementação em particular):

- Pode ser ativada uma única *thread* para servir invocações sucessivas;
- Pode ser usada uma *thread* por invocação (de forma transparente ao programador )

Na prática verifica-se que invocações remotas a um dado objeto que chegam concorrentemente são despachadas para *threads* diferentes

Implementação de objetos remotos tem necessidade de controlo de concorrência e devem ser *thread safe* (protecção contar *race conditions*)

### **JAVA RMI:**

Mecanismo de *Remote Method Invocation* (RMI) para a linguagem Java

Permite a um objecto numa *Java Virtual Machine* (JVM) invocar os métodos de um objecto numa outra JVM de forma quase transparente para o programador

Um programa servidor cria objetos remotos, disponibiliza a referência para esses objetos e aguarda invocações nos respetivos métodos

Um programa cliente obtém referências para os objetos remotos que necessita e invoca os respetivos métodos remotos

### Interfaces remotas:

Em java, uma interface remota é definida com recurso à interface *Remote* - uma interface remota em java herda a declaração de métodos da interface *Remote*.

Todos os métodos devem lançar a exceção *RemoteException*

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface NewPresencesInterface extends Remote {
    public void setNewPresence(String IPAddress) throws RemoteException;
}

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Vector;

public interface PresencesInterface extends Remote {
    public Vector<String> getPresences(String IPAddress, NewPresencesInterface cl) throws RemoteException;
}
```

### Passagem de parâmetros:

Em JAVA RMI um método tem parâmetros de entrada e devolve um único resultado

Qualquer objeto *serializável* pode ser um parâmetro de entrada ou saída.

Sempre que o tipo de um parâmetro de entrada ou saída é um objeto remoto, o argumento ou resultado correspondente é sempre transmitido como uma referência remota.

Quando um processo recebe uma referência remota, pode usar essa referência para invocar os métodos do objeto a que esta se refere.

Objetos locais são passados por valor

Quando um objeto é passado por valor, o processo destino cria um novo objeto; Os métodos do novo objeto podem ser invocados localmente

#### Java RMIRegistry:

Os objetos remotos são “localizados” pelos clientes através da consulta de um serviço de nomes:  
**rmiregistry**

O RMI Registry é um serviço de nomes, sem hierarquia

Permite associar nomes a objetos

Fornece as operações tradicionais de *bind*, *rebind*, *unbind*, *lookup* e *list*

Nomes dos objetos utilizam um URL com o seguinte formato:

rmi://host:port/name

Cada objecto remoto regista-se normalmente no RMI Registry da máquina onde se encontra

##### *The Registry class of Java RMIRegistry*

*void rebind (String name, Remote obj)*

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 5.18, line 3.

*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, as shown in Figure 5.20, line 1. A remote object reference is returned.

*String [] list()*

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

#### **RMI - Implementação do sistema - Exemplo:**

Passo 1 – Definição de  
Interfaces Remotas

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Vector;

public interface PresencesInterface extends Remote {
    public Vector<String> getPresences(String IPAddress) throws RemoteException;
}
```

Passo 2: Implementação da classe  
que implementa a interface remota

```

import .....

public class Presences extends UnicastRemoteObject implements PresencesInterface {
    private Hashtable<String, IPInfo> presentiPs = new Hashtable<String, IPInfo>();
    public Presences() throws RemoteException {
        super();
    }
    public Vector<String> getPresences(String IPAddress) throws RemoteException {
        long actualTime = new Date().getTime();
        Vector<String> result = new Vector<String>();
        //Insere o IP e obtém a lista de IPs presentes
        .....
        return result;
    }
    .....
}

```

Passo 3: Implementação do programa servidor (*main* do servidor) que cria e hospeda o objeto remoto

```

import .....
public class PresencesServer {
    public static void main(String[] args) {
        .....
        Presences presences = null;
        presences = new Presences();
        if( System.getSecurityManager() == null ) {
            System.setSecurityManager(new RMISecurityManager());
        }
        LocateRegistry.createRegistry(1099);
        LocateRegistry.getRegistry("127.0.0.1",1099).rebind("/PresencesRemote", presences);
    }
}

```

```

public class Client {
    public static void main(String[] args) {
        Client client;
        .....
        PresencesInterface presences = (PresencesInterface) LocateRegistry.getRegistry("127.0.0.1").lookup(SERVICE_NAME);
        Vector<String> presencesList = presences.getPresences(args[0]);
        .....
    }
}

```

Passo 5: Execução do sistema

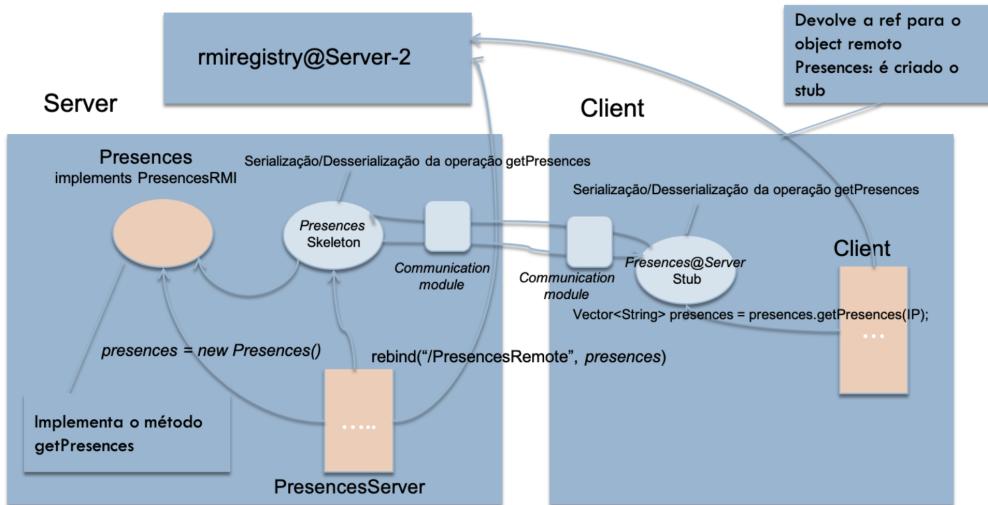
para executar o serviço invocar *java PresencesServer*

para invocar a aplicação cliente invocar *java Client <ip Cliente> <ip Servidor de nomes Java RMI>*

Nas versões mais recentes do java RMI (pos-v5.0), não é necessário gerar explicitamente os *proxies* e *skeletons*

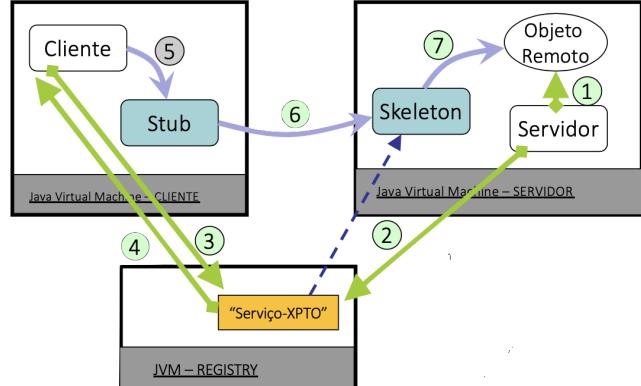
*Proxies e Skeletons* são gerados no momento da compilação dos programas

Passo 4:  
Implementação  
do programa  
cliente



### Remote Method Invocation (Visão Geral):

1. Servidor cria o Objeto Remoto
2. Servidor Regista o Objeto Remoto no *rmiregistry*
3. Cliente pede ao *rmiregistry* referência para Objeto Remoto
4. *rmiregistry* devolve referência remota e o *stub* é criado
5. Cliente invoca método no *stub*
6. *Stub* contacta o *Skeleton* (via módulo de comunicação)
7. *Skeleton* invoca método no objeto remoto



## Exercícios

1. An Election interface provides two remote methods:

**vote:** This method has two parameters through which the client supplies the name of a candidate (a string) and the 'voter's number' (an integer used to ensure each user votes once only). The voter's numbers are allocated sparsely from the range of integers to make them hard to guess.

**result:** This method has two parameters through which the server supplies the client with the name of a candidate and the number of votes for that candidate.

Define the interface to the Election service in Java RMI.

```

class Result {
    String name;
    int votes;
}

public interface Election extends Remote{
    void vote(String name, int number) throws RemoteException;
    Result result () throws RemoteException; }

```

2. The Election service must ensure that a vote is recorded whenever any user thinks they have cast a vote. Discuss the effect of maybe call semantics on the Election service. Would at least-once call semantics be acceptable for the Election service or would you recommend at most-once call semantics?

Quando uma invocação remota oferece a semântica “maybe” ou talvez, quer dizer que a operação remota talvez tenha sido executada. Para este cenário, é claro que esta semântica não é adequada porque um eleitor quer ter a certeza que o seu voto é contabilizado. Quando uma invocação remota oferece a semântica “at-least-once” (pelo menos uma vez), quer dizer que é garantido que a operação remota é executada uma vez, podendo no entanto ser executada mais vezes.

O servidor da eleição mantém informação sobre quem já votou; A semântica é adequada; se a mensagem de pedido for repetida, a lógica do servidor está preparada para ignorar as repetições.

3. Assume the Election service is implemented in RMI and that the service uses a Vector to store the votes (or other more efficient structure) and must ensure that all votes are safely stored even when the server process *crashes*. Explain how this can be achieved.

Se um processo termina abruptamente, são perdidos todos os dados em memória RAM, por isso, para que nenhum voto seja perdido quando o processo reinicia, é necessário gravar os votos em memória persistente (ficheiros ou base de dados).

Para dar garantia ao cliente que o método foi contabilizado, o servidor deverá guardar o voto em memória persistente antes de responder ao cliente. Assim, quando o cliente recebe a resposta, sabe que a operação de voto foi executada pelo menos uma vez e que o voto foi armazenado em memória persistente.

## Web services

Existem dois estilos de paradigmas de Web services:

- RPC Web services
- REST Web services

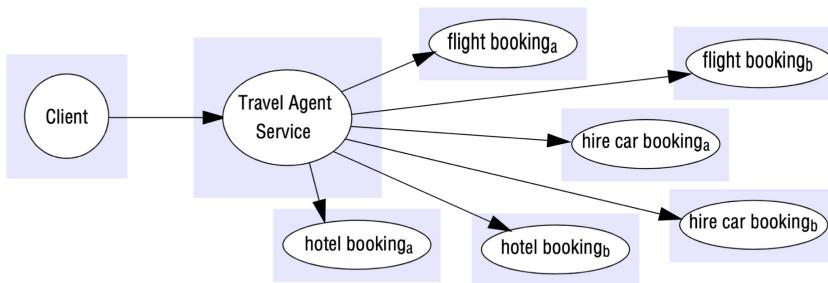
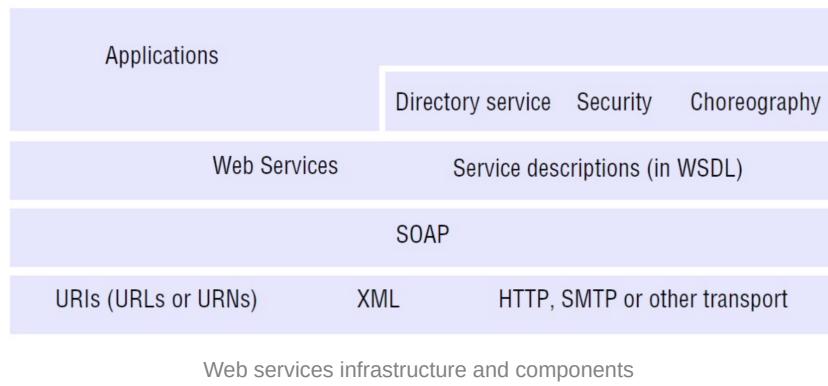
### RPC Web Services

A tecnologia de RPC Web services implementa o paradigma de invocação de procedimentos remotos no contexto das tecnologias Web.

Um web service é um conjunto de operações que podem ser usadas por aplicações cliente (acesso programático) usando protocolos da Internet (tipicamente HTTP)

Modelo orientado e especialmente adequado para interoperabilidade à escala da Internet

São extensões à Web e são disponibilizados por servidores Web.



The "travel agent service" combines other web service

### **Padrões de comunicação:**

Paradigma de comunicação direta (protocolo pedido-resposta)

Paradigma de comunicação indireta

Mensagens assíncronas

Eventos

### **Representação externa de dados:**

As mensagens são representados em XML (dados e protocolo)

Representação textual, humanamente legível

Facilidade de *debug*

"Referências" para Web services são normalmente URLs, também conhecidos por *endpoints*

Um Web service é acedido através da máquina especificada no URL, mas pode executar em máquinas diferentes;

O URL é uma referência persistente

Um Web service pode ser executado continuamente ou ativado a pedido

### **Transparência de acesso:**

Detalhes da comunicação (criação das mensagens de pedido-resposta, *marshaling* e *unmarshaling* e representação externa de dados) são normalmente escondidos por bibliotecas locais de uma linguagem de programação como o Java, Python, C#, C++, etc

A transparente da invocação remota é suportada pela geração automática de *proxies*.

## **SOAP:**

### ***Simple Object Access Protocol (SOAP):***

Especifica as regras para definição de mensagens de pedido e de resposta com base em XML.

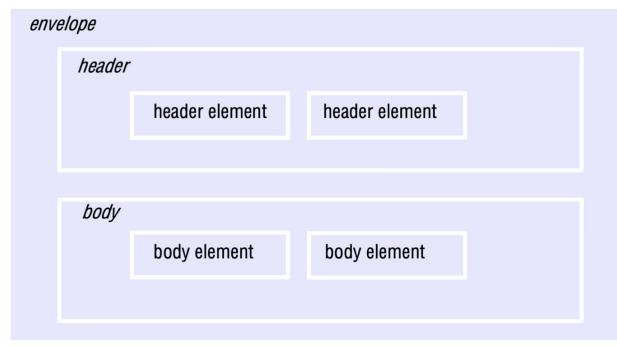
Cliente invoca um procedimento remoto enviando uma mensagem SOAP (XML) para o servidor. Servidor responde com uma outra mensagem SOAP.

Elementos fundamentais

- Um modelo de envelope que descreve o conteúdo da mensagem e a forma como ela deve ser interpretada
- Regras para representação dos dados enviados nas mensagens
- Regras para representar a invocação de procedimentos remotos e respetiva regras

Independente da linguagem e plataforma

### **Elementos SOAP:**



SOAP message in an envelope

### **SOAP request:**

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope  
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">  
        <SOAP-ENV:Header/>  
        <S:Body>  
            <ns2:GetPresence xmlns:ns2="http://soap.ws.sd/">  
                <IPAddress>193.137.8.9</IPAddress>  
            </ns2:GetPresence>  
        </S:Body>  
</S:Envelope>
```

### **SOAP reply**

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP- ENV="http://schemas.xmlsoap.org/soap/envelope/">  
    <SOAP-ENV:Header/>  
    <S:Body>  
        <ns2:GetPresenceResponse xmlns:ns2="http://soap.ws.sd/">  
            <return xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="xs:string">193.137.8.9</return>  
        </ns2:GetPresenceResponse>  
    </S:Body>  
</S:Envelope>
```

---

## **WSDL:**

### **WSDL (Web Services Definition Language):**

A descrição da interface de um serviço é um elemento essencial para permitir aos clientes comunicar com esse serviço

Em web services essa especificação faz parte de uma descrição mais geral que abrange os seguintes elementos:

- Quais as operações suportadas
- Onde é que está disponível (URL)
- Como é que pode ser invocado (e.g. SOAP sobre HTTP)

Descrição de um serviço é feita em XML e permite a quem precisa de usar o web service obter todas as informações necessárias para o efeito

As descrições de interface geralmente são usadas para gerar *proxies* de cliente que implementam automaticamente o comportamento correto para o cliente

### **Exemplo WSDL (*Schemes e types*):**

#### **Schemes e types:**

```
<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns: wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns: wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns: soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns: tns="http://soap.ws.sd/"
  xmlns: xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://soap.ws.sd/" name="PresenceWS_SOAP">
<types>
<xsd:schema>
<xsd:import namespace="http://soap.ws.sd/"
  schemaLocation="http://localhost:8080/PresenceWSApplication/PresenceWS_SOAP?xsd=1"/>
</xsd:schema>
</types>
... (cont.)
```

#### **Messages e interface:**

```

... (cont.)
<message name="GetPresence">
<part name="parameters" element="tns:GetPresence"/>
</message>
<message name="GetPresenceResponse">
<part name="parameters" element="tns:GetPresenceResponse"/>
</message>
<portType name="PresencesWS_SOAP">
<operation name="GetPresence">
<input wsam:Action="http://soap.ws.sd/PresencesWS_SOAP/GetPresenceRequest" message="tns:GetPresence"/>
<output wsam:Action="http://soap.ws.sd/PresencesWS_SOAP/GetPresenceResponse"
message="tns:GetPresenceResponse"/>
</operation>
</portType>
... (cont.)

```

**2 tipos de mensagens:  
GetPresence e  
GetPresenceResponse**

Definição de interface (*portType*):  
operação *GetPresence* que usa  
as mensagens *GetPresence* e  
*GetPresenceResponse*

### Binding:

```

... (cont.)
<binding name="PresencesWS_SOAPPortBinding" type="tns:PresencesWS_SOAP">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
<operation name="GetPresence">
<soap:operation soapAction="" />
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="PresenceWS_SOAP">
<port name="PresencesWS_SOAPPort" binding="tns:PresencesWS_SOAPPortBinding">
<soap:address location="http://localhost:8080/PresenceWSApplication/PresenceWS_SOAP"/>
</port>
</service>
</definitions>

```

Um *binding* para HTTP no  
URL indicado



Copyright © Helena Rodrigues e Rui Insé

### Serviço de Nomes e Diretoria:

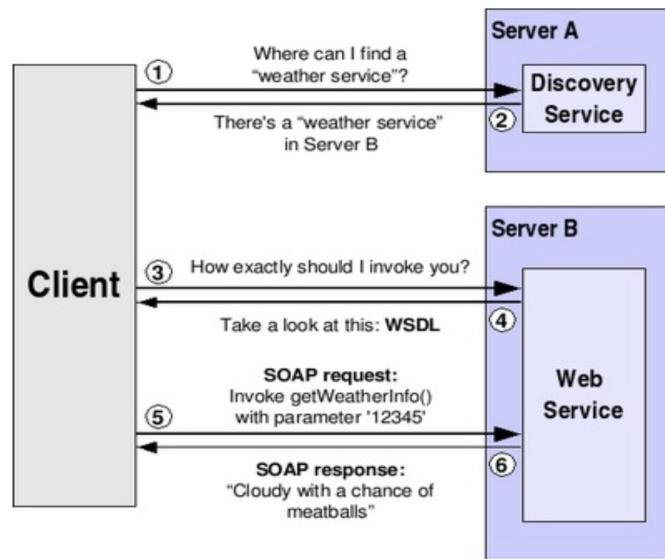
*Universal Description, Discovery, and Integration (UDDI)*

Define as normas para a publicação e descoberta de *web services*

Permite que os *web services* sejam descobertos dinamicamente

---

### Exemplo de Invocação de Web Services



### Web services middleware:

Um dos objetivos mais comuns em *middleware* de sistemas distribuídos é a transparência tentando fazer com que as invocações remotas sejam tanto quanto possível parecidas com as locais

Os Web Services não oferecem diretamente nenhuma implementação de middleware para suportar a invocação remota de Web services. Em princípio seria necessário a uma aplicação gerar as suas próprias mensagens SOAP/XML e enviá-las por HTTP.

Os Web Services não têm nenhum mecanismo para esconder os detalhes de representação externa de dados. Em princípio seria necessário a uma aplicação gerar os formatos XML necessários

Contudo existem muitas bibliotecas para quase todas as linguagens que permitem integrar o suporte a web services diretamente no ambiente de programação

Com base na descrição WSDL é gerado automaticamente o código necessário para interagir com esse serviço (mensagens, marshalling, ...)

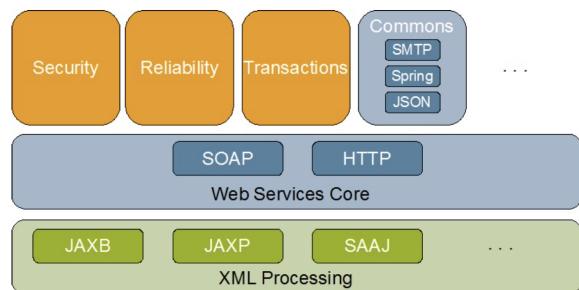
### **Metro – Java web services stack:**

*Middleware* de desenvolvimento e *deployment* de serviços Web baseado em ferramentas e APIs Java (JAX-RPC)

JAXB – Mapeia componentes de um *schema XML* em objetos e classes Java

JAXP – API Java para processamento de XML;

SAAJ – disponibiliza APIs Java para a criação e envio de mensagens SOAP



Metro – GlassFish Web Services Stack  
metro.dev.java.net

### **Objetos distribuídos vs RPC Web services:**

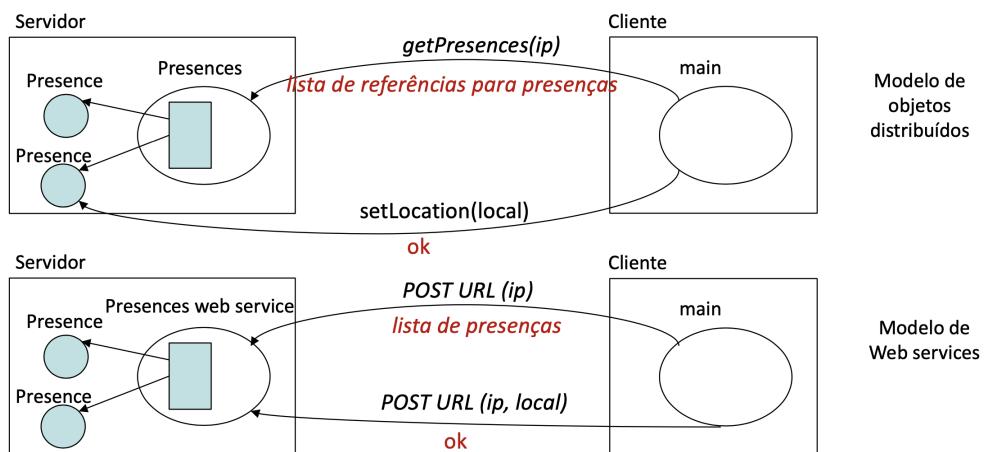
De uma forma superficial, a interação entre cliente e servidor é muito semelhante ao RMI (por exemplo Java RMI), onde um cliente usa uma referência de um objeto remoto para invocar uma operação nesse objeto remoto

Referências para objetos remotos vs URLs

No entanto, no modelo de objetos distribuídos, os objetos podem criar outros objetos remotos dinamicamente e retornar referências remotas para eles.

Os web services não podem criar instâncias de objetos remotos

Efetivamente, um Web service corresponde a um único objeto remoto.



### **RPC Web services:**

#### Desvantagens:

- Pouco eficiência

#### Vantagens:

- Protocolos web são cada vez mais comuns e fáceis de implementar

Privilegia a interoperabilidade em detrimento da eficiência

Ideal para a ligação entre sistemas de domínios diferentes ou para sistemas diversos na mesma empresa

---

### **REST Web Services**

Serviços Web que satisfazem os princípios, ou restrições, REST (*REpresentational State Transfer*)

Os princípios REST caracterizam arquiteturas escaláveis, desacopladas e extensíveis

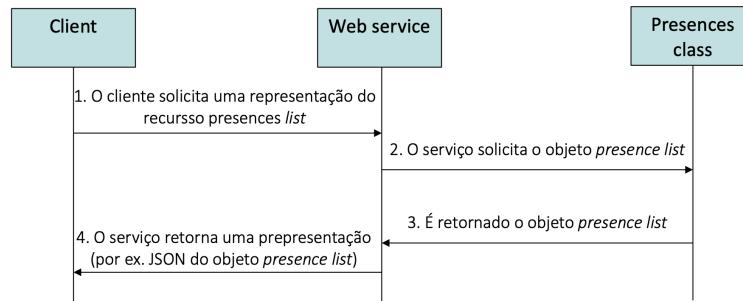
Usam tanto quanto possível os protocolos básicos da Web e respeitam os princípios nos quais estes se baseiam

REST Web services partilham os métodos standard definidos no HTTP

Qualquer aplicação web pode ser um serviço Web

- Não há necessidade de distinguir entre sites para *browsing* e sites para consumo programático.
- Existem representações alternativas para os recursos
- Expõe os recursos como sendo algo que pode ser usado por outras aplicações para objetivos muito diversos (até ser consumido por *browser*)

### REST - *REpresentational State Transfer*



### Princípios/regras RESTful Web services:

- Client-server
- Stateless
- Cached
- Uniform Interface
- Layered system

### Modelo Cliente-servidor:

O sistema deve ser composto por clientes e servidores.

- Separação de papéis entre clientes e servidores
- Servidores geralmente lidam com lógica e persistência
- Os clientes geralmente lidam com a interface e a experiência do utilizador

Servidores podem ser, no entanto, clientes de servidores

Pode existir muitos tipos diferentes de clientes (páginas web, aplicações móveis, outros servidores, etc.) que acedem o mesmo servidor e cada cliente pode evoluir independentemente

### **Stateless:**

- A comunicação entre clientes e servidores deve ser *stateless* ou sem estado
- cada pedido terá de incluir toda a informação necessária para que o pedido possa ser executado
- Se for necessário, cada pedido terá de incluir toda a informação necessária para que o servidor possa executar o pedido
- Maior Robustez, Simplicidade e Escalabilidade
- Maior overhead na comunicação

### **Cached:**

- Respostas dos serviços devem ser assinaladas como *cacheable* ou *non-cacheable*
- Melhor desempenho
- Em oposição a outros protocolos, como por exemplo o protocolo SOAP

***Uniform Interface:***

- Identificação de recursos
- Representação de recursos
- Mensagens auto-descritivas
- *Conectividade* (ou *Hypermedia as the Engine of Application State*)

**Recursos:**

- Qualquer entidade do sistema que mereça/necessite ser referenciada
- Algo que pode ter uma representação
- Algo sobre o qual um programa possa atuar
- Algo que pode ser referido por outros recursos
- Tem de ter um URI associado

Um REST Web service é *endereçável* porque expõe partes relevantes dos seus dados como recursos

Dado que os recursos são expostos através de URIs, um web service irá expor um URL por cada parte de informação que pretende expor.

Normalmente isto resulta num número de URLs muito grande e impossível de contabilizar

exemplos:

<https://api.exemplo.com/clientes>

-> a lista de clientes

<https://api.exemplo.com/clientes/123456>

-> o cliente cujo ID é 123456

<https://api.exemplo.com/clientes?sobrenome=Sousa>

-> o cliente cujo sobrenome é "Sousa"

<https://api.exemplo.com/galerias/{id}/fotos>

-> a lista de todas as fotos da galeria cujo ID é id

**Representações:**

Recursos tem de ter uma representação

Representações são estruturas de dados serializadas

XML, JSON, ..

Representações corresponde a informação sobre o estado do recurso

Pode haver várias representações para o mesmo recurso

XML vs JSON, PT vs Eng

Promove fraco acoplamento entre clientes e servidores

Facilidade de integração com sistemas legados

### Mensagens auto-descritivas:

Cada mensagem de pedido e resposta terá de incluir informação suficiente para que o receptor a possa interpretar isoladamente.

Associar um *media type* (e.g. *application/json*)

Usar os métodos HTTP (GET, POST, PUT , DELETE) conforme o seu significado

Resource	GET	PUT	POST	DELETE
Collection URI, such as <code>http://example.com/resources</code>	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.	Create a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation. <sup>[16]</sup>	Delete the entire collection.
Element URI, such as <code>http://example.com/resources/item17</code>	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	Replace the addressed member of the collection, or if it doesn't exist, create it.	Not generally used. Treat the addressed member as a collection in its own right and create a new entry in it. <sup>[16]</sup>	Delete the addressed member of the collection.

### exemplos:

Task	Method	Path	
Create a new customer	POST	/customers	
Delete an existing customer	DELETE	/customers/{id}	
Get a specific customer	GET	/customers/{id}	
Search for customers	GET	/customers	
Update an existing customer	PUT	/customers/{id}	

Twitter API      vs     

GET /direct_messages/show.json?id={id} POST /direct_messages/destroy.json?id={id} POST /direct_messages/new.json
GET /direct-messages/{id} POST /direct-messages DELETE /direct-messages/{id}

### Uniform Interface:

#### REST Web services vs RPC Web services

Qual a informação necessária para desencadear a invocação de um método remoto

- A referência do objeto/serviço no qual o método vai ser invocado
- A identificação do método a invocar
- Os parâmetros de invocação

Em REST

- A referência é o URL do recurso
- Os métodos são os métodos standard do HTTP
- Os parâmetros estão embbebidos no URL ou corpo da mensagem HTTP

RPC Web services

Operações

getUser()

REST Web services

- Recursos

addUser()	• Location
removeUser()	User URLs <a href="http://api.example.org/locations/us/ny">http://api.example.org/locations/us/ny</a> ou
updateUser()	<a href="http://api.example.org/locations?pais=us&amp;cidade=ny">http://api.example.org/locations? pais=us&amp;cidade=ny</a>
getLocation()	<a href="http://api.example.org/users/joao">http://api.example.org/users/joao</a>
addLocation()	Métodos standard do HTTP
removeLocation()	GET, POST, PUT, DELETE, ...
updateLocation()	

listUsers()	
listLocations()	
findLocation()	
findUser()	Verbos vs Nomes

### ***Conektividade:***

O estado da aplicação é exposto por meio de um URI

As aplicações cliente devem seguir os links nos recursos como forma de navegação no estado da aplicação (*connectedness*)

Evitar construir manualmente o URI

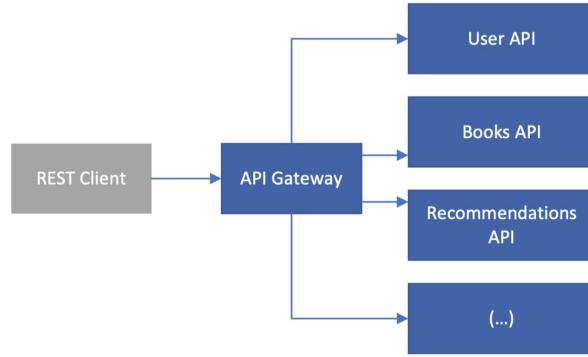
Melhora a robustez

```
{
  "id": 12,
  "firstname": "Roy",
  "lastname": "Fielding",
  "_links": {
    "self": {
      "href": "https://api.example.com/customers/12"
    },
    "orders": {
      "href": "https://api.example.com/orders?customerId=12"
    }
  }
}
```

### ***Layered System:***

O cliente deve conhecer apenas a camada com a qual comunica diretamente não deve conhecer nenhuma camada que possa ou estar atrás do servidor.

exemplo:



### REST examples:

*WordPress: Create a New Blog Post*

```

POST /wp/v2/posts
{
  "title": "7 Things You Didn't Know about Star Wars",
  "content": "George Lucas accidentally revealed that..."
}

```

*Movie App: Create a Movie*

```

POST /api/movies
{
  "title": "The Empire Strikes Back",
  "director": "Irvin Kershner",
  "releaseYear": 1980,
  "genre": "Sci Fi"
}

```

```

POST http://127.0.0.1:8080/PresenceRESTWS/webresources/presences
Host: 127.0.0.1:8080
Content-Type: text/xml; charset=utf-8
Content-Length: ...

{ "presenceId": "193.137.1.2",
  "locations": [
    {
      "lastSeenAtLocation": "Apr 27,2018 16:33",
      "locationId": "DSI"
    },
  ]
}

```

```

GET http://127.0.0.1:8080/PresenceRESTWS/webresources/presences/193.137.1.2
Host: 127.0.0.1:8080
Content-Type: text/xml; charset=utf-8
Content-Length: ...

Resposta:
{
  "presenceId": "193.137.1.2",
  "locations": [
    {
      "href": "http://127.0.0.1:8080/PresenceRESTWS/webresources/presences/93.137.8.0/locations"
    }
  ]
}

```

```

GET http://127.0.0.1:8080/PresenceRESTWS/webresources/presences/93.137.8.0/locations
Host: 127.0.0.1:8080
Content-Type: text/xml; charset=utf-8
Content-Length: ...

Resposta:
{
  "lastSeenAtLocation" : "Apr 27,2018 16:33",
  "locationId" : "Azurem"
}

```

### Análise:

Argumentos contra a conceção non- REST

Não satisfazem a arquitetura Web, particularmente a questão da

## Cache e da Interface Uniforme

Pior escalabilidade, robustez e maior custos de coordenação

O protocolo SOAP “abusa” do HTTP ao tratar o HTTP apenas como protocolo de transporte

Organizar uma aplicação distribuída num conjunto de recursos endereçáveis por URLs

Usar apenas mensagens HTTP para oferecer toda a funcionalidade da aplicação

---

## 5 - Segurança

Segurança informática é impedir os agressores de alcançar os seus objectivos através do acesso não autorizado ou do uso não autorizado de computadores ou de redes de computadores (*CERT Coordination Center*)

Segurança informática apenas considera ataques intencionais ao funcionamento do sistema. Não inclui ameaças ambientais, roubo de equipamentos, ou falhas de redes ou de software (exceto quando estas podem ser aproveitadas para fornecer acesso ou uso não autorizado)

### Motivação para os ataques:

- Desafio, notoriedade
- Benefícios financeiros
- Benefícios políticos
- Danos

### Safety vs Security:

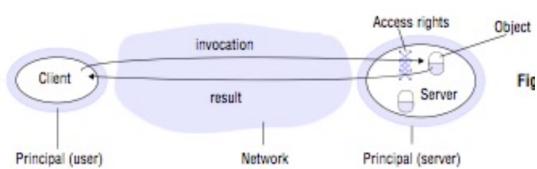
**Safety:** Prevenção de acidentes, que podem ou não envolver agentes humanos, mas que em qualquer caso não são intencionais. Exemplo: Garantir que o funcionamento de um veículo autónomo não coloca em risco os passageiros ou as pessoas e bens à sua volta. (Não é segurança informática)

**Security:** Prevenção de atividades maliciosas por parte de pessoas (assaltos, roubos, atividades terroristas, etc.). Exemplo: Evitar que uma entidade terceiro possa interferir ou ganhar controlo de um veículo autónomo. (Segurança informática)

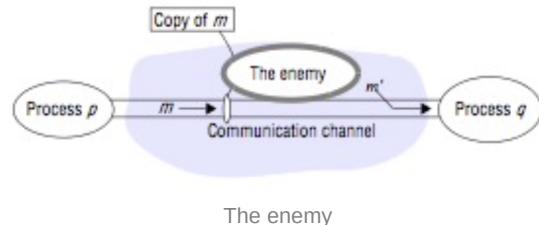
## Modelo de ameaças para um sistema distribuído (Unidade temática 2)

O inimigo ou adversário

- Pode enviar quaisquer mensagens para qualquer processo, possivelmente fazendo-se passar por uma outra entidade e acceder a recursos de forma não autorizada
- Pode ler, guardar, ou reenviar quaisquer mensagens enviadas pelos processos



Objects and principals



The enemy

## Política de Segurança

Define quem pode ter acesso a que recursos

Identifica as ameaças e define como evitá-las

É independente da tecnologia

"Porta da entrada deve estar sempre fechada e só pode ser aberta por pessoas autorizadas"

Suportada por mecanismos de segurança que permitam forçar o seu cumprimento

"Só pode abrir a porta da entrada quem tiver um cartão de acesso"

A identificação das potenciais ameaças é o ponto de partida para a implementação de um política de segurança

Assumir sempre que o pior pode acontecer

Um modelo realista de ameaças ajuda a conseguir o melhor equilíbrio entre riscos e proteção, entre segurança e conveniência ou entre segurança e custos

Considerar as possíveis motivações de atacantes, que tipos de ataques ou abusos poderão ser cometidos, que ações devem ser tomadas em cada caso

Respostas são diferentes para cada sistema e podem levar a mecanismos diferentes de segurança ou a níveis diferentes da sua aplicação

## Tipologias de ameaças

Ameaças gerais no **acesso aos recursos**:

- *Leakage*: Acesso à informação por partes não autorizadas
- *Tampering*: Alteração não autorizada da informação
- *Vandalism*: Interferir na operação normal de um sistema sem benefício próprio

Ameaças ao **acesso ao canal de comunicação**:

- *Eavesdropping*: Obter cópias de mensagens sem autorização
- *Masquerading*: enviar ou receber mensagens usando a entidade de um *principal*
- *Message Tampering*: Interceção e alteração de mensagens antes de as enviar ao *destinatário* (*Man-in-the-middle-attack*)
- *Replaying*: Guardar mensagens intercetadas e reenviá-las mais tarde
- *Denial of Service (DoS)*: Sobrecarregar um serviço com pedidos de forma a inviabilizar a sua operação

## Ameaças de código móvel

Programas podem ser carregados a partir de servidores remotos e integrados em processos locais

Interfaces e recursos internos são expostos a ataques desses programas

Estas situações são normalmente geridas por gestores de segurança local que restringe o acesso por programas externos aos recursos locais

Java security manager (Java), Web engine (javascript)

## Requisitos de segurança

**Autenticação:** Garantir a identidade no acesso a um serviço ou que quem enviou uma mensagem foi mesmo quem diz ser

**Confidencialidade:** Garantir que apenas as entidades autorizadas têm acesso ao conteúdo da mensagem

**Integridade:** Garantir que o conteúdo de uma mensagem não pode ser alterado

**Não-repúdio:** Evitar que uma entidade possa negar o envio ou recepção de uma mensagem

**Controlo de acessos:** Garantir que apenas as entidades autorizadas têm acesso a um recurso

**Disponibilidade:** Garantir que a disponibilidade de um recurso não pode ser intencionalmente posta em causa.

## Segurança

**Principal:** Entidade que é suposto ter acesso aos recursos

Nomes padrão ajudam a clarificar a descrição de protocolos e facilitam a identificação de vulnerabilidades

**Familiar names for the protagonists in security protocols:**

Alice	First participant
Bob	Second participant
Carol	Participant in tree- and four-party protocols
Dave	Participant in four-party protocols
Eve	Eavesdropper
Mallory	Malicious attacker
Sara	A server

## Mecanismos de Segurança

1. Criptografia
2. Certificados Digitais
3. Assinatura Digital

## Criptografia

Processo de cifrar uma mensagem de forma a que o seu conteúdo não seja acessível a qualquer entidade (“escrita secreta”)

Base para muitas das tecnologias de segurança

Criptografia pressupõe a existência de segredos que apenas são conhecidos pelos *principals*, evitando assim que terceiros tenham acesso à informação

Segredo é uma chave secreta usada num algoritmo conhecido

Toda a gente pode tentar quebrar o algoritmo. A sua validade é atestada pela sua resistência a esta exposição pública.

Chaves secretas podem ocasionalmente ser comprometidas mas podem ir sendo substituídas com frequência, sem que isso ponha em causa o algoritmo

#### **Criptografia de chave secreta (Simétrica):**

Emissor e receptor partilham uma mesma chave que não pode ser revelada a ninguém

#### **Criptografia de chave pública (Assimétrica):**

Cada *principal* possui duas chaves geradas em conjunto: uma privada que não pode ser revelada a ninguém e uma pública que é para divulgação pública

Tipicamente, algoritmos de chave pública usam 100 a 1000 vezes mais poder de processamento

---

$K_A$	Alice's secret key
$K_B$	Bob's secret key
$K_{AB}$	Secret key shared between Alice and Bob
$K_{A\text{priv}}$	Alice's private key (known only to Alice)
$K_{A\text{pub}}$	Alice's public key (published by Alice for all to read)
$\{M\}_K$	Message $M$ encrypted with key $K$
$[M]_K$	Message $M$ signed with key $K$

---

Cryptography notations

#### **Criptografia de chave secreta (simétrica):**

**Alice e Bob partilham uma chave comum**

Chave é um segredo partilhado entre ambos

Mesma chave é usada para cifrar e decifrar mensagens

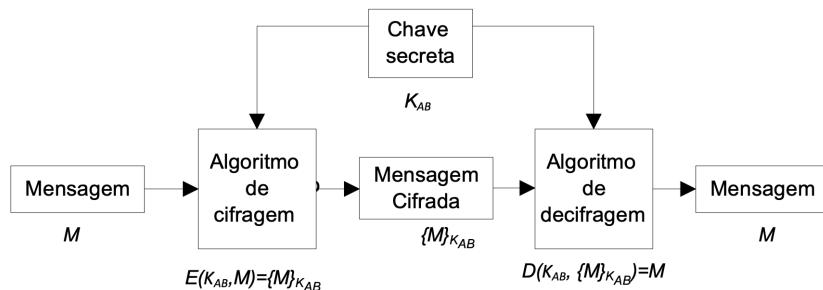
$$\begin{aligned} E(K_{AB}, M) &= \{M\}_{KAB} \\ D(K_{AB}, \{M\}_{KAB}) &= M \end{aligned}$$

Se a mensagem incluir um *checksum*, pode verificar-se se a integridade foi mantida

**Confidencialidade:** Garantir que apenas as entidades autorizadas têm acesso à mensagem

**Integridade:** Garantir que o conteúdo de uma mensagem não pode ser alterado

**Cenário:** Alice pretende enviar uma mensagem confidencial a Bob. Alice e Bob partilham uma chave secreta  $K_{AB}$ .



### Chave secreta na criptografia simétrica:

Keyspace é a gama de valores possíveis para as chaves e depende do tamanho da chave (número de bits)

Chave apenas é usada por sistemas computacionais, não por Humanos

Chaves maiores são mais seguras porque tornam mais difícil pesquisa exaustiva do espaço de soluções (*Brute force attack*)

Chaves maiores demoram mais tempo a realizar a cifragem/decifragem

Tamanho utilizado depende do domínio de aplicação

64 bits: Proteção básica

128 bits: Tamanho comum no SSL e suficiente para a maior parte das aplicações

168 ou 256 bits; Sistemas com requisitos especiais

### Limitações da criptografia chave secreta:

Grande desvantagem da criptografia simétrica resulta da necessidade de partilha da chave entre uma ou mais entidades

Muitas chaves para diferentes contextos de comunicação

Como partilhar uma chave com outra entidade através da rede (canal não seguro)

Como é que se distingue cópias de mensagens (em casos de *Replaying*); Não suporta não-repúdio

### Criptografia de chave pública (assimétrica):

Grande limitação dos algoritmos de chave secreta é obrigarem à partilha inicial de um segredo entre as partes que pretendem comunicar.

Não existe forma segura de partilhar o segredo

Necessário partilhar um segredo com cada uma das entidades com as quais se pretenda comunicar

Os algoritmos de chave pública permitem resolver este problema ao permitirem trocas seguras de informação entre entidades que não partilham à partida qualquer chave secreta entre si

**Alice gera um par de chaves que estão associadas uma à outra**

Chave privada da Alice ( $K_{APriv}$ )

Deve ser mantida secreta

Chave pública da Alice ( $K_{APub}$ )

Deve ser amplamente divulgada a todas as entidades com que se pretenda comunicar

Não é possível determinar a chave privada com base na chave pública

Mensagens cifradas com uma chave podem ser decifradas com a outra

Principal algoritmo é o RSA (Rivest, Shamir and Adleman)

Publicado em 1977

O comprimento da chave é variável

Atualmente 1024 bits é considerado o mínimo

Tamanhos habituais para as chaves 1,024 a 4,096 bits

**Tem como base funções matemáticas one-way:**

Muito fácil realizar a operação num determinado sentido, mas atualmente computacionalmente complexo realizar a operação inversa sem ter toda a informação inicial

Em particular, é simples gerar o par de chaves mas é atualmente computacionalmente inviável gerar a chave privada apenas com base na chave pública

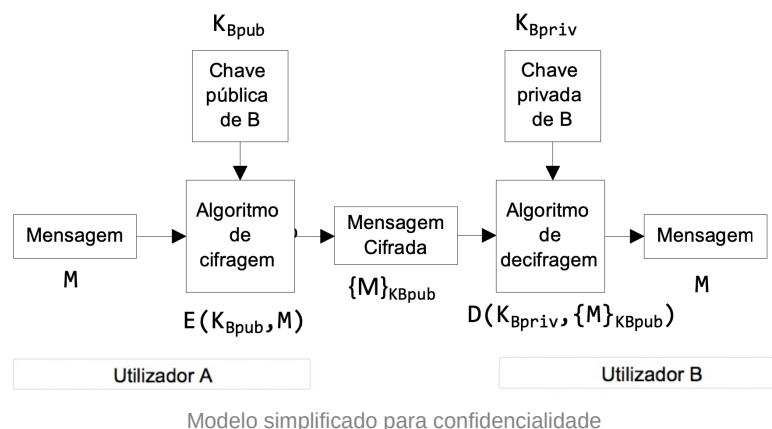
**Criptografia de chave pública permite suportar dois serviços de segurança:**

Confidencialidade

Se uma mensagem for cifrada com a chave pública da Alice, o que pode ser feita por qualquer entidade, então a mensagem apenas poderá ser decifrada com chave privada da Alice a que só ela tem acesso

Assinatura Digital (integridade e não-repúdio)

Se a mensagem for cifrada com a chave privada da Alice, o que só ela pode fazer, então qualquer entidade pode decifrar a mensagem usando a chave pública da Alice e confirmar assim que a mensagem tinha sido cifrada pela Alice e não foi alterada



Modelo simplificado para confidencialidade

RSA é um algoritmo lento a cifrar dados

Utilização de funções de números muito grandes e chaves de grande dimensão (+1024 bits) implicam processamentos computacionalmente complexos que são muito mais lentos do que os da criptografia simétrica.

O mais habitual é a utilização combinada de criptografia de chave secreta e criptografia de chave pública

RSA é usado apenas para a troca inicial de uma chave secreta de forma segura/confidencial e, a partir daí, toda a encriptação de dados é realizada com base em algoritmos de criptografia simétrica que são muito mais rápidos; Exemplo: Protocolo TSL (*Transport Layer Security*)

## Assinatura Digital

**Autenticidade e integridade:** O destinatário deve poder comprovar que o documento recebido é mesmo o que foi assinado sem que depois tenham sido feitas alterações

**Não falsificável:** Prova que o assinante, assinou intencionalmente aquele documento e não permite que a assinatura seja transferível para outro documento

**Não-repúdio:** Quem assina não pode alegar de forma credível que não assinou o documento

Assinatura digital não é assinatura digitalizada!

Assinatura digital não é confidencialidade!

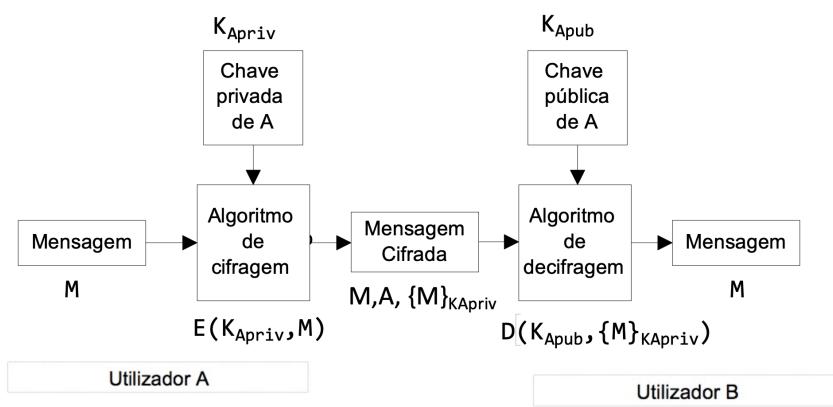
Exemplos de aplicação

Mensagens de correio eletrónico

Distribuição de software

Distribuição de documentos que atestam algo

...



Modelo simplificado para integridade e não repúdio

### Cifrar uma mensagem com a chave privada garante:

Integridade

Autenticidade

Não repúdio

Mas cifrar mensagens longas com criptografia assimétrica é pouco eficaz porque exige muito tempo de processamento

**Solução:** Produzir um sumário da mensagem e assinar apenas esse sumário

### **Assinatura digital uKliza sumários dos documentos:**

- *Message Digest ou secure hash*
- O sumário de uma mensagem representa-se por  $H(M)$ .
- Dado um bloco de dados produz um sumário pequeno e de tamanho fixo independentemente do tamanho da mensagem
- Tamanho epico do sumário 128-256 bits

### **Propriedades de uma função de geração de sumários:**

Dada uma mensagem  $M$  é simples calcular o seu *Message Digest*  $H(M)$

Dado  $H(M)$  é inviável calcular  $M$

Não é possível determinar a mensagem com base no sumário

Dado  $M$  e  $H(M)$  é computacionalmente inviável gerar uma  $M'$  tal que  $H(M')=H(M)$

Uma alteração nem que seja num único bit da mensagem  $M$  deve ser suficiente para produzir um  $H(M)$  substancialmente diferente

### **Exemplos de algoritmos de geração de sumários:**

MD5- Message Digest 5

Produz um sumário de 128 bits

SHA – Secure Hash Algorithm

Inspirado no MD5

Produz um sumário de 160 bits

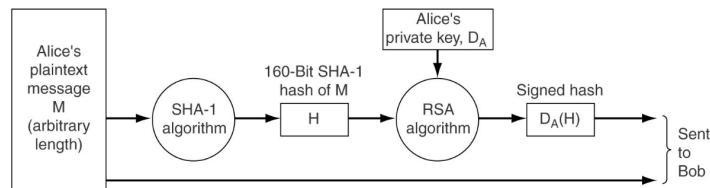
Suporta sumários de 224, 256 e 512 bits

### **Assinatura digitais com chaves públicas:**

Combina o conceito de sumário (*digest*) com o de criptografia assimétrica:

O sumário é gerado

O sumário é codificado com a chave privada de quem envia e constitui a assinatura

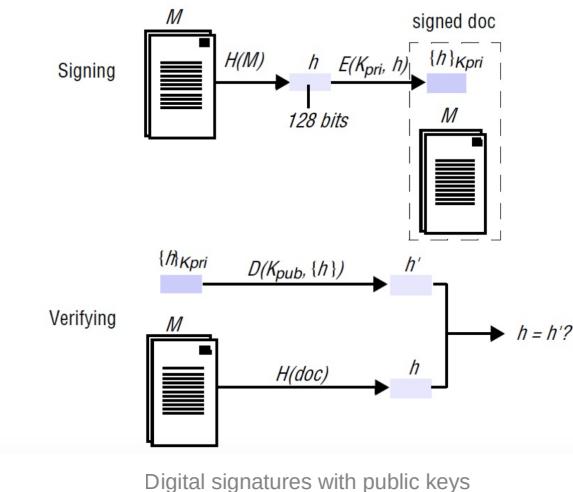


Alice calcula um sumário  $H(M)$  do documento  $M$

Alice cifra o sumário com a sua chave privada ( $K_{APriv}$ ), adiciona o resultado à mensagem  $M$  e envia ambos. Qualquer pessoa pode ver o conteúdo da mensagem. Qualquer pessoa que receba a mensagem pode verificar a assinatura da mesma.

Bob recebe a mensagem  $(M, \{H(M)\}_{K_{APriv}})$ , extrai  $M$  e calcula o respetivo  $H(M)$  com o mesmo algoritmo utilizado por Alice.

Bob decifra  $\{H(M)\}_{K_{APriv}}$  utilizando a chave pública de Alice e compara o resultado com o  $H(M)$  anteriormente calculado. Se coincidirem, a assinatura é válida e Bob sabe que o emissor foi a Alice e que documento não sofreu alterações.



### Problema de gestão de chaves:

Como ter a certeza de que a chave pública de outra entidade com quem se vai comunicar é de facto dessa entidade?

Certificados são documentos digitais que atestam a associação entre uma chave pública e um indivíduo ou entidade

Como qualquer documento de identificação, um certificado:

Fornece identificação de uma entidade

É resistente à falsificação

É possível verificar que foi emitido por um entidade oficial e de confiança

## Certificados digitais

### Elementos habituais num certificado digital:

- Chave pública
- Nome da entidade que é certificada (proprietário da chave)
- Nome da “autoridade” certificadora
- Número de série
- Data de emissão e validade
- Outra informação de certificação (ID, fotografia, etc.)
- Assinatura digital de entidade emissora (assinatura digital do certificado)

I hereby certify that the public key 19836A8B03030CF83737E3837837FC3s87092827262643FFA82710382828282A belongs to Robert John Smith 12345 University Avenue Berkeley, CA 94702 Birthday: July 4, 1958 Email: bob@superdupernet.com
SHA-1 hash of the above certificate signed with the CA's private key

Para que um certificado seja útil, é necessário que a sua estrutura corresponda a um formato normalizado, de modo que, independentemente de quem emitiu o certificado, a sua informação possa ser lida e usada corretamente.

As normas X.509 definem o standard utilizado globalmente para o formato dos certificados digitais

### Certificados atestam identidade, mas:

Quem são as entidades de confiança que vão emitir os certificados?

Como é que um certificado pode ser verificado para se comprovar a sua legitimidade?

### Public Key Infrastructure (PKI):

Consiste num conjunto de entidades, procedimentos, normas e protocolos de modo a suportar um sistema de certificação digital

Composta por Autoridades de Certificação, ou *Certification Authorities* (CA)

PKI desempenha um papel de intermediário que proporciona credibilidade e confiança em transações que utilizam certificados digitais

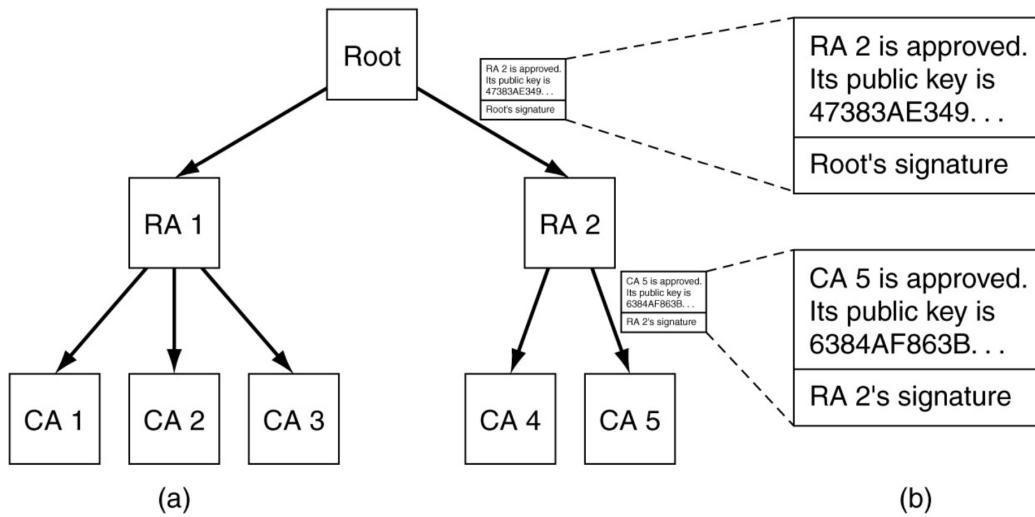
### Validação dos Certificados digitais:

Um certificado digital é assinado por uma autoridade de certificação “Certificate Authority” ou Certification authority (CA)

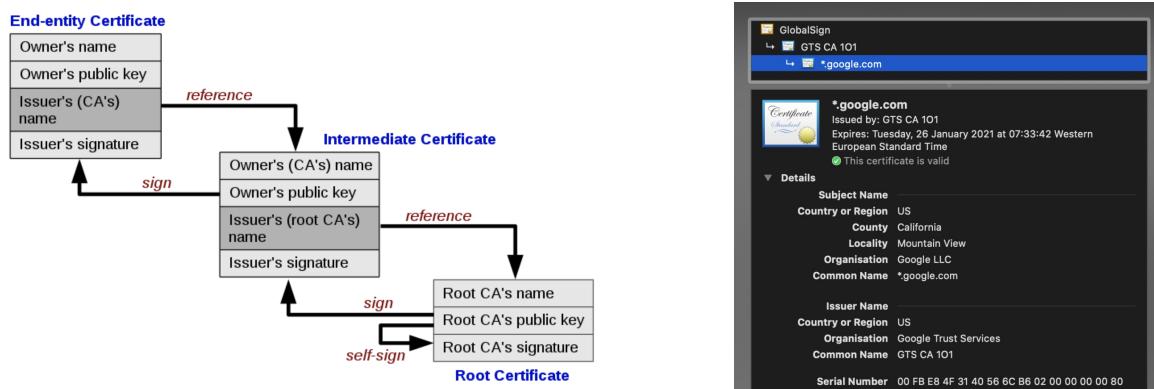
Assinatura pela CA permite confirmar que certificado é válido e portanto a respetiva chave pública está associada à entidade referida no certificado

Certificados podem ser trocados e verificadas sem necessidade de contactar nenhuma entidade terceira

Problema fica reduzida à confiança no CA



Hierarquias de classificação



### **Certificados de topo da hierarquia:**

Podem existir vários CAs no topo da hierarquia da infraestrutura de chaves públicas

O browser ou o sistema operativo mantêm uma lista de certificados de confiança, correspondentes a CAs de topo que lhes permitem fechar a cadeia de validação dos certificados digitais

Estes certificados vêm pré-instalados com o software e é neles que se baseia a hierarquia de confiança

## **Transport Layer Security (TLS) Protocol**

Permite criar um canal seguro sobre um meio de comunicação não seguro

TLS é o sucessor dos protocolos Secure Sockets Layer (SSL)

*The Secure Sockets Layer version 3.0 (SSLv3), as specified in RFC 6101, is not sufficiently secure. This document requires that SSLv3 not be used. The replacement versions, in particular, Transport Layer Security (TLS) 1.2 (RFC 5246), are considerably more secure and capable protocols.*

*Internet Engineering Task Force (IETF). Request for Comments (RFC) 7568. June 2015*

Protocolos TLS foram atualizados no RFC 6176 em Março de 2011, deixando de ser compatíveis com SSL

SSL/TLS são frequentemente usados de forma indistinta

Protocolo mais usado para garantir a autenticação de web sites e a confidencialidade na comunicação com web sites

Usado igualmente em muitos outros serviços como IMAP, SMTP, POP, Secure Shell ou vídeo-conferência.

Suporta autenticação do cliente e do servidor (servidor é mais comum)

### **O protocolo define duas fases principais:**

Estabelecimento da comunicação (TLS/SSL Handshake)

Comunicação (TLS/SSL Record Protocol)

TLS funciona sobre TCP mas existe uma versão para UDP (DTLS)

### **Estabelecimento da comunicação (handshake):**

O cliente e o servidor trocam entre eles os seus certificados (X.509) para assegurar a sua identidade (autenticação, opcional para o cliente)

O cliente gera chave para cifragem dos dados

Seguidamente é negociado qual vai ser o algoritmo usado na cifragem e a função de hash que será usada para a verificação da integridade. Nem todos os sistemas suportam os mesmos algoritmos

### **Fase de comunicação:**

Usa chave simétrica escolhida durante o estabelecimento da comunicação (handshake)

Oferece uma abstração de canal seguro:

- Confidencialidade nas mensagens
- Integridade da mensagem
- Autenticação das partes

Fornece ainda serviços de compressão

## HTTP Secure (HTTPS)

TLS pode ser usado como base para protocolos de aplicação

Uma utilização muito comum é a combinação do HTTP com o TLS

Por convenção, um URL que tenha de ser acedido por HTTP sobre uma ligação TLS começa por h\_ps: em vez de h\_p:

O servidor que aceite ligações TLS funciona numa porta própria (normalmente 443) da que é usada pelo servidor de HTTP (80 por defeito)

Na mesma máquina podemos ter um servidor de HTTP e um de HTTPS

Informação de acesso restrito deve ser servida pela porta HTTPS

### Comunicação HTTPS:

- O *browser* envia dados iniciais (versão TLS, ...)
- O servidor responde com o seu certificado digital
- O *browser* verifica se:
  - O certificado é válido
  - O CA é reconhecido
  - A assinatura do CA é válida
  - O domínio do servidor corresponde ao do certificado
- Em caso de sucesso o *browser* gera uma chave de sessão e envia essa chave para o servidor (cifrada com chave pública do servidor)
- O inicio da sessão TLS é negociado e as mensagens passam a ser codificadas com a chave de sessão
- No final a chave de sessão é eliminada

### Exercícios:

1. Descreva as principais ameaças no acesso de a um serviço de *homebanking* na situação em que não é utilizado nenhum mecanismo de segurança. Identifique as principais ameaças e políticas de segurança e os mecanismos de segurança adequados.