Sistemas distribuídos (1. ciclo)

Capítulo 1

Introdução

Nota prévia

- A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 5th Edition, 2011

Organização do capítulo

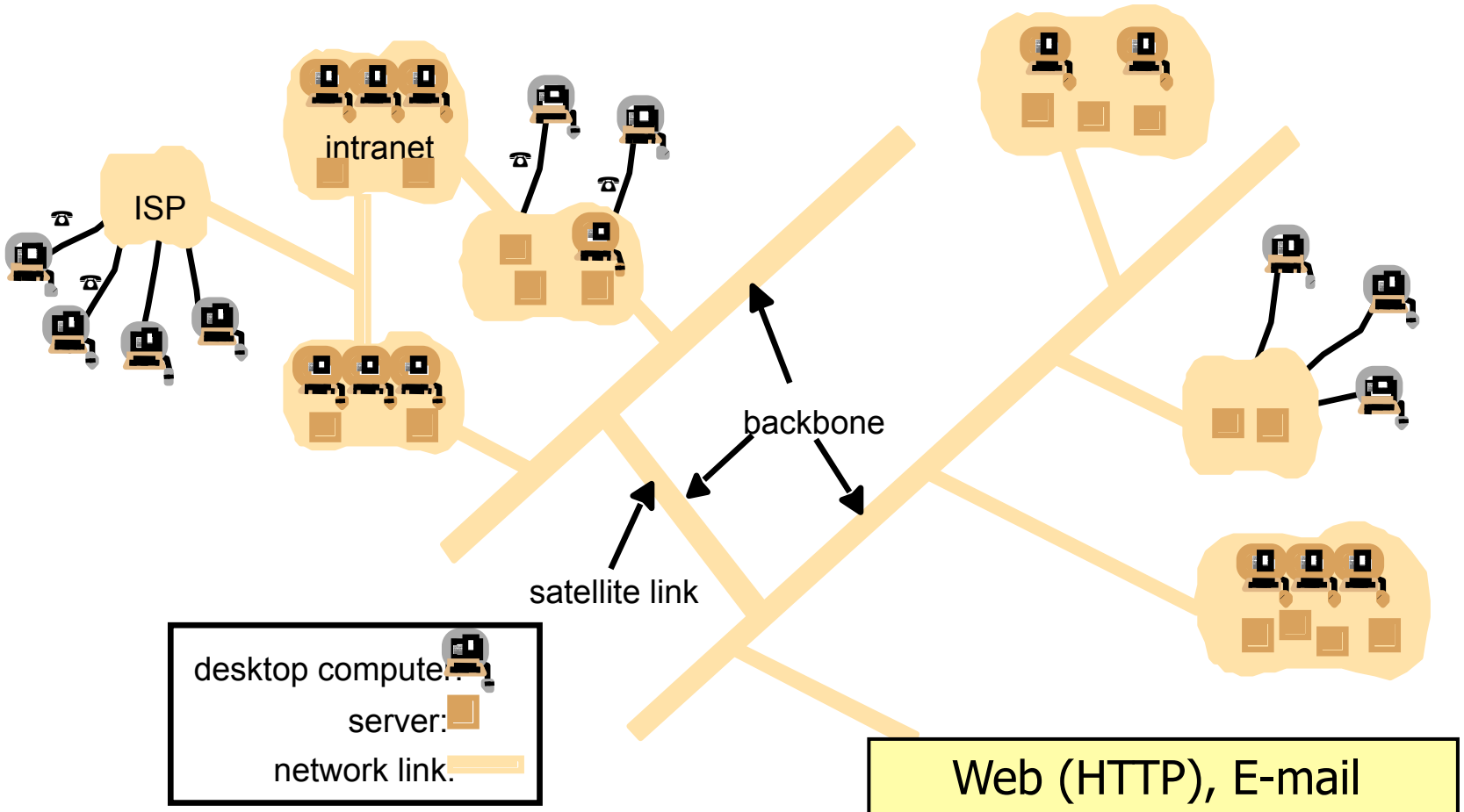
- Definição, exemplos
- Características essenciais dos sistemas distribuídos
- Desafios: heterogeneidade, abertura, segurança, escala, falhas, concorrência, transparência

Definição de sistema distribuído

- Um sistema distribuído é um conjunto de componentes hardware (nós, hosts, máquinas ou computadores) e software interligados através de uma infra-estrutura de comunicações (geralmente uma rede de computadores ou um bus especial, …), que cooperam e se coordenam entre si apenas pela troca de mensagens, para execução de aplicações distribuídas (pelos diferentes computadores)

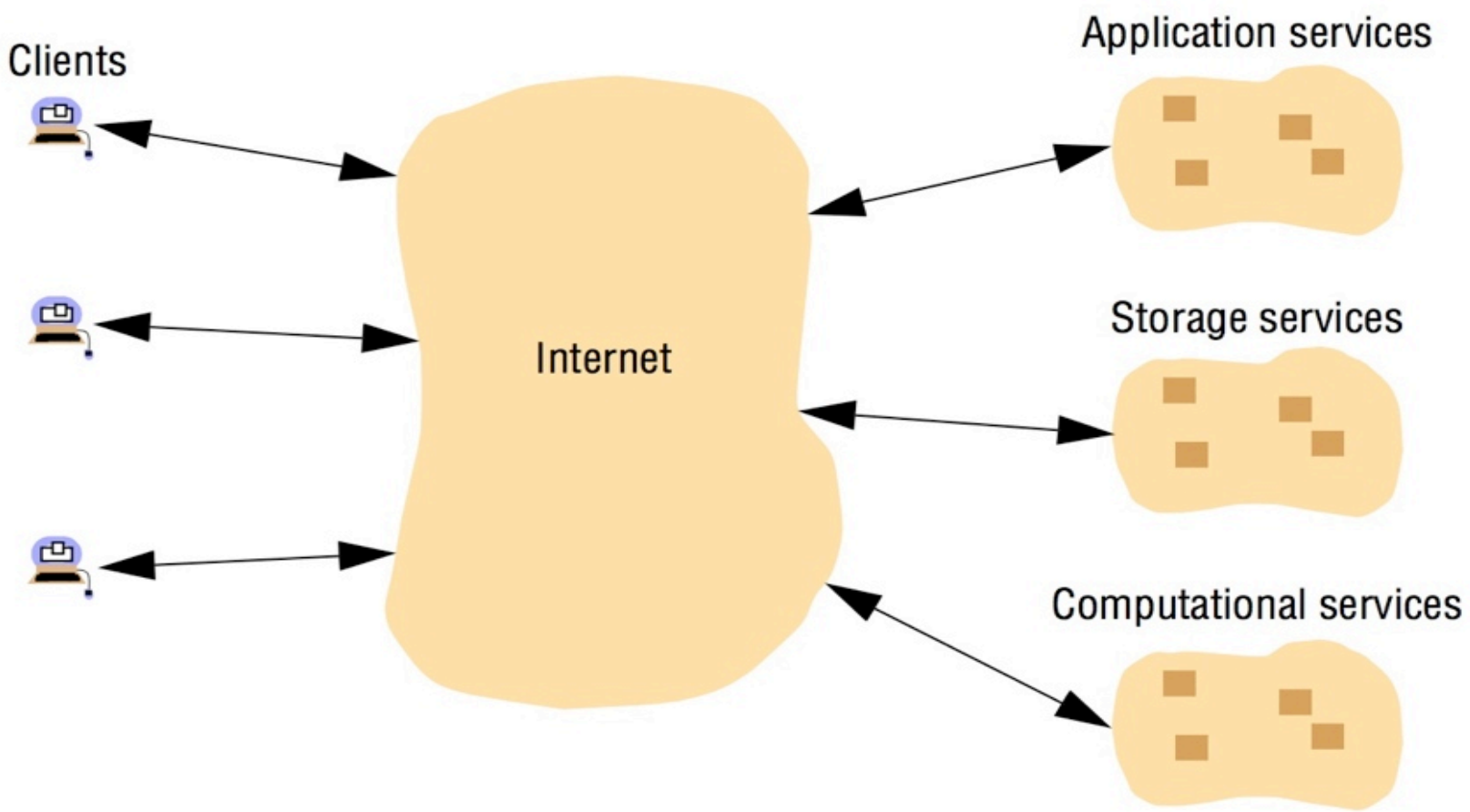
- Assim, no âmbito desta cadeira, não estamos interessados nos sistemas que cooperam e se coordenam pela partilha de memória física comum

Exemplo: serviços na Internet



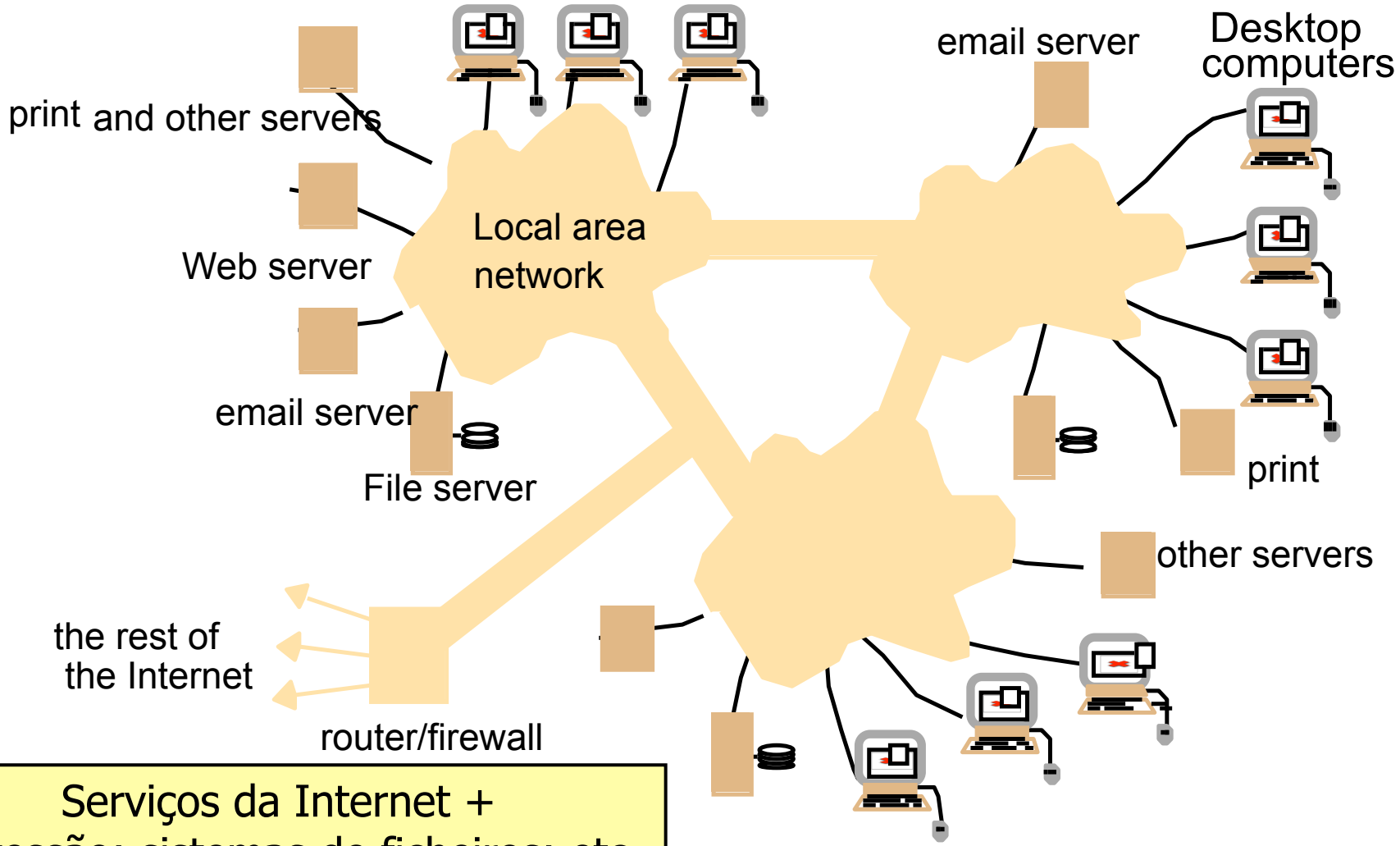
Web (HTTP), E-mail (SMTP); instant-messaging (e.g.Msn); VoIP (e.g.Skype); etc.

Exemplo: cloud computing



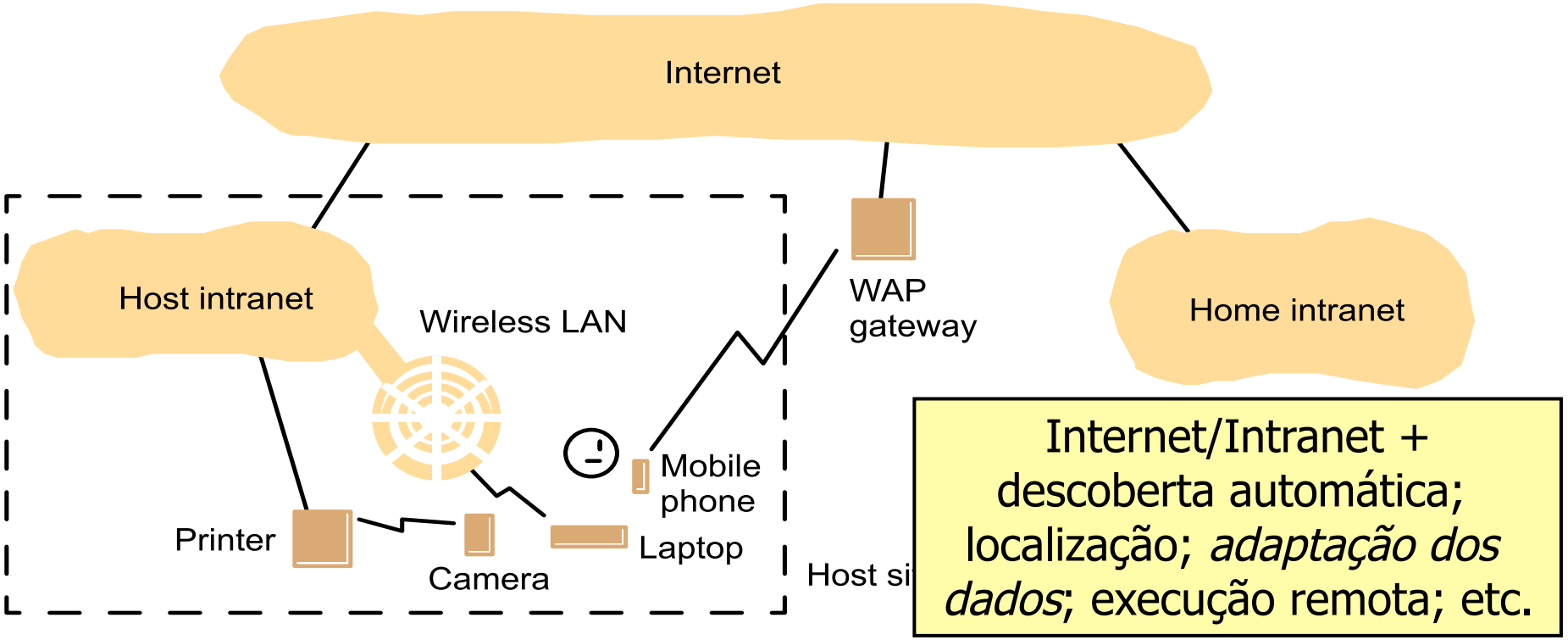
Exemplo: serviços em intranets

Intranets: redes isoladas fisicamente, redes isoladas logicamente (private virtual network), ligação à Internet através de firewalls, etc.



Serviços da Internet + impressão; sistemas de ficheiros; etc.

Exemplo: serviços em ambientes de computação móvel



- Dispositivos portáteis podem ser usados para efectuar computação
  - Múltiplos dispositivos portáteis
  - Diversas redes sem fios: GSM, UMTS, Wi-Max, Wi-Fi, Bluetooth

Exemplo: computação ubíqua

- Explosão no número de dispositivos existentes
  - Dispositivos associados a utilizadores
    - Computadores, telemóveis, etc. RF-ids ??
  - Dispositivos associados a objectos, animais
    - Dispositivos embebidos
    - RF-ids
    - Nós de redes de sensores

- Abre a possibilidade de criar novas aplicações distribuídas
  - Numa casa: controlo de luzes, frigorífico, TV, etc.
  - No supermercado: controlo de compras com RF-ids, etiquetas electrónicas
  - No ambiente: monitoração de condições atmosféricas em vinhas, florestas, etc.; localização em hospitais; criação de cercas virtuais; etc.
  - Etc.

- Abordado na discipliana de Sistemas de Computação Móvel e Ubíqua (2º ciclo)

Outros exemplos

- Sistemas de controlo de processos industriais em fábricas (por exemplo, linhas de montagem)
- Dispositivos ou máquinas especiais controlados através de conjuntos de computadores embebidos (por exemplo: um avião ou um carro, uma fábrica)
- Clusters de computadores interligados através de redes de alta velocidade para cálculo paralelo

Motivações dos Sistemas Distribuídos

- Acesso generalizado sem restrições de localização
  - Acessibilidade ubíqua (suporte para utilizadores *fixos*, móveis)
- Partilha dos recursos distribuídos pelos diferentes utilizadores
  - Exemplos: impressores, ficheiros
- Distribuição da carga – melhoria do desempenho
- Tolerância a falhas
- Flexibilidade e adaptabilidade
  - Decomposição de um sistema complexo num conjunto de sistemas mais simples

Vantagens potenciais

- Ubiquidade e mobilidade
  - pode ser utilizado independentemente da localização
- Custo inferior, ou pelo menos mais racional
  - várias pequenas máquinas em vez de uma só gigantesca
- Melhor desempenho
  - paralelismo real
- Maior disponibilidade
  - se uma máquina falha, as outras podem continuar
- Extensibilidade
  - crescimento incremental
- Simplicidade associada à modularidade física
  - pode ter processadores dedicados a funções
  - pode ser distribuído de acordo com a organização utilizadora

Características fundamentais

- Componentes do sistema executam de forma concorrente – paralelismo real
  - Necessidade de coordenação entre os vários componentes
- Falhas independentes das componentes e das comunicações
  - Impossível determinar se existe uma falha dum componente ou do sistema de comunicações
  - Necessidade de tratar as falhas
- Ausência de relógio global – existem limites para a precisão da sincronização dos relógios locais
  - Impossível usar relógios locais para ordenar globalmente todos os eventos

Implicações

- Nenhum componente tem uma visão exacta instantânea do estado global de todo o sistema
- Os componentes têm uma visão parcial do estado global do sistema
  - Os componentes do sistemas estão distribuídos e só podem cooperar através da troca de mensagens, as quais levam um tempo não nulo a propagarem-se
- Na presença de falhas, o estado global pode tornar-se incoerente, i.e., as visões parciais do estado global podem tornar-se incoerentes
  - Por exemplo, réplicas de um objecto podem ficar incoerentes

Desafios

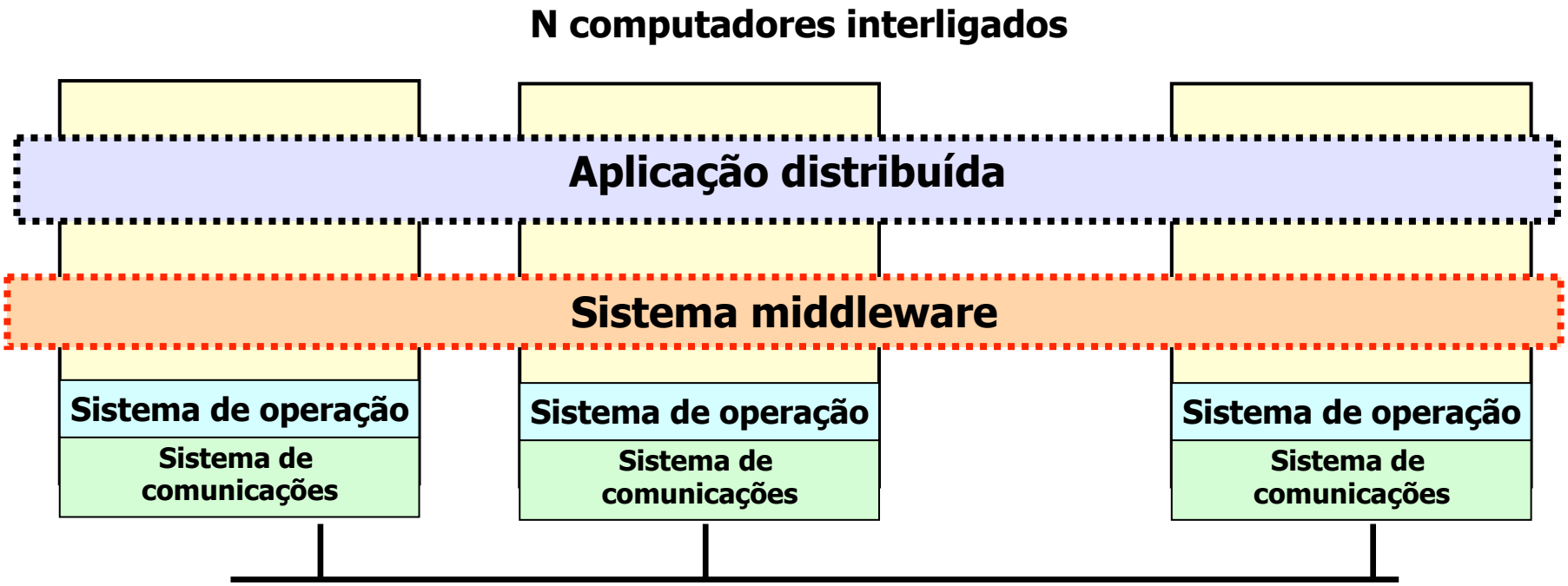
- Heterogeneidade
- Abertura
- Transparência
  - Concorrência
- Segurança
- Escala
- Tratamento das falhas

Heterogeneidade

- Do hardware: telemóveis, PDAs, portáteis (laptops), workstations (desktops), servidores, multiprocessadores, clusters (agregados), grids,...
  - Diferentes características dos processadores, da memória, da representação dos dados, dos códigos de caracteres,...
- Das redes de interligação e dos protocolos de transporte: Redes móveis (GSM, UMTS, CDMA), Wi-Max, WLANs, wired LANs, WANs, ..., TCP/IP, ...
- Do sistema de operação: Windows, MacOS, Mac, iOS, Android,...
  - Diferentes interfaces para as mesmas funcionalidades
- Das linguagens de programação....

- Construir um sistema único, integrado, coerente, etc. é geralmente um desafio gigantesco

Middleware



- Sistemas operativos
  - Interfaces heterogénea
  - Serviços básicos
- Sistema *middleware*
  - Interface homogénea
  - Serviços mais complexos (invocação remota: RMI, Web-services; comunicação em grupo: JGroups, etc)
  - Verdadeira interoperabilidade requer idênticos interface e protocolos

Máquina virtual

- Objectivo: permitir executar os mesmos programas em máquinas com diferentes características
- Fornece um ambiente de execução local uniforme, independentemente do hardware e sistema de operação
  - Programas escritos numa única linguagem (JavaVM) ou em múltiplas linguagens (Microsoft CLR)
- Num sistema distribuído permite simplificar o *deployment* de uma aplicação e a utilização de *código móvel* (applets, agentes)

Abertura

- A abertura de um sistema determina o modo como pode ser estendido e re-implementado
- Sistemas abertos
  - Interfaces e modelo (incluindo protocolos de comunicação) conhecidos
  - Evolução controlada por organismos de normalização independentes ou consórcios industriais
  - Permite a interoperação de componentes com diferentes implementações
- Sistemas proprietários
  - Podem ser modificados pelo seu "dono"

Vantagens e desvantagens de cada aproximação?

- Outras questões
  - Middleware suportando várias linguagens / Middleware integrado com uma só linguagem de programação

Transparência

- A transparência (da distribuição) é a propriedade relativa a esconder ao utilizador e ao programador das aplicações a separação física dos elementos que compõem um sistema distribuído
  - Simplicidade e flexibilidade são objectivos
- Em certas circunstâncias, a transparência é indesejável
  - Implementação complexa e desempenho baixo
  - Pode levar a inflexibilidade
- A transparência pode manifestar-se em diversos níveis e de várias formas

Alguns tipos de transparência

Acesso	Esconde as diferenças no modo de acesso a um recurso
Localização	Esconde a localização de um recurso
Mobilidade	Esconde a mudança de localização dos recursos e clientes
Replicação	Esconde que o recurso é replicado
Concorrência	Esconde que o recurso pode ser usado por vários utilizadores
Falhas	Esconde as falhas e o processo de recuperação
Desempenho	Esconde a reconfiguração para melhorar o desempenho

Concorrência

- O acesso simultâneo a um recurso por vários clientes pode causar problemas
- Necessário sincronizar as operações para que os dados permaneçam consistentes
  - Num único computadores, soluções possíveis são e.g. o uso de semáforos, monitores, transacções
  - Necessário implementar técnicas de sincronização num sistema distribuído

Segurança

- Necessidade de proteger os recursos e informação gerida num sistema distribuído
  - Recursos têm valor para os seus utilizadores
- Segurança tem três componentes:
  - Confidencialidade: indivíduos não autorizados não podem obter informação
  - Integridade: dados não podem ser alterados ou corrompidos
  - Disponibilidade: acesso aos dados deve continuar disponível
- Aspectos envolvidos
  - Autenticação dos parceiros
  - Canais seguros
  - Prevenção de ataques de "negação de serviço" (denial of service attacks)
  - Autenticação das plataformas de hardware/software
  - Segurança na presença de código móvel

Escala

- A escala de um sistema distribuído é o âmbito que o mesmo abrange assim como o número de componentes.
- A escala de um sistema tem várias facetas:
  - recursos e utilizadores
  - âmbito geográfico (rede local, país, mundo, ...)
  - âmbito administrativo (uma organização, inter-organizações)
- Um sistema de pequena escala é um sistema confinado a uma organização pequena / média, com poucos utilizadores e poucos componentes.
- Um sistema de grande escala é um sistema com muitos recursos ou utilizadores, ou cobrindo uma área geográfica significativa, ou envolvendo muitas organizações.

Escala

- A escala de um sistema distribuído é o âmbito que o mesmo abrange assim como o número de componentes.
- A escala de um sistema tem várias facetas:
  - recursos e utilizadores
  - âmbito geográfico (rede local, país, mundo, ...)
  - âmbito administrativo (uma organização, inter-organizações)
- Um sistema de pequena escala é um sistema confinado a uma organização pequena / média, com poucos utilizadores e poucos componentes.
- Um sistema de grande escala é [illegible] utilizadores, ou cobrindo uma ár[illegible] envolvendo muitas organizações

Data	Computadores	Servidores Web
Dez. 1979	188	0
Jul. 1989	130.000	0
Jul. 1999	56.218.000	5.560.866
Jul. 2006	439.286.364	88.166.395

fonte: http://www.zakon.org/robert/internet/timeline

Sistema escalável

- Um sistema capaz de escalar (escalável) é um sistema que continua eficaz quando há um aumento significativo do número de recursos e utilizadores
  - i.e., em que não é necessário alterar a implementação dos componentes e da forma de interacção dos mesmos
- Num sistema escalável, as seguintes propriedades devem ser verdadeiras:
  - Os recursos envolvidos devem aumentar linearmente com a dimensão do sistema
  - O aumento na dimensão dos dados/recursos não deve levar a uma diminuição significativa do desempenho – a redução não deve ser superior a O(log n)
    - Possível aproximação: recurso a estruturas hierárquicas (ex.: DNS)

Como lidar com a escala ?

- Para reduzir o número de pedidos tratados por cada componente
  - Divisão de um componente em partes e sua distribuição
  - Replicação e caching (problema da consistência entre réplicas e caches)
- Para reduzir dependências entre componentes
  - Meios de comunicação assíncronos
- Para simplificar o sistema
  - Uniformidade de acesso aos recursos e dos mecanismos de cooperação, sincronização, etc.
  - Meios de designação universais (independentes da localização e dos recursos)

Avarias, erros e falhas

- Os componentes de um sistema podem falhar, i.e., comportar-se de forma não prevista e não de acordo com a especificação devido a erros (por exemplo a presença de ruído num canal de comunicação ou um erro de software) ou avarias (mecanismo que que entra em mau funcionamento)

- Num sistema distribuído, as falhas são geralmente parciais (num componente do sistema) e independentes
  - Um componente em falha pode induzir uma mudança de estado incorrecta noutro componente, levando eventualmente o sistema a falhas, i.e., a ter um comportamento não de acordo com a sua especificação.

Como lidar com as falhas

- Detectar falhas
  - Possível: e.g.: mensagens corrompidas através de checksums
  - Pode ser impossível: Falha (crash) num computador remoto
    - Desafio: Funcionar através da suspeição das falhas
- Mascarar falhas (após a sua detecção)
  - Exemplos: retransmissão de mensagens, redundância

Como lidar com as falhas

- Tolerar falhas
  - Definição do comportamento na presença de falhas
    - Continuar a funcionar de acordo com a especificação recorrendo a componentes redundantes
    - Parar o sistema até falhas serem resolvidas
- Recuperação de falhas
  - Mesmo num sistema que tolere falhas é necessário recuperar os componentes falhados. Porquê?
  - Problema: recuperar estado do serviço, considerando que algumas operações podem ter sido parcialmente executadas

Para saber mais

- G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems – Concepts and Design, Addison-Wesley, 4th Edition, 2005

  Capítulo 1.

- G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems – Concepts and Design, Addison-Wesley, 5th Edition, 2011

  Capítulo 1.

Sistemas distribuídos – 1. ciclo

Capítulo 2

Arquitecturas e Modelos de Sistemas Distribuídos

Nota prévia

- A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

  G. Coulouris, J. Dollimore and T. Kindberg,
  Distributed Systems - Concepts and Design,
  Addison-Wesley, 4th Edition, 2005

Organização do capítulo

- Modelos arquitecturais
  - Arquitectura/camadas de software
  - Cliente/servidor, peer-to-peer, variantes
- Modelos fundamentais – usados para descrever propriedades parciais, comuns a todas as arquitecturas
  - Modelo de interacção
  - Modelo de falhas
  - Modelo de segurança

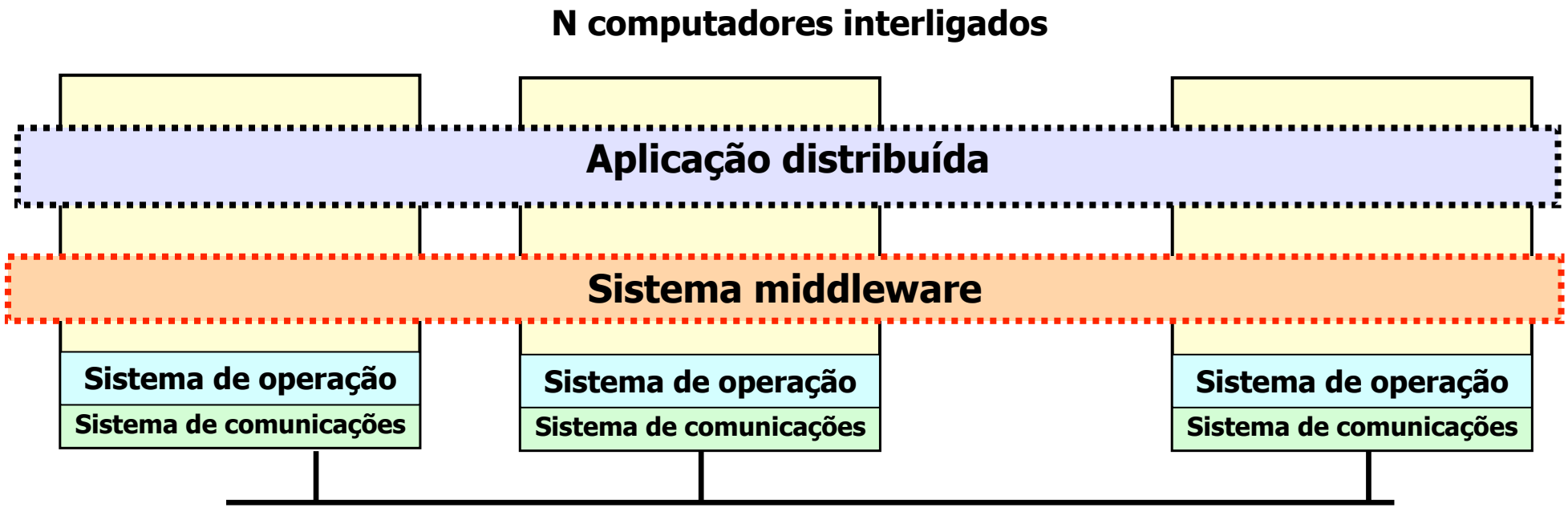
Terminologia

- **Processo**: programa a executar num computador
- **Objecto**: quaisquer dados manipulados por um processo (ex.: ficheiro, base de dados, objecto)
- **Serviço**: conjunto de operações disponibilizado para execução por processos remotos
  - Necessário definir a interface e o protocolo de acesso ao serviço
  - Um serviço pode ser fornecido por qualquer arquitectura discutida neste capítulo
- **Sistema**: conjunto de processos inter-relacionados a executar num ou em vários computadores

Contexto

- Camadas de software
  - Especifica quais os componentes de software num processo
  - Tem implicações na simplicidade da criação do software
- Arquitectura de software
  - Especifica como se organizam e quais as interacções entre os processos a executar numa máquina
- Arquitectura distribuída
  - Especifica como se organizam e quais as interacções entre os vários processos que compõem um sistema dstribuído
  - Tem implicações no desempenho, fiabilidade e segurança do sistema

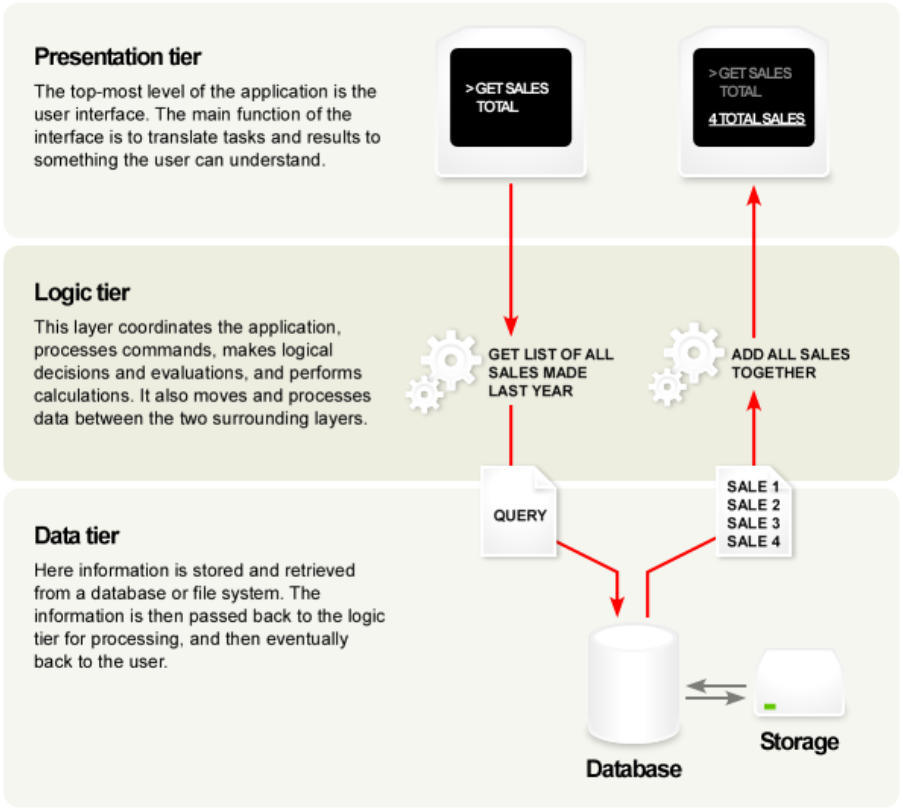
Camadas de software



- O *middleware* fornece uma interface homogénea e serviços mais complexos que os disponibilizados pelo sistema de operação
- Limitações: algumas funcionalidades apenas podem ser implementadas de forma eficaz com o conhecimento da semântica da aplicação, pelo que fornecer essa funcionalidade no sistema de middleware seria contraproducente (correcção de erros, segurança, etc.)

Arquitectura de software: modelo *three-tier*

- Em aplicações de acesso a sistemas de informação
- Arquitectura de três níveis (3-tier)
  - Apresentação
  - Aplicação
  - Dados
- Arquitectura típica: cliente/ servidor

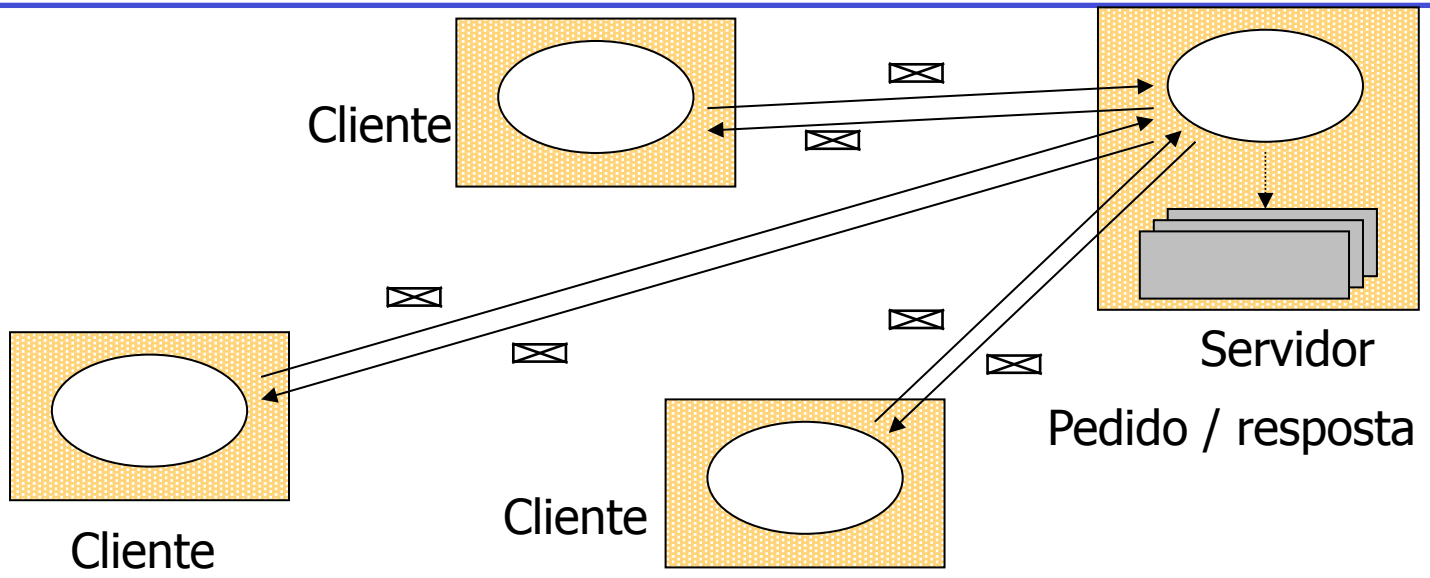


Arquitectura do sistema

- Arquitectura do sistema
  - Define os componentes do sistema distribuído, identificando as suas funções de forma simples e abstracta
  - Identifica a localização dos componentes numa rede
  - Identifica as inter-relações entre os componentes (incluindo os padrões de comunicação)
- A arquitectura tem implicações no desempenho, fiabilidade e segurança do sistema

- Arquitectura mais comum e usada na prática

Cliente/Servidor



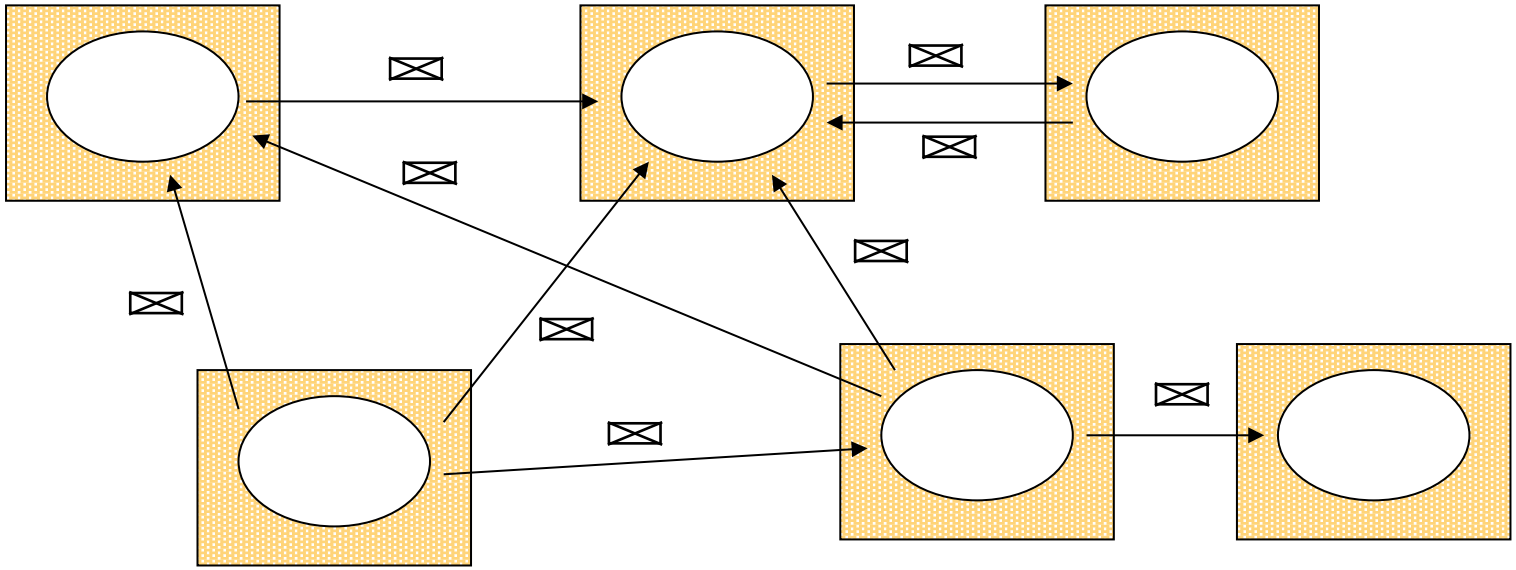
- Sistema em que os processos podem ser divididos em dois tipos, de acordo com o seu modo de operação:
  - **Cliente**: programa que solicita pedidos a um processo servidor
  - **Servidor**: programa que executa operações solicitadas pelos clientes, enviando o respectiva resultado

Cliente/Servidor: propriedades

- Arquitectura mais comum e usado na prática
- Positivo
  - Interacção simples facilita implementação
  - Segurança apenas tem de se concentrar no servidor
- Negativo
  - Servidor é um ponto de falha único
  - Não escala para além dum dado limite (servidor pode tornar-se bottleneck)
- Variantes podem eliminar/diminuir problemas

- Modelo alternativo para lidar com limitações do modelo cliente/servidor

Modelo *peer-to-peer*



- Todos os processos têm funcionalidades semelhantes
  - Durante a sua operação podem assumir o papel de clientes e servidores do mesmo serviço em diferentes momentos
- Exemplos: partilha de ficheiros, VoIP, edição colaborativa

Modelo *peer-to-peer*: propriedades

- Positivo
  - Não existe ponto único de falha
  - Melhor potencial de escalabilidade
- Negativo
  - Interacção mais complexa (do que num sistema cliente/servidor) leva a implementações mais complexas
    - Operações de pesquisa são complexas
  - Maior número de computadores envolvidos pode colocar questões relativas a heterogeneidade e segurança
- Apropriado para ambientes em que todos os participantes querem cooperar para fornecer uma dado serviço
  - Capacidade agregada >> capacidade individual

- Variantes do modelo cliente/servidor para lidar com limitações de escalabilidade e tolerância a falhas

Variantes do modelo cliente/servidor: servidor

- Cliente/servidor replicado
  - Existem vários servidores idênticos (i.e. capazes de responder aos mesmo pedidos)
  - Positivo
    - Permite distribuir a carga, melhorando o desempenho (potencialmente)
    - Não existe um ponto de falha único
  - Negativo
    - Manter estado do servidor coerente em todas as réplicas
    - Recuperar da falha parcial de um servidor

Variantes do modelo cliente/servidor: servidor

- Cliente/servidor particionado
  - Existem vários servidores com a mesma interface, cada um capaz de responder a uma parte dos pedidos (ex. DNS)
    - Servidor redirige cliente para outro servidor (iterativo)
    - Servidor invoca pedido noutro servidor (recursivo)
  - Positivo
    - Permite distribuir a carga, melhorando o desempenho (potencialmente)
    - Não existe um ponto de falha único
  - Negativo
    - Falha de um servidor impede acesso aos dados presentes nesse servidor

Variantes do modelo cliente/servidor: cliente

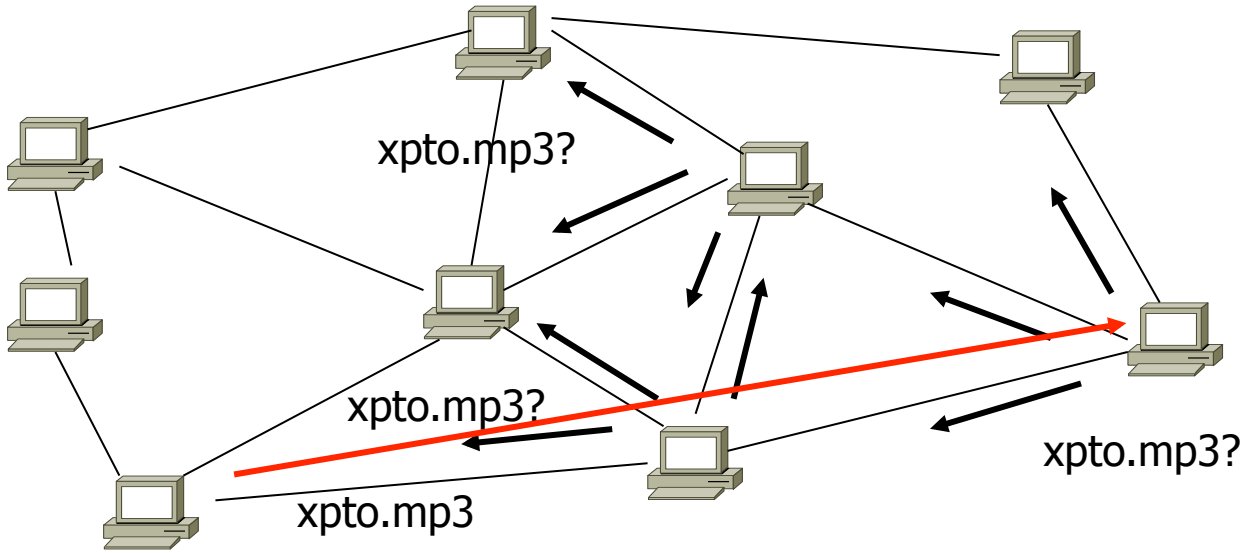
- Cliente leve (thin client)/servidor
  - O cliente apenas inclui um interface (gráfico) para executar operações no servidor (ex: browser)
  - Positivo:
    - Cliente pode ser muito simples
  - Negativo
    - Maior peso no servidor

Variantes do modelo cliente/servidor: cliente

- Cliente completo(estendido)/servidor
  - O cliente executa localmente algumas operações que seriam executadas pelo servidor
  - Positivo:
    - Permite funcionar quando não é possível contactar o servidor (recorrendo a *caching*)
    - Permite diminuir a carga do servidor e melhorar o desempenho
  - Negativo:
    - Implementação do cliente mais complexa
    - Necessário tratar da coerência dos dados entre o cliente e o servidor
    - E.g.: alguns sistemas aplicam o princípio à Web: Google Gears

- Variantes do modelo peer-to-peer têm diferentes propriedades

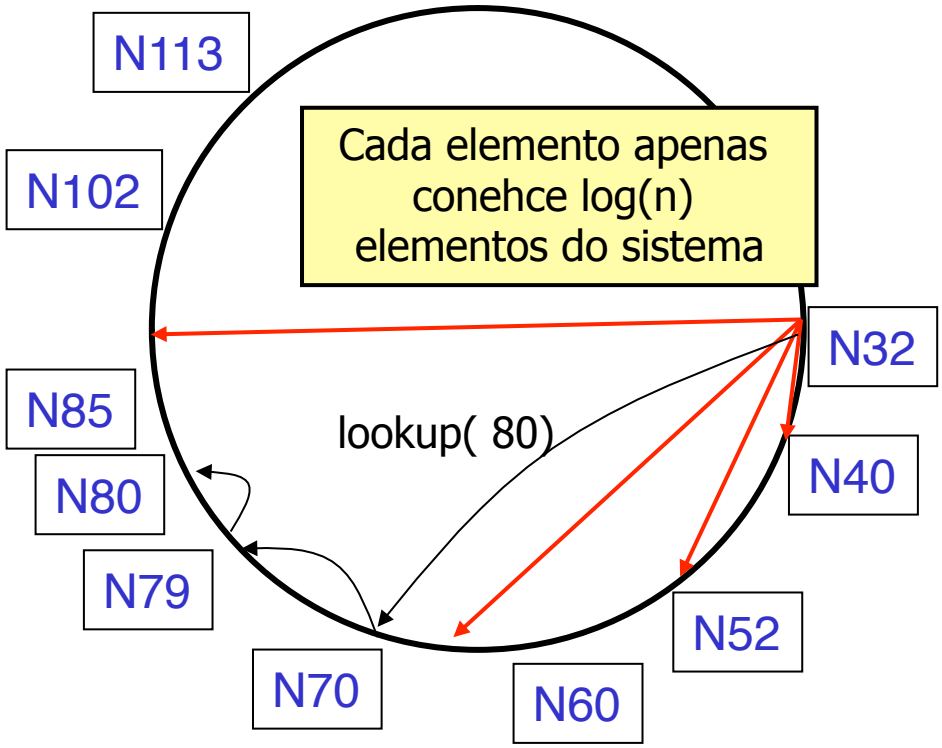
Variantes do modelo *peer-to-peer*



- Sistema peer-to-peer não estruturado
  - As ligações entre os membros são formadas de forma não-determinista
    - E.g. quando se junta à rede, um membro escolhe um conjunto de contactos (os contactos podem variar durante a execução do sistema)
  - Positivo:
    - Simplicidade
  - Negativo:
    - Pesquisa pesada (geralmente por inundação)
    - Latência/escalabilidade depende da árvore formada

Variantes do modelo *peer-to-peer*

- Sistemas *peer-to-peer* estruturados
  - Os membros do sistema comunicam de acordo com uma organização definida de forma determinista
    - E.g. DHT permitem pesquisar valores conhecidos
      - Nó responsável por uma dada chave depende do identificador do nó
  - Positivo
    - Boa latência/escalabilidade - O (log n)
  - Negativo
    - Maior complexidade



DHTs: características interessantes

- Identificar informação usando hash seguro – hash(info)
  - Quais as implicações?

- Cada nó fica responsável por tratar um conjunto de identificadores (de forma determinista)
  - E.g. cada nó tem associado identificador único gerado aleatoriamente, ficando responsável por manter informação cujo identificador é mais próximo do seu identificador
  - Implicações? Pesquisa? Distribuição da informação? Replicação da informação?

- Usar identificadores para criar um sistema de encaminhamento overlay, em que qualquer percurso tem, no máximo, log(n) nós

- Combinando os dois modelos pode-se ter o melhor dos dois

Combinação dos modelos c/s e p2p

- Cliente + peer-to-peer
  - Num sistema peer-to-peer, podem existir elementos que disponibilizam o serviço a outros processos (clientes) que não pertencem ao sistema peer-to-peer
- Propriedades:
  - Permite a um host aceder a um serviço disponibilizado por um sistema peer-to-peer
  - Permite limitar o número de processos que fazem parte do sistema peer-to-peer

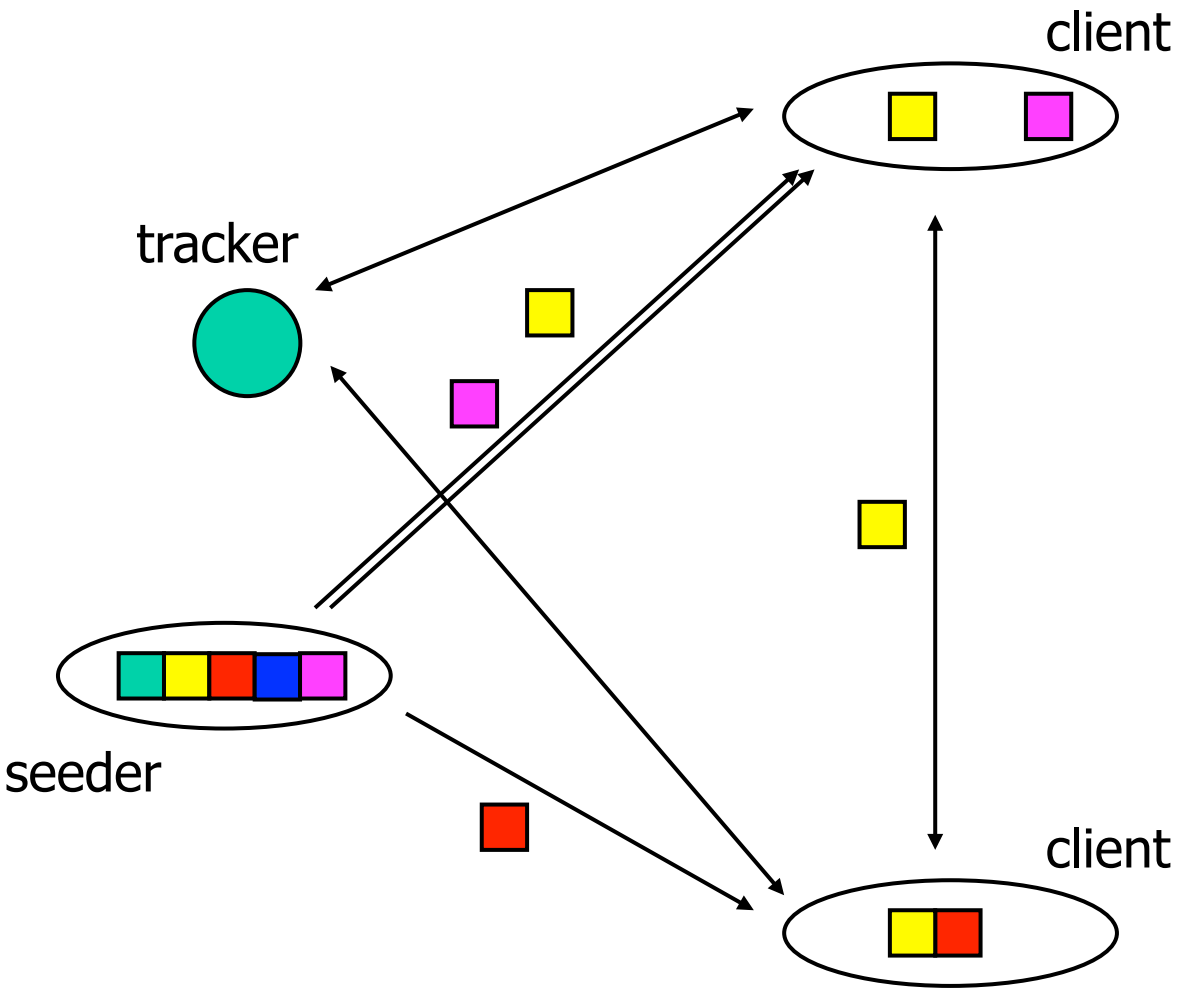
Combinação dos modelos c/s e p2p

- Cliente/servidor + peer-to-peer
  - O serviço disponibilizado por um sistema pode ser dividido em várias funcionalidades, sendo umas fornecidas por um sistema cliente/servidor e outras por um sistema peer-to-peer.
    Neste caso é comum o sistema cliente/servidor servir como serviço de directório.
    - E.g. BitTorrent
- Propriedades:
  - Permite combinar as vantagens de ambos os sistemas

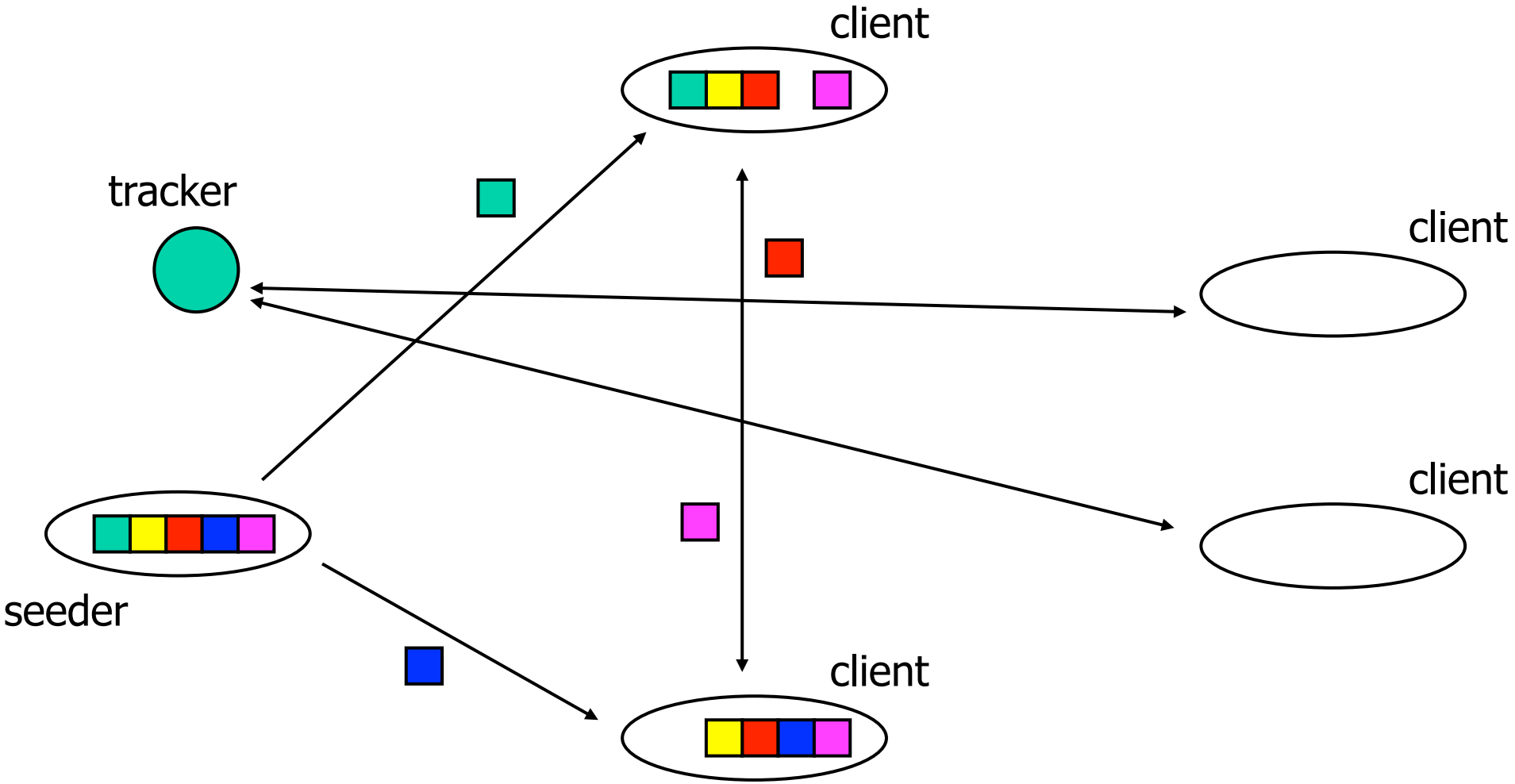
BitTorrent (resumido e simplificado)

- .torrent: contém informação sobre ficheiro a ser partilhado, incluindo: url do tracker, hash seguro de cada bloco do ficheiro (SHA-1)
- Arquitectura:
  - Cliente/servidor
    - Tracker: servidor centralizado mantém informação sobre quais os peers que estão a partilhar/descarregar o ficheiro
    - Peers: acedem ao tracker para obter lista de outros peers a descarregar o ficheiros
  - Peer-to-peer
    - Peers comunicam entre si para trocar blocos do ficheiro
      - Toma-lá dá-cá (tit-for-tat): peer só envia bloco em troca de outro bloco
      - Peer envia alguns blocos a outros peers (sem contrapartida - aleatoriamente)
      - Peer descarrega blocos aleatoriamente
      - Para que serve cada uma destas propriedades?

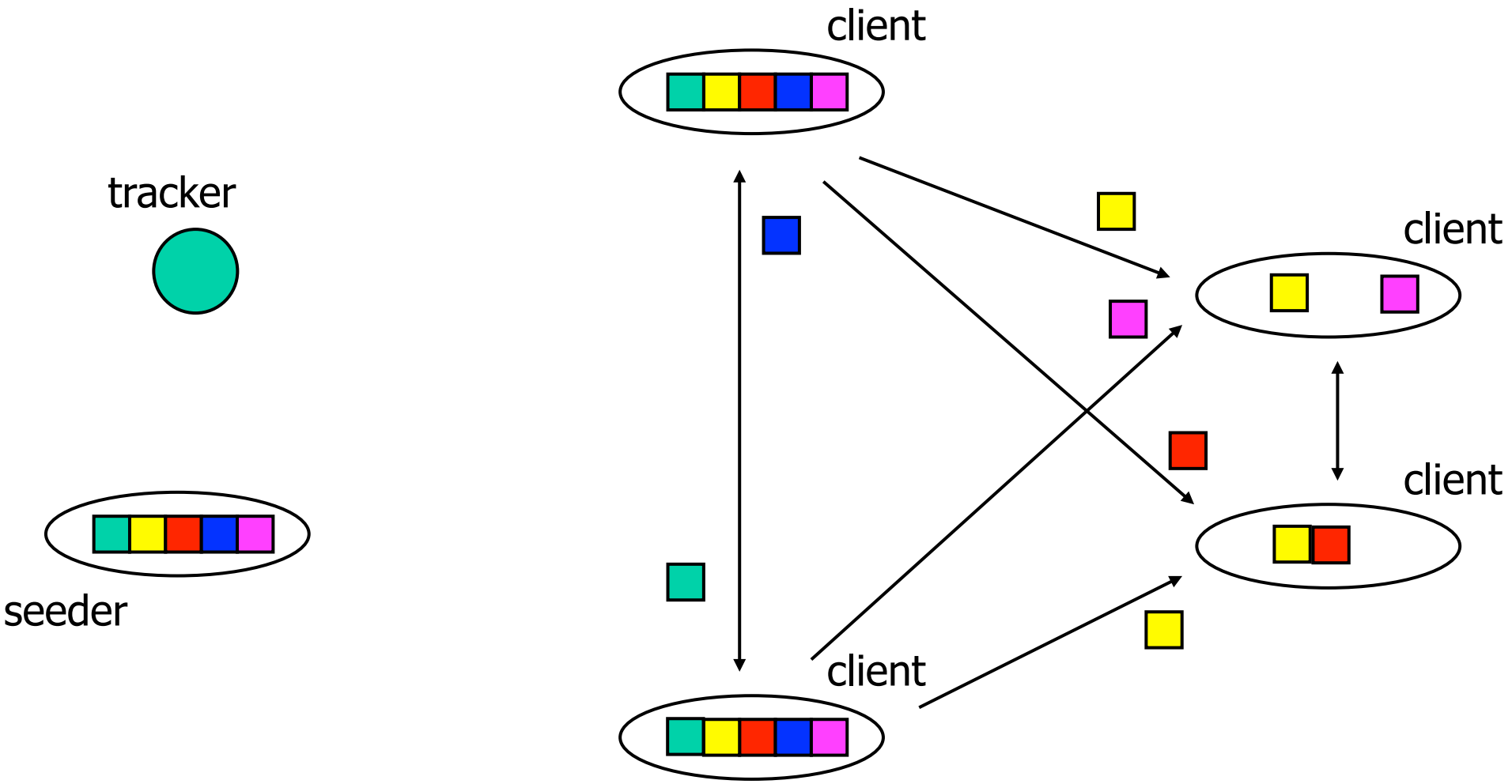
BitTorrent



BitTorrent



BitTorrent



BitTorrent DHT

- BitTorrent tem a possibilidade de funcionar sem trackers
- Neste caso, os peers formam uma DHT
  - Cada nó gera um identificador único
  - Cada *torrent* tem um identificador único
  - Informação sobre quais os nós que estão a partilhar/descarregar um ficheiro armazenada nos nós com identificadores mais próximos
  - Nota: baseado no sistema Kademlia (2002)

eDonkey/eMule

- Arquitectura:
  - Cliente/servidor
    - Servidor: mantém informação sobre os ficheiros que cada peer tem
    - Peers: acedem ao servidor para fazer: (1) pesquisas; (2) obter informação sobre quais os peers que partilham cada ficheiro
  - Peer-to-peer
    - Peers comunicam entre si para trocar blocos do ficheiro
      - Cada peer tem uma lista de outros peers que lhe fizeram pedidos. Esta lista é ordenada em função: do tempo de espera, dos créditos (função dos uploads recebidos)
      - Ordem dos pedidos tenta que aumente o número de fontes
    - Servidor podia ser envolvido na comunicação peer-to-peer para contornar clientes bloqueados por *firewalls*

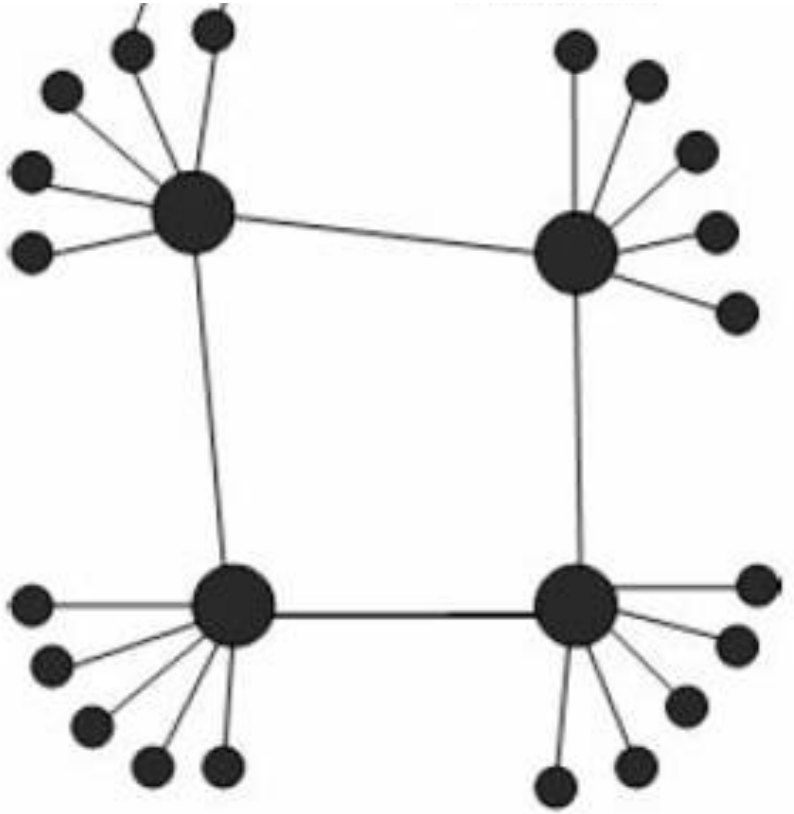
Ainda outros exemplos...

- Sistemas peer-to-peer hierárquicos
  - Subconjunto de super-nós que se agrupam como num sistema peer-to-peer
  - Nós ligam-se a um super-nó

...

KaZaA

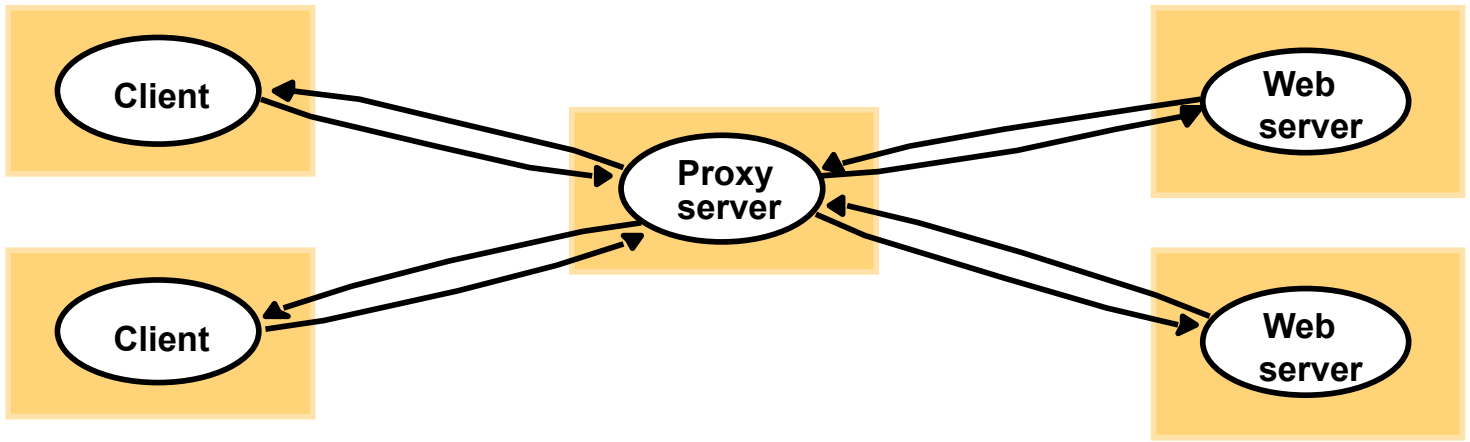
- Arquitectura:
  - Super-nós (SN): nós com maior capacidade tornam-se super-nós
    - Cada SN tem 60-150 nós ligados
    - Cada SN liga-se a outros 30-50 SN
  - Peers (ordinary node – ON)
    - ON liga-se a um dos SNs que conhece (após se ter ligado, actualiza lista de SNs)
- Pesquisa:
  - ON contacta o seu SN
  - SNs propagam pedidos entre si
- Transferência de ficheiros:
  - Directamente entre ONs



- Nota: Skype tem aproximação semelhante

- Na prática, tendem a existir mais componentes num sistemas distribuído

Noção de proxy de um serviço



- Proxy de um serviço
  - Processo que fornece um serviço recorrendo a um servidor (desse serviço) para executar o serviço
- Utilizações possíveis
  - Intermediário simples
  - Intermediário complexo (*gateway*)
    - Transformação dos pedidos
  - Serviço adicional através do *caching* das respostas
    - Diminuição do tempo de resposta (latência inferior para o proxy)
    - Diminuição da carga do servidor
    - Mascarar falhas do servidor / desconexão

Edge-server

- Sistemas edge-server
  - Existem servidores colocados nos ISP para responder a pedidos
    - E.g. akamai (páginas web), content-distribution networks
  - Propriedades
    - Menor latência, filtragem, distribuição de carga, etc.

Arquitecturas orientadas a serviços – service-oriented architectures (SOA)

- Arquitectura em que os recursos são disponibilizados como serviços independentes que podem ser acedidos conhecendo apenas a interface e protocolo de acesso. Pressupõe-se que:
  - Existe mecanismo de descoberta de serviços
  - Serviços tendem a ser *stateless*
  - Pode-se implementar usando qualquer tecnologia: Web Services, RPCs, RMI, Jini, etc.
    - Surge normalmente associado com Web Services (permitem a invocação remota usando protocolo sobre HTTP)

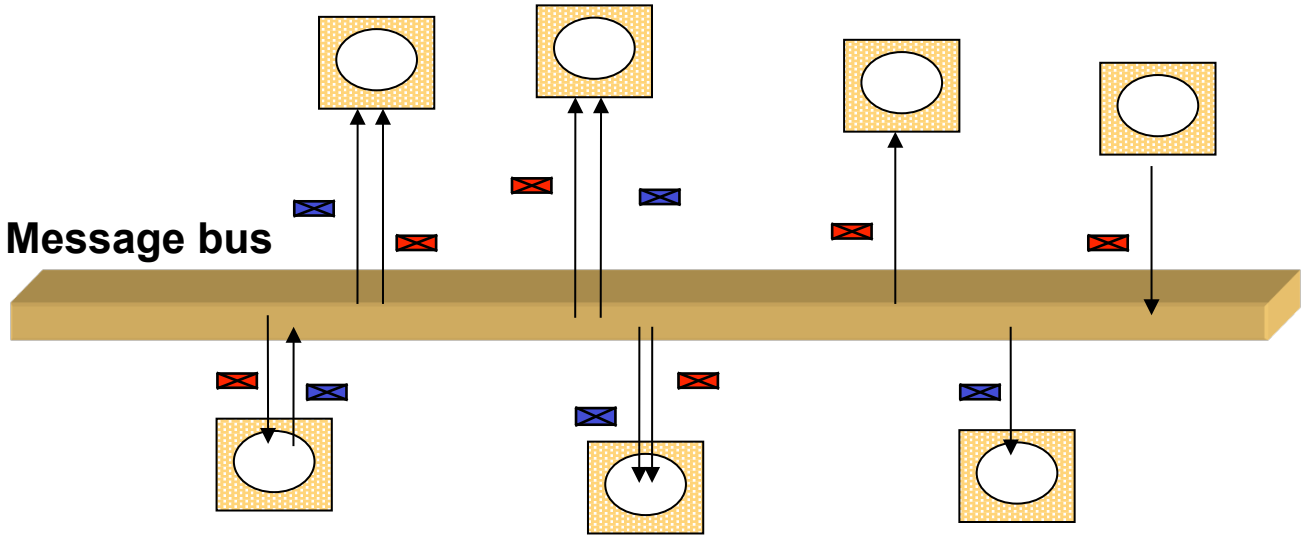
- Propriedades desejáveis:
  - Reutilização, modularidade, possível compor serviços mais complexos, descrição dos serviços

- Desafios:
  - Segurança, interoperablidade, estado nos serviços

Modelo de interacção (1)

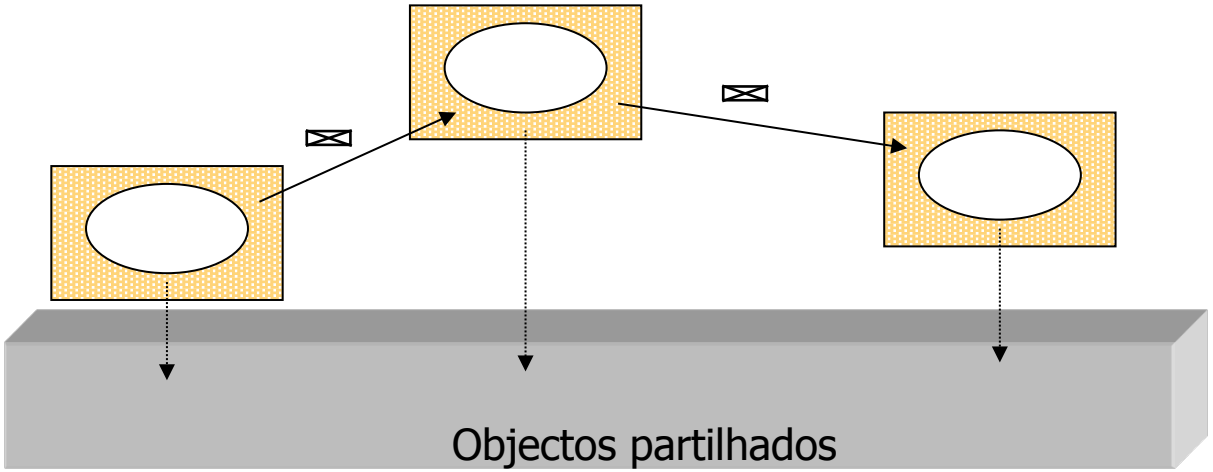
- Tipo de interacção
  - Activa
    - Processo solicita execução de operação noutro processo
  - Reactiva
    - Evento no sistema desencadeia acção num processo
  - Indirecta
    - Processos comunicam através de um espaço partilhado

Modelo reactivo (publish/subscribe – event-based system)



- No modelo "publish/subscribe", um processo (subscritor/consumidor) subscreve o interesse num conjunto de eventos junto de outro processo *(publisher*/produtor*)* que os fornece
- O *publisher* envia os eventos de um dado tipo para todos os processos que o subscreveram
- Um exemplo particular: *invocation return calback*

Modelo de interacção indirecta (ex.: memória partilhada distribuída)



- Processos comunicam e coordenam-se através dum espaço de "memória partilhada distribuída"
  - Processos escrevem e lêem dados no espaço de memória partilhada
  - Exemplos:
    - Sem suporte de hardware, memória partilhada implementada como serviço distribuído
    - Espaço de tuplos – operações de sincronização (leitura/escrita bloqueantes) – e.g. JavaSpaces
    - Base de dados

Modelo de interacção (2)

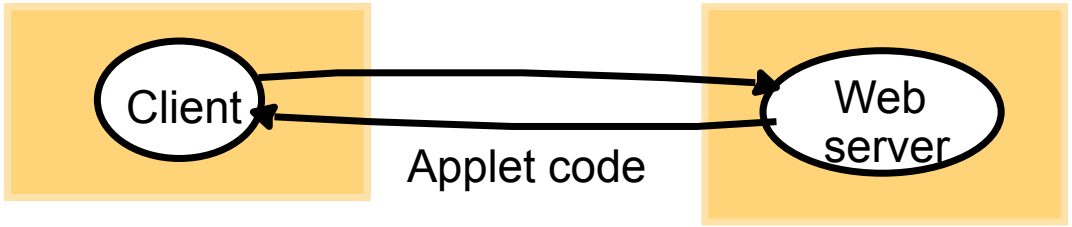
- Tipo de interacção
  - Activa
    - Processo solicita execução de operação noutro processo
  - Reactiva
    - Evento no sistema desencadeia acção num processo
  - Indirecta
    - Processos comunicam através de um espaço partilhado
- Conteúdo da interacção
  - Informação/dados
  - Código

Conteúdo da interacção: dados

- Processos trocam dados
  - Pedido (operação a invocar + parâmetros + inf. utilizador + ...)
  - Resposta (resultado de operação + ...)
  - Evento (num sistema de eventos)
- Propriedades
  - Parceiros devem conhecer formato das mensagens
  - Parceiros devem saber processar mensagens (operações)

Conteúdo da interacção: código móvel – cliente/servidor

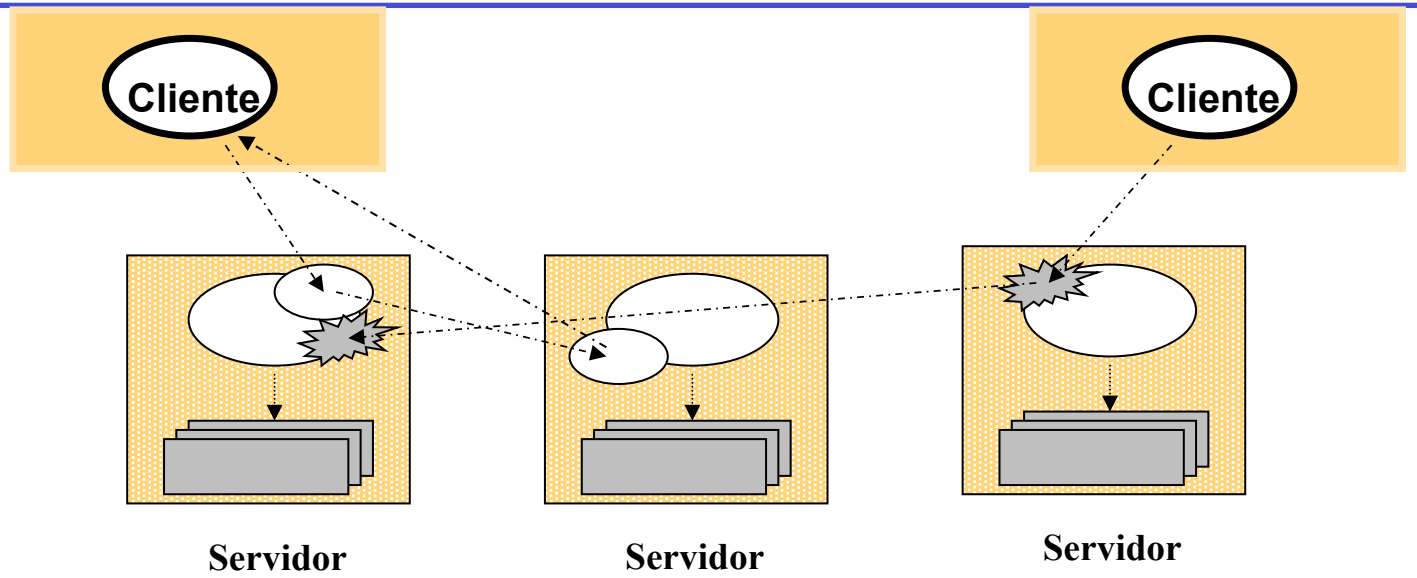
a) client request results in the downloading of applet code



b) client interacts with the applet

- A execução do código no cliente pode:
  - Melhorar o desempenho
  - Ser usado para implementar funcionalidade adicional
- Segurança
  - Necessário proteger o cliente do código executado

Conteúdo da interacção: código móvel – agentes



- Ideia: agentes navegam entre servidores, executando cada parte no servidor em que é mais apropriado
  - Exemplo: ?
- Problemas
  - Proteger informação do agente do ambiente de execução (impossível?)
  - Desenhar sistema de forma que recursos usados sejam menores do que outra arquitectura

Modelo de falhas

- Num sistema distribuído, tanto os processos (e computadores) como os canais de comunicação podem falhar
  - Não é possível conceber componentes sem falhas, apenas se pode diminuir a probabilidade de as mesmas ocorrerem
- O modelo de falhas consiste na definição rigorosa de quais os erros ou avarias, assim como das falhas que podem ter lugar nas diferentes componentes. O modelo de falhas abrange ainda a indicação rigorosa do comportamento global do sistema na presença dos diferentes tipos de falhas.

Algumas definições

- Fault tolerance - tolerância a falhas. Propriedade de um sistema distribuído que lhe permite recuperar da existência de falhas sem introduzir comportamentos incorrectos. Um sistema deste tipo pode mascarar as falhas e continuar a operar, ou parar e voltar a operar mais tarde, de forma coerente, após reparação da falha.

Algumas definições

- Availability – disponibilidade. Mede a fracção de tempo em que um serviço está a operar correctamente, isto é, de acordo com a sua especificação.
  - Para um sistema ser altamente disponível (highly available) deve combinar um reduzido número de falhas com um curto período de recuperação das falhas (durante o qual não está disponível).
- Reliability - fiabilidade. Mede o tempo desde um instante inicial até à primeira falha, i.e., o tempo que um sistema funciona correctamente sem falhas.
  - Um sistema que falha com grande frequência e recupere rapidamente tem baixa fiabilidade, mas alta disponibilidade.

Classificação dos sistemas

Classe	Disponibilidade	Indisponibilidade (por ano – máximo)	Exemplos
1	90,0%	36d 12h	Personal clients, experimental systems
2	99,0%	87h 36min	entry-level business systems
3	99,9%	8h 46min	top Internet Service Providers, mainstream business systems
4	99,99%	52min 33s	high-end business systems, data centers
5	99,999% (alta disponibilidade)	5min 15s	carrier-grade telephony; health systems; banking
6	99,9999%	31,5 seg	military defense systems

Mais definições

- *Timeliness* - adequação temporal ou pontualidade. Em sistemas de tempo real é a garantia de que o sistema é capaz de obedecer a constrangimentos temporais, isto é, a capacidade que o sistema tem de garantir limites para o tempo que as diferentes acções levam a executar.

Tipos de falhas dos componentes

- Uma falha por omissão dá-se quando um processo ou um canal de comunicação falha a execução de uma acção que devia executar
  - Exemplo: uma mensagem que devia chegar não chegou, processo falha (crash)
- Uma falha temporal dá-se quando um evento que se devia produzir num determinado período de tempo ocorreu mais tarde
- Uma falha arbitrária ou bizantina dá-se quando se produziu algo não previsto
  - Exemplo: chegou uma mensagem corrompida, um atacante produziu uma mensagem não esperada.
  - Para lidar com estas falhas é necessário garantir que elas não levam a que outros componentes passem a estados incorrectos

Mascarar falhas de componentes

- Para compensar os problemas levantados pelas falhas usam-se técnicas para as mascarar. Desta forma é possível confinar os seus efeitos sobre o sistema.
- As falhas de omissão podem ser mascaradas por

  replicação ou repetição
  - Exemplo: se uma mensagem não chegou dentro de um certo período – o que se detecta por um timeout – então pode-se emiti-la novamente. Outra hipótese é duplicar o canal, enviar mais do que uma cópia em paralelo e filtrar as mensagens duplicadas.
- As falhas arbitrárias podem ser difíceis de mascarar. Pode-se tentar transformá-las em falhas por omissão
  - Exemplo: um CRC numa mensagem permite transformar uma falha bizantina do canal numa falha por omissão
- O mesmo tipo de técnicas usam-se muitas vezes nos componentes hardware do tipo discos, memórias, etc... mesmo nos sistemas centralizados.

Falhas temporais

- As falhas temporais são difíceis ou impossíveis de mascarar.
- Normalmente apenas os sistemas de tempo real se preocupam com este tipo de falha.

Classe de falhas	Afecta	Descrição
Relógio	Processo	O relógio do processo tem um desvio superior ao permitido em relação ao tempo real
Desempenho	Processo	O processo leva mais tempo do que o estipulado a executar uma operação
Desempenho	Canal	A mensagem levou mais tempo do que o previsto

Tipos de erro/falha: duração

- Permanentes: mantêm-se enquanto não forem reparadas (ex: computador avaria)
  - Mais fáceis de detectar
  - Mais difíceis de reparar
- Temporárias: ocorrem durante um intervalo de tempo limitado, geralmente por influência externa
  - Mais difíceis de reproduzir, detectar
  - Mais fáceis de reparar

  - Erros transientes: ocorrem instantaneamente, ficam reparados imediatamente após terem ocorrido (ex.: perda de mensagem)

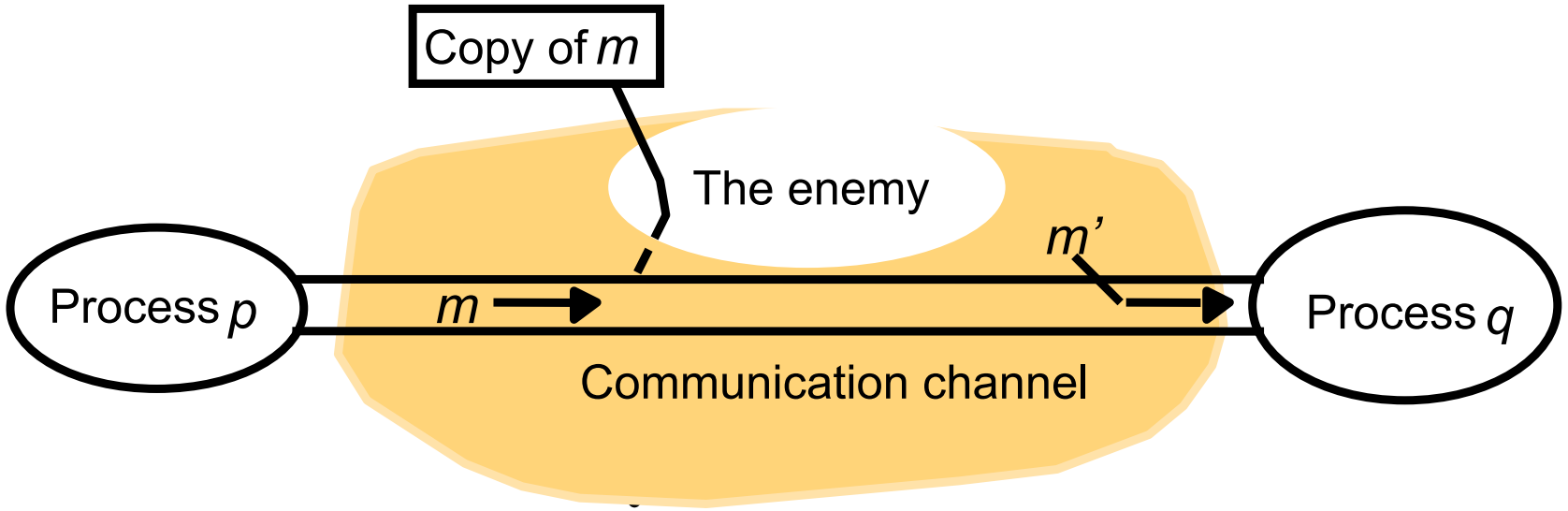
Segurança num sistema distribuído (modelo de segurança)

- A informação gerida por um sistema distribuído tem valor para os utilizadores
- O modelo de segurança consiste em definir quais as ameaças das quais um sistema se consegue defender
- A segurança de um sistema distribuído pode ser obtida:
  - Protegendo os processos e os canais usados para a interacção entre processos;
  - Protegendo os objectos (dados) geridos pelos processos contra acessos não autorizados.

Definição

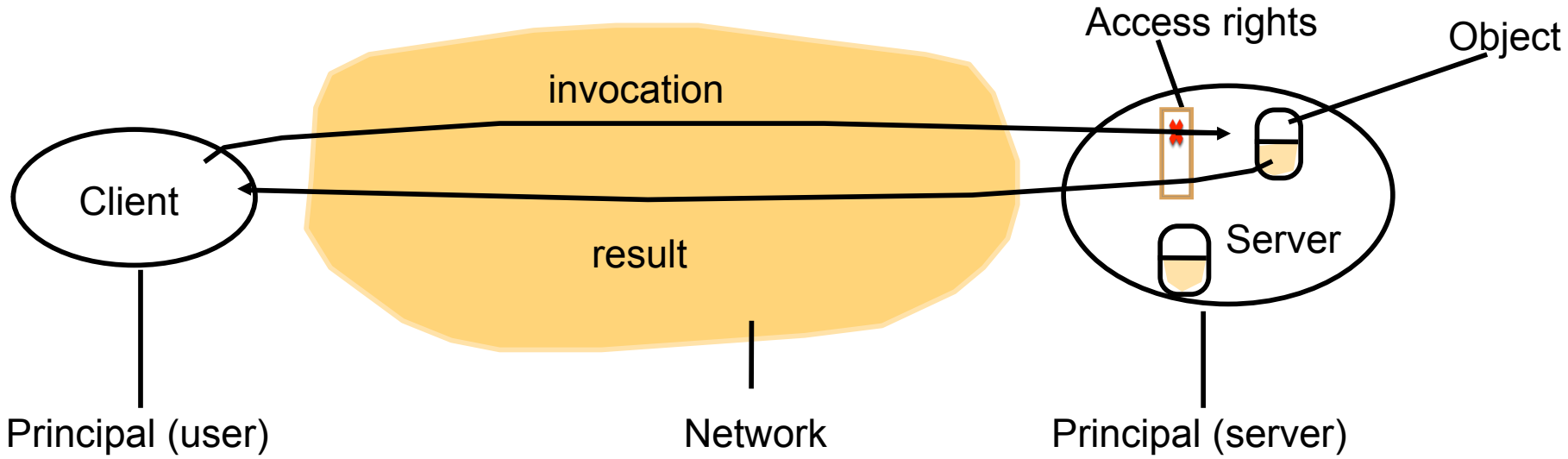
- Num sistema existem entidades que do ponto de vista da segurança têm identidade própria, direitos e deveres – essas entidades podem ser utilizadores, processos, etc. Utiliza-se o termo *principal* para as designar.
  - Em Inglês, no contexto legal, um principal é alguém que está a ser julgado pelos seus actos.

Ameaças básicas aos canais de comunicação (e consequentemente aos processos)



- Para modelar ameaças de segurança, assume-se que o inimigo tem grande capacidade e pode:
  - Enviar mensagens para qualquer processo
    - Forjar endereço das mensagens, fazendo-se passar por outro processo
  - Ler, copiar, remover e reenviar mensagens que passam no canal
    - Impedir interacção, repetir interacção, etc.

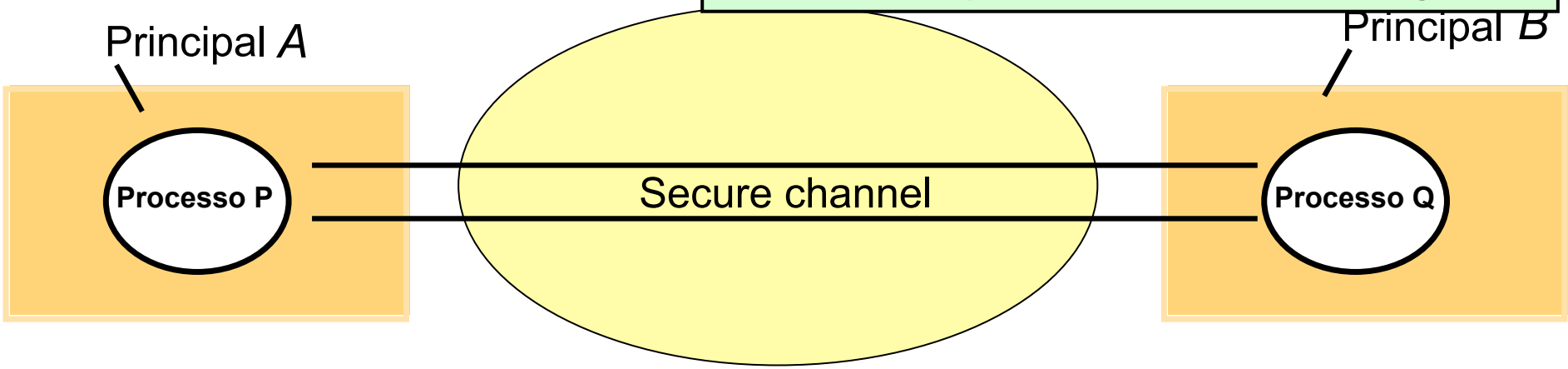
Principais, objectos, controlo de acesso e canais



- A segurança num sistema distribuído passa por:
  - Autenticar os principais
  - Verificar direitos de acesso aos objectos – controlo de acessos
  - Utilizar canais seguros

Canais seguros

Existem técnicas criptográficas (com partilha de segredos) que podem ser usadas na autenticação de parceiros e cifra de informação:
Permitem implementar um canal seguro



- Num canal seguro os interlocutores (A e B) estão autenticados
- O inimigo não pode copiar, alterar ou introduzir mensagens
- O inimigo não pode fazer *replaying* de mensagens
- O inimigo não pode reordenar as mensagens

Outras ameaças

- *Denial of service*: ataque em que o inimigo interfere (impede) actividade dos utilizadores autorizados

- *Código móvel*: problemas de segurança para os processos que recebem e devem executar código móvel

- Outras???

Para saber mais

- G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems – Concepts and Design, Addison-Wesley, 4th Edition, 2005

  Capítulo 2.

Sistemas Distribuídos I

Capítulo 3 - Comunicação em

Sistemas Distribuídos – Comunicação Ponto a Ponto

Parte 1

Nota prévia

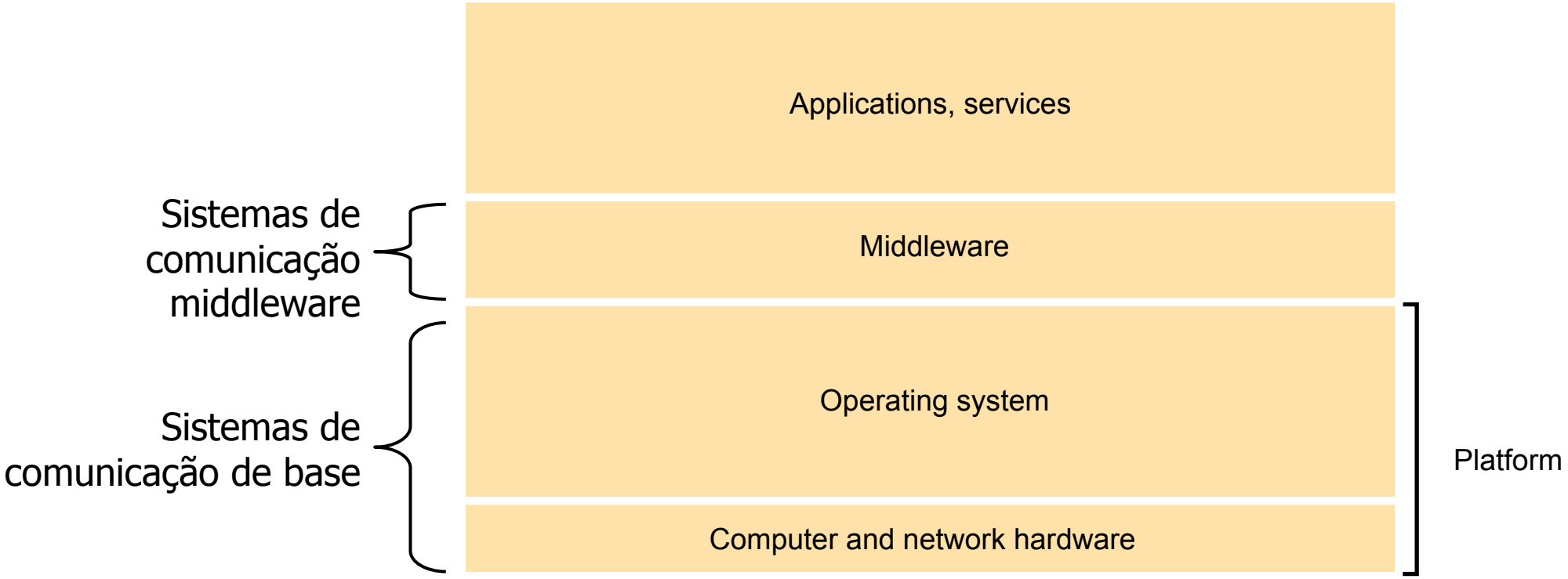
- A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

  G. Coulouris, J. Dollimore and T. Kindberg,
  Distributed Systems - Concepts and Design,
  Addison-Wesley, 4th Edition, 2005

Organização do capítulo

- Níveis dos sistemas de comunicação
- Formas de comunicação ao nível do middleware e sua caracterização
  - Comunicação cliente/servidor
- Heterogeneidade e representação dos dados

Comunicação num sistema distribuídos



Sistemas de comunicação de base

- Os sistemas de operação suportam a comunicação de dados entre os diferentes computadores envolvidos num sistema distribuído
  - Interfaces geralmente de baixo nível
  - Protocolo mais popular: TPC/IP
    - Oferece os seguintes protocolos de comunicação base:
    - **TCP – *streams***
      - Dados transmitidos como fluxo contínuo.
      - Dados chegam de forma fiável a menos que o stream seja quebrado.
    - **UDP – datagrama**
      - Comunicação por mensagens.
      - Mensagens podem-se perder, duplicar e chegar fora de ordem.
    - **IP multicast**
      - Comunicação por mensagens com múltiplos receptores.
      - Semântica semelhante ao UDP.

Comunicação no nível middleware

- Modelo de comunicação simples
  - Comunicação por streams (fluxos)
    - 1 ou N destinatários;
    - Uni-direccional ou bi-direccional.
  - Comunicação por mensagens.
    - 1 ou N destinatários;
    - Fiabilidade no envio de mensagens ou não;
    - Diferentes garantias de ordenação na propgação das mensagens.

- Comunicação baseada no paradigma pedido / resposta.
  - Invocação de métodos/procedimentos remotos (RMI / RPC)

- Comunicação baseada no paradigma do código móvel (“agentes”)

Facetas da comunicação ponto-a-ponto

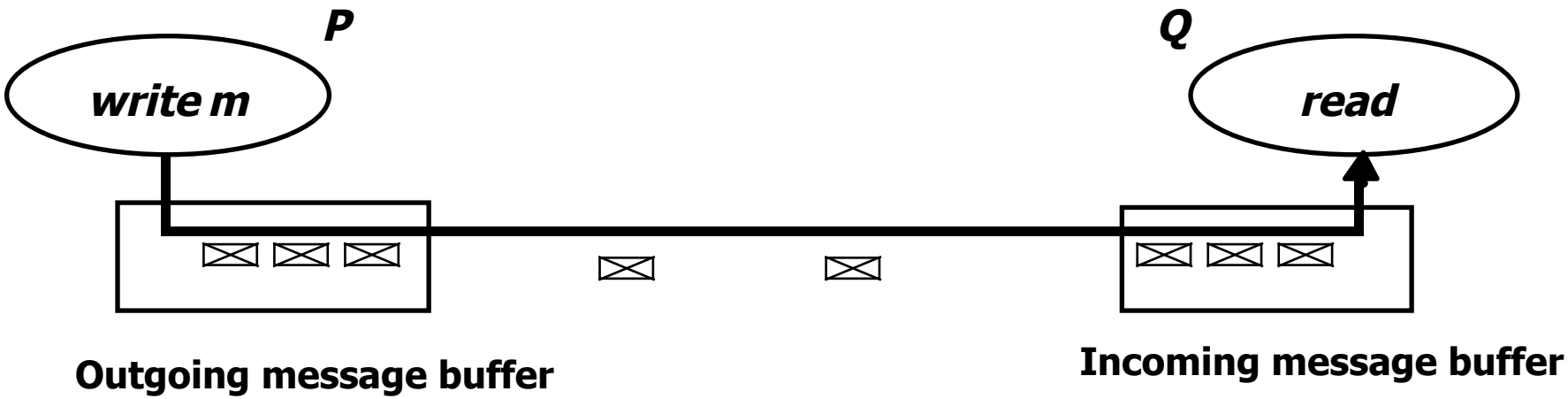
- Forma da interacção
  - Streams
  - Mensagens
    - Ordenação das mensagens
- Número de destinatários
  - Ponto-a-ponto
  - Multi-ponto (estudado mais tarde)
- Direcção de interacção
  - Uni-direccional
  - Bi-direccional
- Tipo de sincronização
  - Comunicação síncrona
  - Comunicação assíncrona
- Persistência
  - Comunicação persistente
  - Comunicação volátil
- Fiabilidade (modelo de falhas)

Definição

- Entrega de um mensagem num sistema de comunicação representa a accção do sistema disponibilizar a mensagem para ser lida pelas aplicações
  - NOTA: internamente, o sistema de comunicação pode receber mensagens e nãs as disponibilizar imediatamente às aplicações. Possíveis razões?

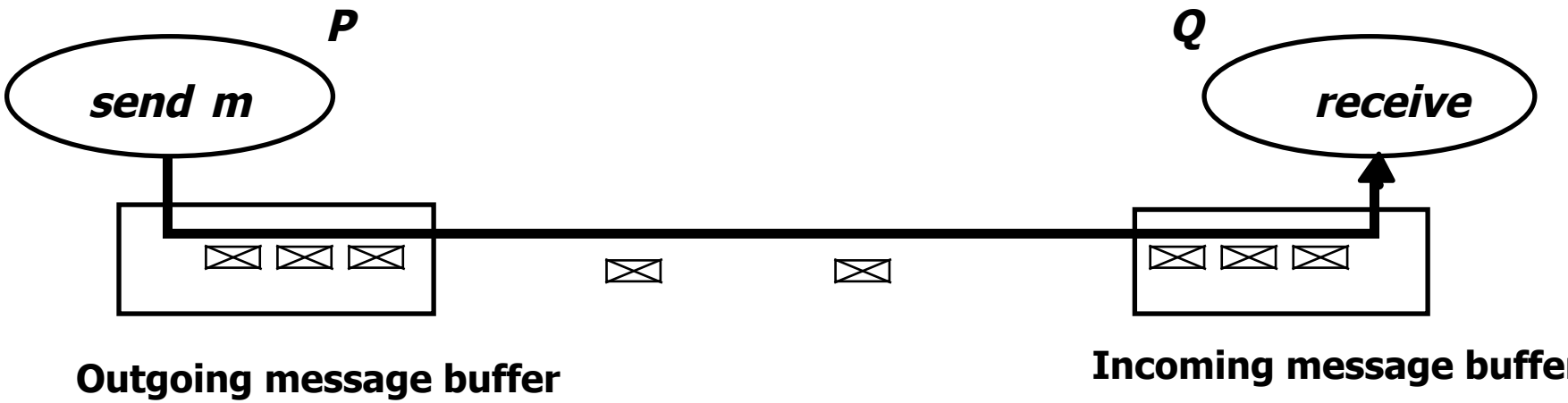
Streams

- Emissor e receptor estabelecem um fluxo contínuo de dados
  - Ordem dos dados enviados é mantida;
  - Dimensão (fronteira) dos dados/mensagens não é mantida.
- Algumas variantes
  - Um stream pode ser composto por vários *substreams* (e.g. vídeo+audio)
    - Problemas de sincronização dos *substreams*
  - Qualidade de serviço
    - Condições temporais para propagação entre extremos (síncrono)
    - Reserva de largura de banda (ex: RSVP)



Comunicação por mensagens

- Emissor e receptor comunicam trocando mensagens
  - Cada mensagem tem um limite (e dimensão) bem-definida.



Ordenação das mensagens (comunicação por mensagens ponto-a-ponto)

- Sem garantias de ordem
  - Sistema não garante que as mensagens são entregues pela ordem que foram enviadas

- Entrega pela mesma ordem da emissão – FIFO (first in first out)
  - Sistema garante que as mensagens dum emissor são entregues pela mesma ordem que foram enviadas. Como implementar em TCP/UDP?

Número de destinatários

- Comunicação ponto-a-ponto
  - Comunicação entre um emissor e um receptor
- Comunicação multi-ponto
  - Comunicação entre um emissor e um conjunto de receptores
  - (detalhado na segunda parte deste capítulo)

Direcção de interacção

- Comunicação uni-direccional:
  - Comunicação apenas num sentido: emissor->receptor
- Comunicação bi-direccional:
  - Comunicação nos dois sentidos

Sincronização

- Comunicação assíncrona:
  - o emissor só fica bloqueado até o seu pedido de envio ser tomado em consideração
  - o receptor fica bloqueado até ser possível receber dados
    - Em geral, o sistema de comunicação do receptor armazena (algumas) mensagens caso não exista nenhum receptor bloqueado no momento da sua recepção. Assim, funciona como um *buffer* entre o emissor e o receptor
    - É possível variante em que o receptor não fica bloqueado e devolve erro ou a recepção é efectuada em background

- Comunicação síncrona:
  - o emissor fica bloqueado até:
    - o receptor "receber" os dados – comunicação síncrona unidireccional
    - receber a resposta do receptor – comunicação pedido / resposta ou cliente / servidor
  - o receptor fica bloqueado até ser possível consumir dados

Persistência

- Comunicação volátil: mensagens apenas são encaminhadas se o receptor existir e estiver a executar, caso contrário são destruídas.
  - Exemplo: ???

- Comunicação persistente: mensagens são guardadas pelo sistema de comunicação até serem consumidas pelos destinatários, que podem não estar a executar. Mensagens são guardadas num receptáculo independente do receptor – mailbox, canal, porta persistente, etc.
  - Exemplo: ???

Fiabilidade

- Comunicação fiável: o sistema garante a entrega das mensagens em caso de falha temporária. Como implementar?
- Comunicação não-fiável: em caso de falha, as mensagens podem-se perder
  - Não se aplica a sistemas de comunicação baseados em *streams*

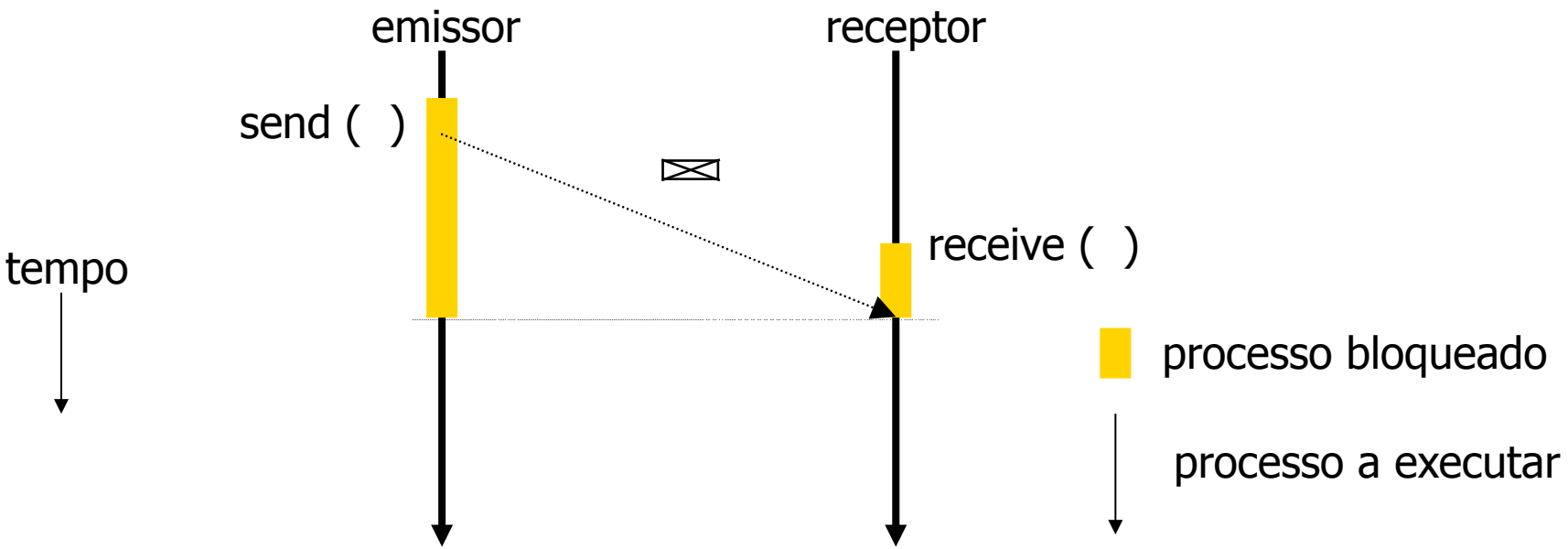
Sistemas de comunicação *middleware*

- Podem ter diferentes modelos paradigmas
  - Simples por mensagens
  - Simples por streams
  - Pedido/resposta
  - Baseado no paradigma de código móvel
- Fornecem diferentes propriedades relativas às facetas abordadas
  - Forma da interacção
  - Número de destinatários
  - Direcção de interacção
  - Tipo de sincronização
  - Persistência
  - Fiabilidade (modelo de falhas)

Sistemas de comuncação

- Alguns exemplos
  - Streams assíncronos (TCP)
  - Comunicação assíncrona por mensagens (UDP)
  - Comunicação síncrona unidireccional
  - Comunicação síncrona pedido / resposta
  - Comunicação persistente síncrona
  - Comunicação persistente assíncrona
  - Message-Oriented Middleware (MOM)

Comunicação síncrona unidireccional

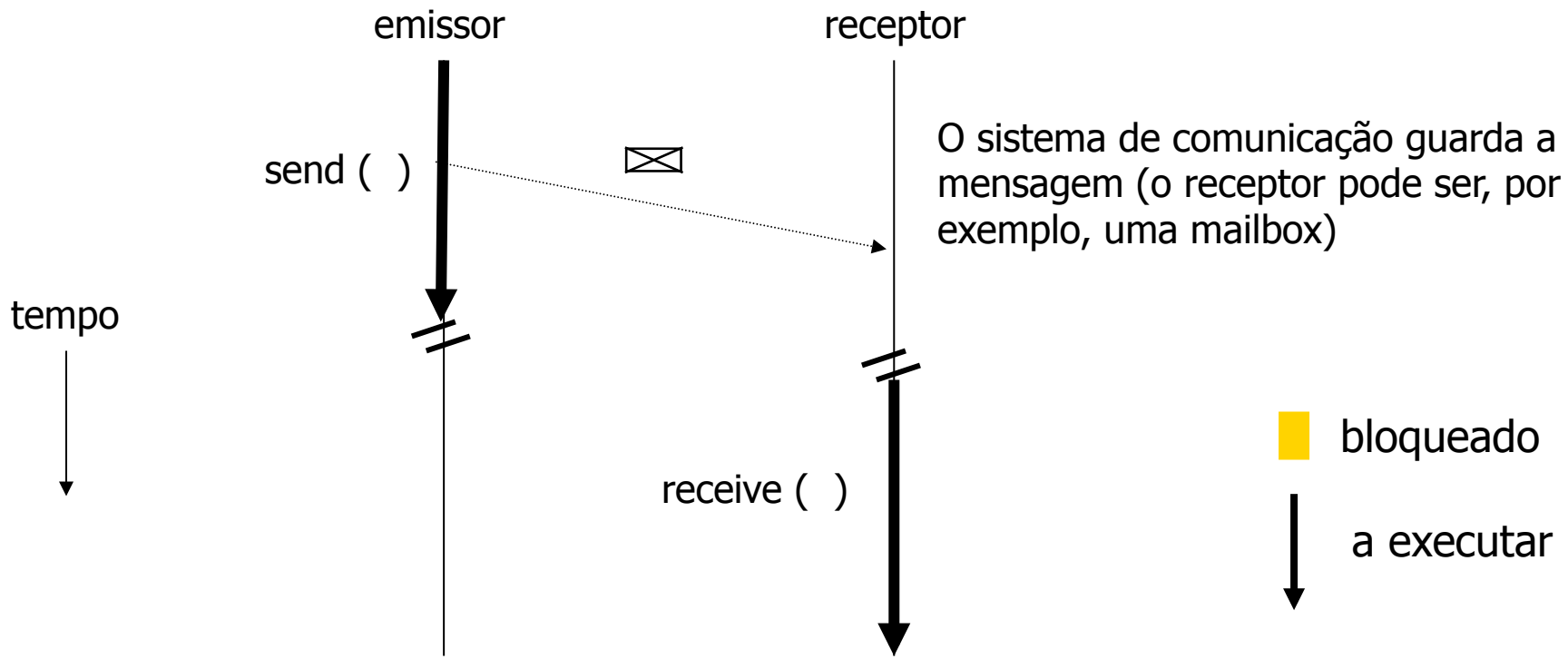


- O emissor fica bloqueado à espera de o receptor estar disposto a receber a mensagem
- Implementação exige envio de mensagem com informação de recepção para o emissor

Comunicação síncrona unidireccional

- Concorrência:
  - Limita a concorrência, porque o emissor se bloqueia até o receptor receber a mensagem enviada.
- Sincronização:
  - Após o envio o emissor sabe que o receptor acabou de receber a mensagem;
  - Após a recepção, o receptor sabe que o emissor esteve bloqueado até esse momento.
- Ordenação:
  - Em geral, garante a ordem das mensagens do mesmo emissor.
- Modelo de falhas:
  - Em caso de sucesso o emissor sabe que o receptor recebeu a mensagem.
  - Em caso de insucesso não se sabe exactamente o que se passou (problemas de rede ou do receptor ?).
- Variações: é possível o emissor só ficar bloqueado até a mensagem chegar ao site do receptor mesmo que este não a consuma logo.

Comunicação persistente assíncrona



Comunicação persistente assíncrona

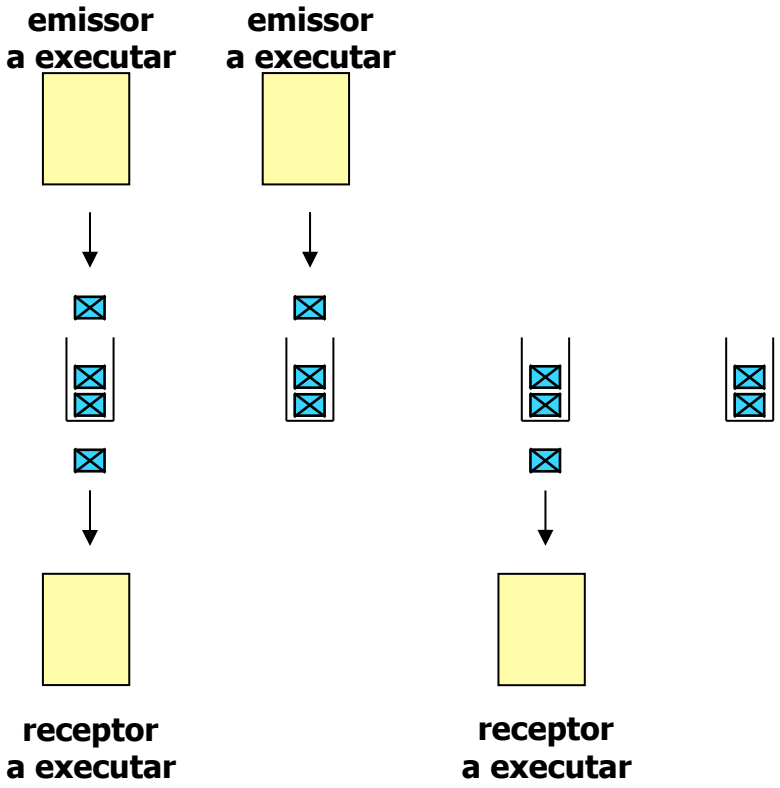
- Concorrência:
  - Emissor nunca se bloqueia. Receptor pode bloquear-se na recepção de mensagens.
- Sincronização:
  - Não existe sincronização entre emissor e receptor.
- Ordenação:
  - Geralmente garante a ordem das mensagens do mesmo emissor.
- Modelo de falhas:
  - Emissor não tem garantia que mensagem foi armazenada. Se for armazenada, em geral, o sistema garante que a mensagem é guardada até o receptor a consumir.
  - Não existem garantias que algum receptor a consuma.

Message-Oriented Middleware (MOM) ou Message-Queuing systems

- Os processos comunicam pela troca de mensagens usando um subsistema intermédio que assegura a persistência através de filas de mensagens (queues)
  - Comunicação assíncrona
  - Suporta transferência de mensagens que duram vários minutos

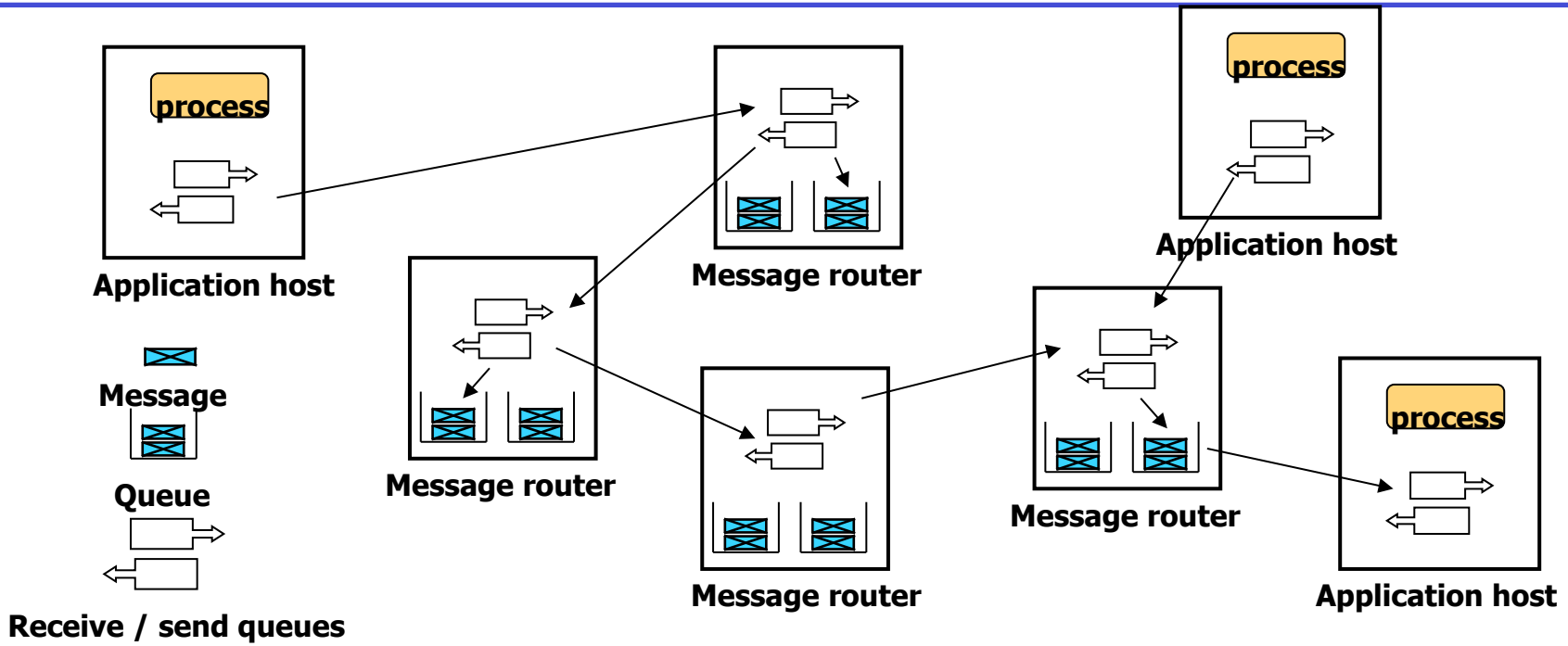
- Uma mensagem é sempre endereçada por um processo para uma queue e só pode ser consumida da queue por um outro processo / aplicação (uma queue pode, no entanto, ser partilhada por vários processos / aplicações)
  - O emissor apenas tem garantias que a mensagem foi entregue na queue

Interacção emissor / receptor



- Primitivas:
  - Put – adiciona uma mensagem a uma *queue*
  - Get – bloqueia até que a *queue* não esteja vazia e remove a primeira mensagem
  - Poll – verificar se existem mensagens na *queue*
  - Notify – Instalar um *handler* para ser notificado se uma mensagem chegar

Estrutura geral de um MOM



- Mensagens podem ser propagadas para o destino através de filas intermédias
  - Algumas máquinas podem ter filas e actuar apenas como *routers*
    - Permite implementar funcionalidade adicional – ex.: log para segurança ou tolerância a falhas, conversão de formato de mensagens/gateway (message brokers)
- Aplicações: e-mail, workflow, processamento em batch, systemas multidatabase (MQSeries)

Heterogeneidade na representação dos dados

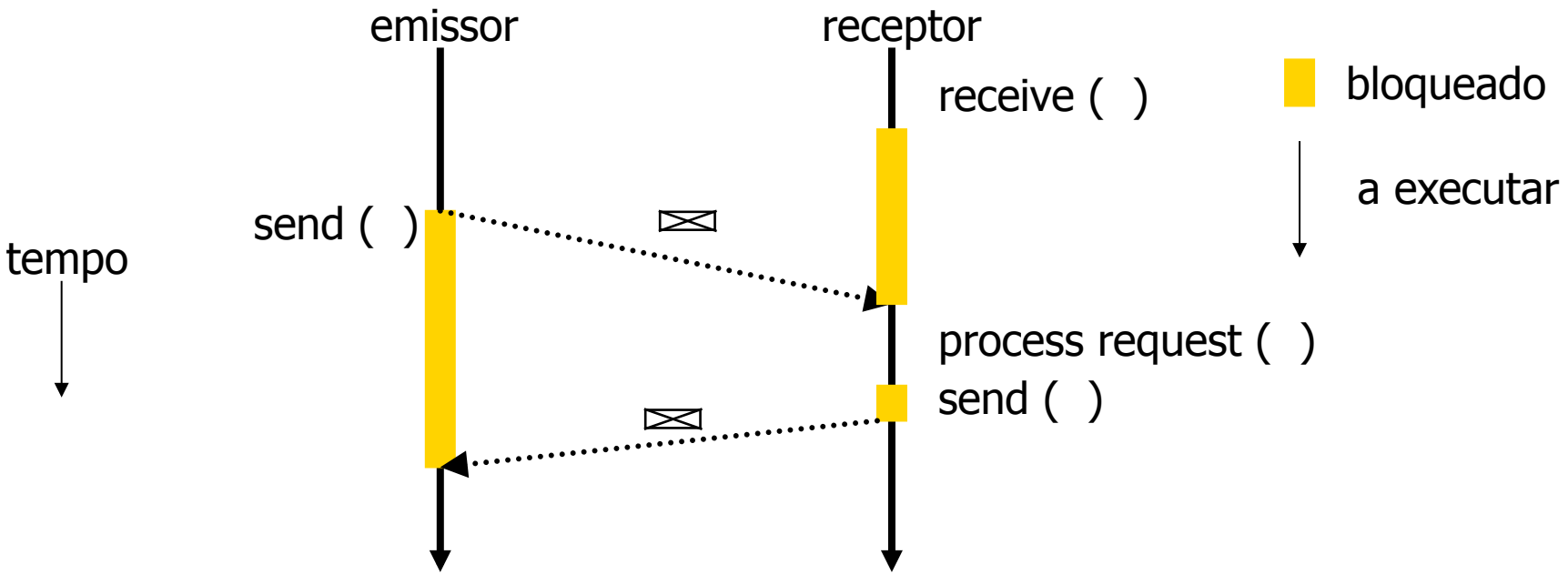
- Diferentes arquitecturas (processadores), sistemas e linguagens podem ter:
  - diferentes representações de números reais
  - diferente ordem na representação de inteiros (big-endian, little-endian)
  - diferentes representações de caracteres (ASCII, Unicode, EBCDIC)
  - diferentes definições do que é um inteiro (2, 4, 8 bytes?), real

- Problema: num sistema distribuídos, como garantir que dois processos a funcionar em diferentes arquitecturas, sistemas ou linguagens conseguem trocar dados?
  - Definir formato no qual os dados devem passar na rede
    - Formato do emissor, formato do receptor, formato independente
  - *Marshalling*: codificação do formato interno para o formato rede
  - *Unmarshalling*: descodificação do formato rede para o formato interno

Para saber mais

- G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems – Concepts and Design, Addison-Wesley, 4th Edition, 2005
  - capítulo 4 : partes sobre heterogeneidade e protocolos de invocação remota serão abordados no próximo capítulo

Comunicação síncrona pedido / resposta



- O emissor fica bloqueado à espera da resposta do receptor

Comunicação síncrona pedido / resposta

- É o modo de comunicação característico do modelo cliente / servidor.
- Concorrência:
  - Limita a concorrência porque o emissor bloqueia-se à espera da resposta do receptor.
- Sincronização:
  - O receptor sabe que a mensagem foi emitida, recebida e tratada.
  - O receptor sabe também que o emissor esperou por ele até receber a resposta.
- Ordenação:
  - Se não houver falhas, garante a ordem das mensagens do mesmo emissor.
- Modelo de falhas:
  - Em caso de sucesso o emissor sabe que o receptor recebeu a mensagem, tratou-a e respondeu. Em caso de insucesso não sabe exactamente o que se passou:
    - Problema de rede ou do receptor?
    - Problema antes ou depois de processar o pedido?

Representação de dados e *marshalling*

- Muitos dos protocolos de rede usam a codificação dos dados através de códigos ASCII (SMTP, FTP, …). Esta também é a opção quando se usa XML.
- Alguns sistemas fornecem suporte para codificar tipos complexos e (hierarquias de) objectos de forma automática. Exemplos: RPC, Corba, RMI.
  - Estas alternativas serão estudadas em detalhe no próximo capítulo

Sistemas Distribuídos

Capítulo 3 - Comunicação em

Sistemas Distribuídos – Comunicação em Grupo

Parte 2

Nota prévia

- A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

  G. Coulouris, J. Dollimore and T. Kindberg,
  Distributed Systems - Concepts and Design,
  Addison-Wesley, 4th Edition, 2005

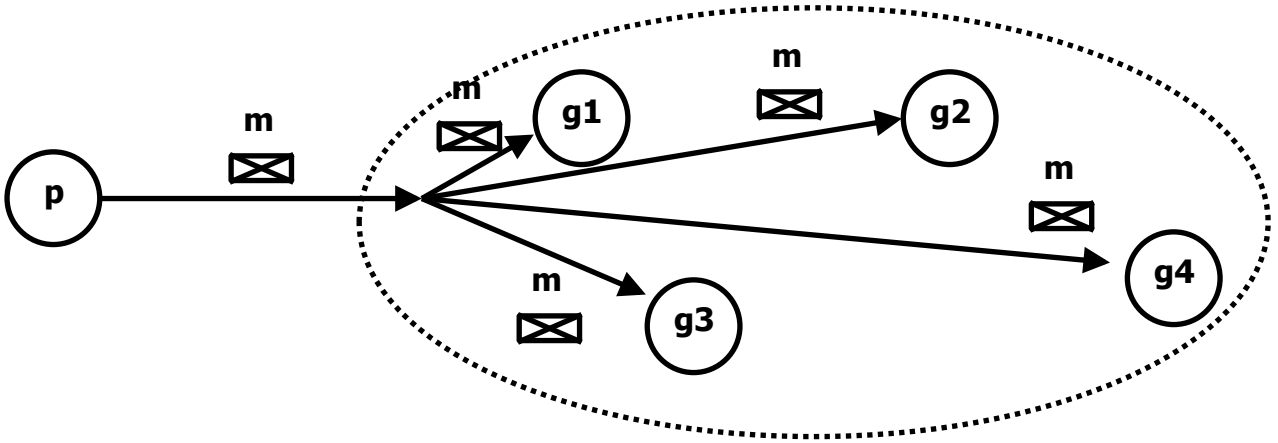
- Para saber mais:
  - secção 12.4 – apenas parte relativa aos tipos de multicast (não entrar na parte de implementação)

Organização do capítulo

- Comunicação em grupo

Comunicação em grupo

- Um grupo é um conjunto dinâmico de *communication end-points* (portas, processos, servidores, etc.) que é tratado pelo sistema como uma entidade única do ponto vista da comunicação
  - Operação *multicast*: envio de uma mensagem para todos os elementos do grupo, usualmente de forma que a filiação (*membership*) do grupo é transparente para o emissor



Algumas definições

- Comunicação ponto a ponto (ou *unicast*) [1x1] – envio de uma mensagem de um único emissor para um único receptor

- Formas de comunicação multi-ponto
  - Comunicação *multicast* ou em grupo [1xN] – envio de uma mensagem de um emissor para um grupo de receptores
    - Pode exigir que o emissor pertença ao grupo ou não
    - Forma de identificar o grupo
  - Comunicação broadcast ou por difusão [1xAll] – envio de uma mensagem de um emissor para todos as máquinas do sistema
  - Comunicação funcional ou anycasting [1x1 among N] – envio de uma mensagem de um emissor para um de um grupo de receptores

A noção de grupo aparece a vários níveis

- Nível físico
  - Ex.: ethernet broadcasting e multicasting
- Nível datalink e nível rede
  - Ex.: IP multicasting
- Sistema de operação [distribuídos]
  - Ex.: grupos de processos, grupos de portas
- Plataformas midleware:
  - Ex.: Isis, Horus e Ensemble (Univ. Cornell), Consul (Univ. Arizona), Totem (Univ. California at Santa Barbara), JGroups, Appia (FCUL)
- Plataformas middleware publish / subscribe e de disseminação de eventos
  - Ex. Java Message Service (JMS), iBUS, BEA MessageQ

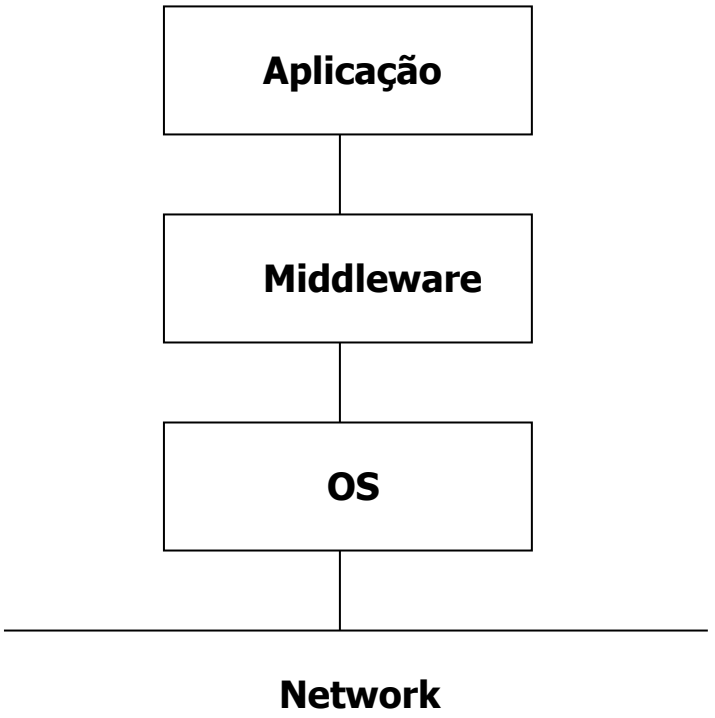
Características do *multicast*

- *Multicasting* pressupõe:
  - endereço / identificador único do grupo
  - composição dinâmica não necessariamente visível
  - a modificação da composição é feita de forma independente dos emissores
  - gestão, localização, comunicação, etc. com ou dos membros do grupo é assegurada pelo suporte de materialização da comunicação multi-ponto
- Interface
  - createGroup, destroyGroup, joinGroup, leaveGroup, send, receive
  - (disseminação de eventos) open, destroy, subscribe, unsubscribe, publish, receive (call-back)

Entrega das mensagens

- Diz-se que sistema de comunicação entrega uma mensagem quando a mensagem é entregue à aplicação

- O *middleware* pode reordenar as mensagens recebidas localmente atrasando a entrega das que já recebeu para garantir as propriedades desejadas
  - Para esse efeito mantém uma fila local de mensagens recebidas e ainda não entregues



Caracterização do multicast: fiabilidade

- Multicast não fiável (unreliable multicast) – em caso de falha, não existem garantias sobre a entrega das mensagem aos vários elementos do grupo
  - e.g. IP mutlicast
- Multicast fiável (reliable multicast) – uma mensagem enviada para um grupo ou é entregue por todos os membros correctos (que não falham) ou por nenhum
  - Membros que falham podem ter entregue a mensagem ou não
  - Implementação simples (e errada): o emissor emite para cada um dos elementos do grupo de forma fiável (acks+retransmissões)
    - **Porquê errada?**

Caracterização do multicast: fiabilidade

- Multicast não fiável (unreliable multicast) – em caso de falha, não existem garantias sobre a entrega das mensagem aos vários elementos do grupo
  - e.g. IP mutlicast
- Multicast fiável (reliable multicast) – uma mensagem enviada para um grupo ou é entregue por todos os membros correctos (que não falham) ou por nenhum
  - Membros que falham podem ter entregue a mensagem ou não
  - Implementação simples (e errada): o emissor emite para cada um dos elementos do grupo de forma fiável (acks+retransmissões)
    - Em caso de falha do emissor, é necessário que os elementos do grupo propaguem as mensagens recebidas para os elementos que ainda não as receberam
  - Uniform agreement: se algum processo que falha entrega a mensagem, todos os processos correctos devem entregar a mensagem

Caracterização do multicast: ordem

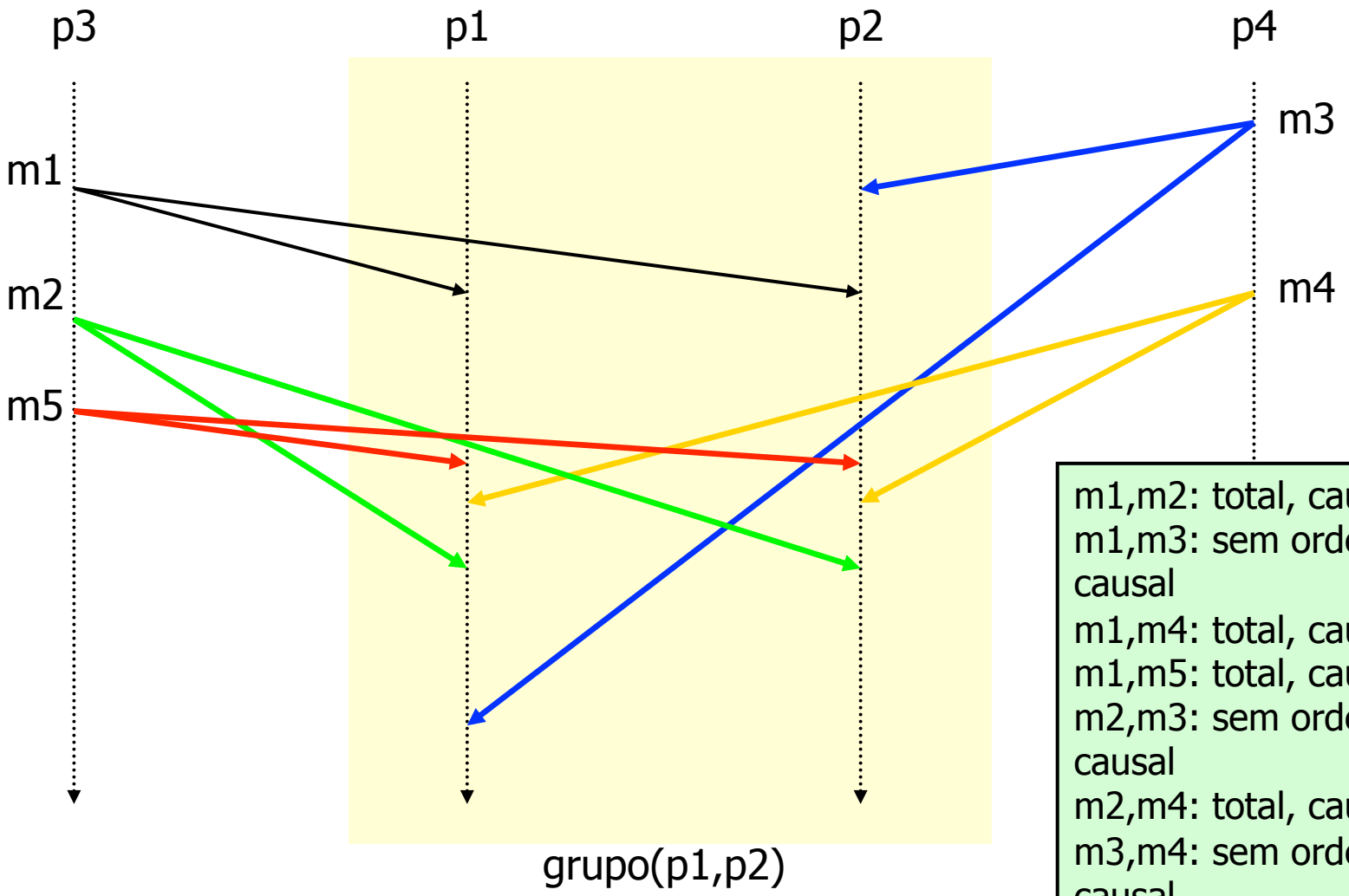
- Sem ordem – as mensagens podem ser entregues por diferentes ordens em diferentes processos
- Ordem FIFO – as mensagens de um processo são entregues pela ordem de emissão em todos os processos
- Ordem total – as mensagens m1, m2 são entregues por ordem total, se forem entregues pela mesma ordem em todos os processos
  - ∀pi, receive(m1,pi) < receive(m2,pi) ou
  - ∀pi, receive(m1,pi) > receive(m2,pi)
  - NOTA: a ordem de entrega das mensagens de um mesmo emissor pode ser diferente da ordem de emissão
  - Um sistema de multicast fiável no qual as mensagens são entregues por ordem total chama-se sistema de **multicast atómico**

Caracterização do multicast: ordem

- Ordem causal – se m1 pode causar m2, m1 deve ser entregue sempre antes de m2
  - Se duas mensagens forem emitidas pelo mesmo processo, considera-se que o envio da primeira mensagem pode causar o envio da segunda
    - Neste caso, a recepção da primeira deve acontecer antes da recepção da segunda
  - Se duas mensagens forem emitidas em processos diferentes, a primeira mensagem pode causar a segunda se for recebida antes do envio da segunda (ou transitivamente incluindo outras mensagens)
    - Se uma mensagem não pode causar a outra, qualquer ordem de entrega respeita a ordem causal

- NOTA: a noção de causalidade estende-se a qualquer tipo de comunicação

Ordens: exemplo



m1,m2: total, causal
m1,m3: sem ordem, causal
m1,m4: total, causal
m1,m5: total, causal
m2,m3: sem ordem, causal
m2,m4: total, causal
m3,m4: sem ordem, não causal
m2,m5: total, não causal, não FIFO

Caracterização do multicast: vistas

- Def: uma vista (view) é o conjunto de elementos de um grupo num dado momento.
  - Nos sistemas de comunicação em grupo fiáveis, a mudança de vista é efectuada através do envio de uma mensagem multicast
- Sincronia virtual (virtual synchrony) – um sistema de multicast fiável implementa sincronia virtual se:
  - as mudanças de vista são entregues em todos os processos pela mesma ordem
  - o conjunto de mensagens entregues entre a entrega de cada duas vistas consecutivas é idêntico para todos os processos que observam as duas vistas

Facetas da comunicação multicast: resumo

- Fiabilidade
  - Multicast fiável
  - Multicast não-fiável
- Ordenação das mensagen
  - Sem ordem
  - Ordem FIFO
  - Ordem total
  - Ordem casual
- Relativamente às vistas
  - Sincronia virtual

Sistemas de disseminação de eventos / publish/subscribe

- Extensão dos sistemas Message-Oriented Middleware (MOM) para grupos
  - Pode ser visto como um sistema de multicast com interface diferente
  - Comunicação por mensagens

- Ideia base:
  - Processos produzem mensagens que difundem
    - Produtores não conhecem subscritores
    - Interacção assíncrona
  - Subscritores recebem subconjunto das mensagens emitidas

Publish/subscribe: baseado em canais/tópicos

- Existem canais dedicados a determinados tópicos
  - Todos os subscritores recebem todas as mensagens

- Interface (e.g.):

```
Channel c = open( "SD")// abre canal para publicar mensagens
c.publish( "msg 1")    // publica mensagem
c.close()

Channel c = open( "SD")
c.subscribe( new MsgListener() {
  public void newMessage( Message msg) {
   ... // processo mensagem
  }
})
```

Publish/subscribe: com filtragem por conteúdo

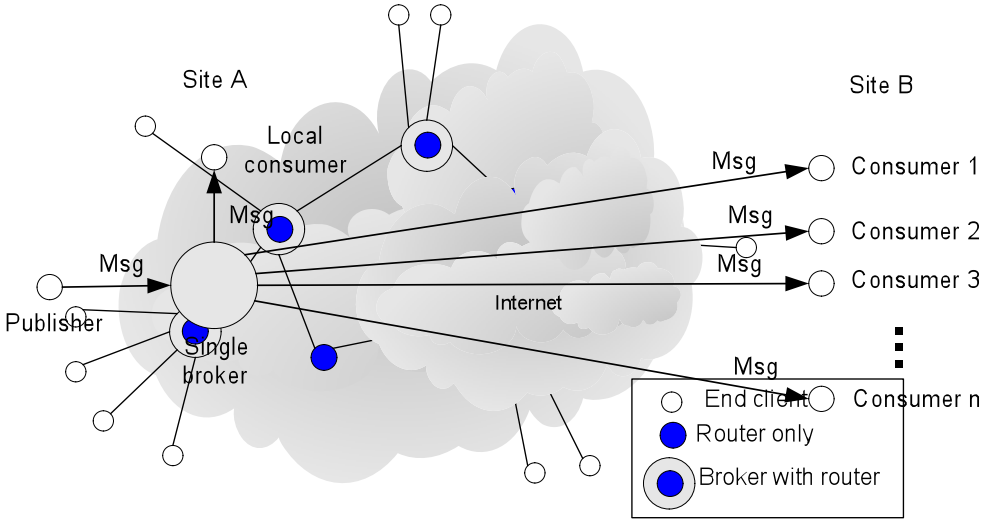
- As mensagens enviadas têm, geralmente, um conjunto de atributos definidos pelo emissor
  - Os subscritors definem quais as mensagens que estão interessados em receber através de condições sobre os atributos

- Interface (e.g.):

```
Channel c = open() // abre canal para publicar mensagens
c.publish( "msg 1", {"SD"})       // publica mensagem
c.close()

Channel c = open()
c.subscribe( new MsgFilter() {
   public boolean match( Message msg, AtributeSet set) {
     return set.contains( "SD");
   }
}, new MsgListener() {
   public void newMessage( Message msg) {
     ... // processo mensagem
   }
})
```

Publish/subscribe: arquitecturas

- Cliente/servidor
  - Permite implementar funcionalidade adicional facilmente – e.g.: persistência, filtragem



- Peer-to-peer
  - Sem ponto central de falhas, potencialmente mais escalável
    - Recorrendo a multicast tradicional ou baseado em comunicação epidémica

Publish/subscribe: propriedades

- Desacoplamento entre componentes
  - Vantagens:
    - Emissor não necessita de conhecer receptor
    - Emissor e receptor não necessitam de estar a funcionar ao mesmo tempo
    - Emissor não necessita de esperar pelo receptor - assincronismo
    - Melhor desempenho
  - Desvantagens:
    - Mais complicado definir propriedades exactas do sistema

- Modelo de programação *event-driven*
  - Modelo reactivo - como nos GUIs

Publish-subscribe: sistemas e *standards*

- Web-services
  - WS-Notification
    - Tem mais funcionalidade que WS-Eventing, mas são mutuamente incompatíveis
  - WS-Eventing
- Java: JMS
- Corba:
  - Event service
  - Notification service
- Sistemas:
  - TiBCO, IBM Gryphon, Apache pubscribe

Sistemas: como implementar?

- Suponham que têm um backbone de caches de internet, uma por ISP em cada país
  - (nota: por exemplo, a akamai tem esta infraestrutura)
- Como actualizar as caches?
- ==========
- Sistema de controlo de tráfego áereo
  - Radares monitorizam espaço aéreo
  - Quem está interessado na informação? Controlo de tráfego aérea, companhias, aeroportos
- ==========
- Editor de texto cooperativo

Sistema: como implementar?

- Sistema de disseminação de informação da bolsa
- ==========
- VC2

Para saber mais

- G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems – Concepts and Design, Addison-Wesley, 4th Edition, 2005
  - capítulo 4 : partes sobre heterogeneidade e protocolos de invocação remota serão abordados no próximo capítulo
  - secção 12.4 – apenas parte relativa aos tipos de multicast (não entrar na parte de implementação)

Sistemas Distribuídos I

Capítulo 4

Invocação de procedimentos e de métodos remotos

Nota prévia

- A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

  G. Coulouris, J. Dollimore and T. Kindberg,
  Distributed Systems - Concepts and Design,
  Addison-Wesley, 4th Edition, 2005

- Para saber mais:
  - RMI/RPCs - capítulo 5.
  - Representação de dados e protocolos - capítulo 4.
  - Web services – capítulo 19.1-19.4
  - Corba – capítulo 20 (interessante para quem queira saber mais).

Organização do capítulo

- Invocação remota de procedimentos/objectos
  - Modelo
  - Definição de interfaces
  - Métodos de passagem de parâmetros (heterogeneidade e representação de dados)
  - Mecanismos de ligação (binding)
  - Semântica na presença de falhas
  - Concorrência no servidor
- Estudo de caso

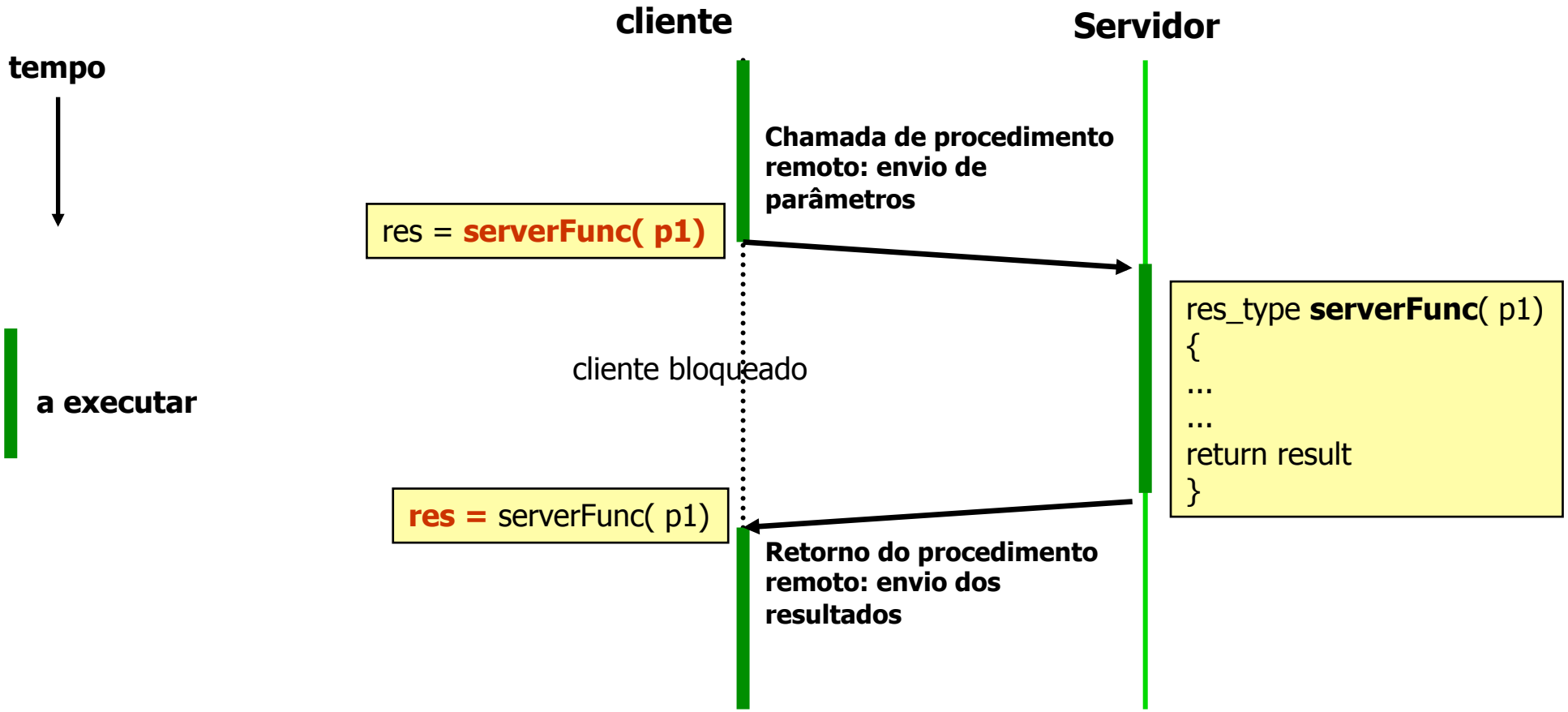
Invocação de procedimentos remotos: motivação

- É possível estruturar uma aplicação distribuída usando como base as interacções através da troca de mensagens entre processos. Os problemas são:
  - é complicado e cheio de detalhes não relevantes
  - os programas ficam estruturados em função dos protocolos (trocas de mensagens)
  - os servidores ficam estruturados em função da lista de mensagens que sabem processar
- Muitas linhas de código são repetitivas, não contêm nenhum significado aplicacional específico e referem-se ao processamento das comunicações:
  - criação de communication end points e sua associação aos processos
  - criação, preenchimento e interpretação das mensagens
  - selecção do código a executar consoante o tipo da mensagem recebida
  - gestão de temporizadores/tratamento das falhas
- Não se poderão automatizar as interacções cliente/servidor?

Invocação remota de procedimentos

- Num programa definido numa linguagem imperativa, definem-se funções/procedimentos para executar uma dada operação
- Uma extensão natural num ambiente distribuído consiste em permitir a execução de procedimentos noutra máquina
  - Ex. ONC/RPC, DCE
- Se os procedimentos estiverem definidos no âmbito de um objecto, chama-se invocação remota de métodos
  - Ex.: Java RMI, CORBA e .NET
  - Web services: permitem efectuar invocação remota de procedimentos usando mensagens (geralmente) representadas em XML e enviadas usando HTTP
    - Web services permitem interacções mais complexas que simples pedido/resposta

Invocação de procedimentos remotos (RPCs)

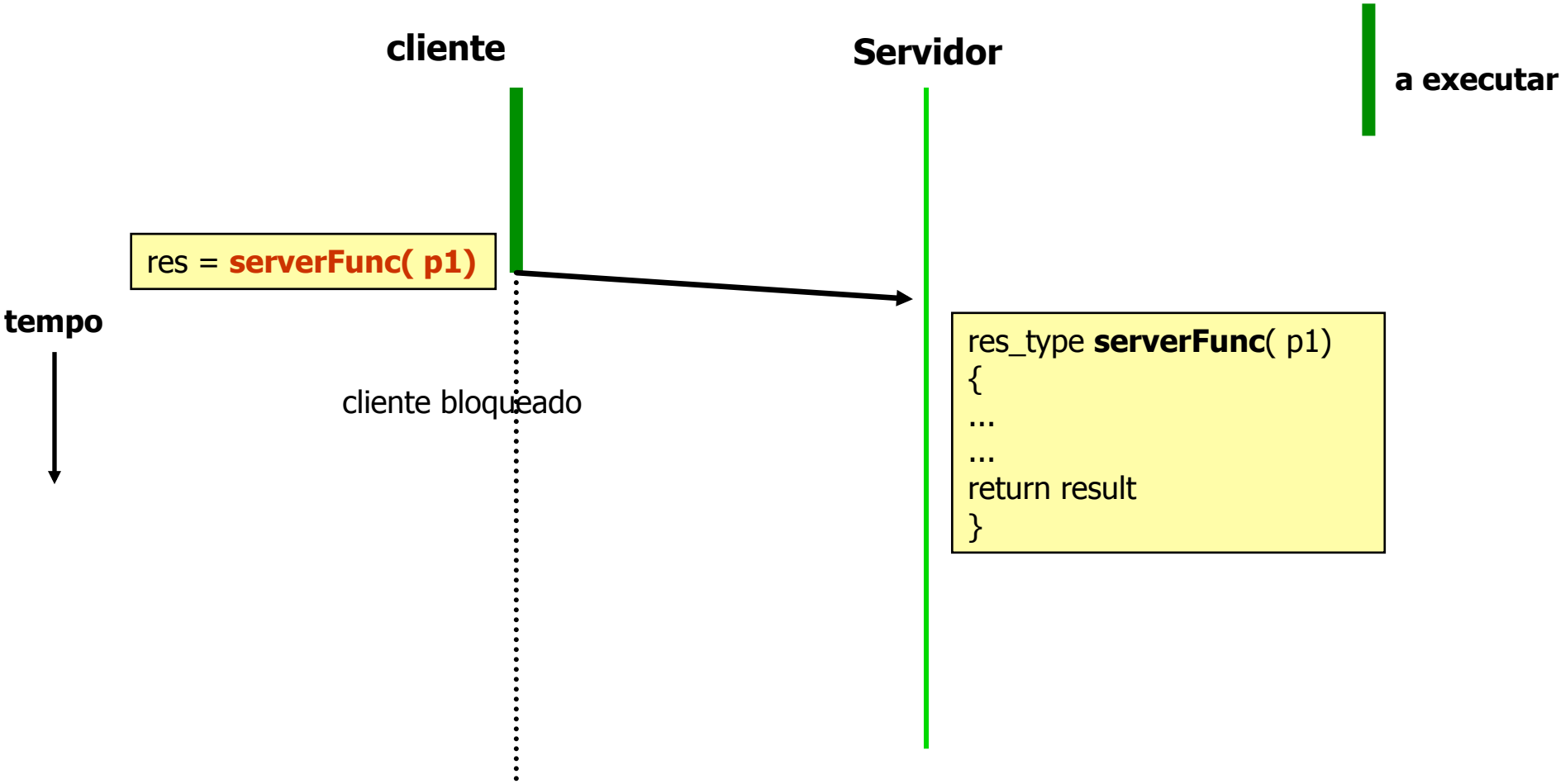


- Modelo
  - Servidor exporta interface com operações que sabe executar
  - Cliente invoca operações que são executadas remotamente e (normalmente) aguarda pelo resultado

Invocação de procedimentos remotos (cont.)

- Propriedades
  - Extensão natural do paradigma imperativo/procedimental a um ambiente distribuído
    - Chamada síncrona de funções – modelo de comunicação ?
  - Esconde detalhes de comunicação (e tarefas repetitivas)
    - Construção, envio, recepção e tratamento das mensagens
    - Tratamento básico de erros (devem ser tratados ao nível da aplicação)
    - Heterogeneidade da representação dos dados
  - Simplifica disponibilização de serviços
    - Interface bem definida, facilmente documentável e independente dos protocolos de transporte
    - Sistema de registo e procura de serviços

RPCs: como implementar



Como esconder os detalhes?
(sem suporte específico do runtime da linguagem)

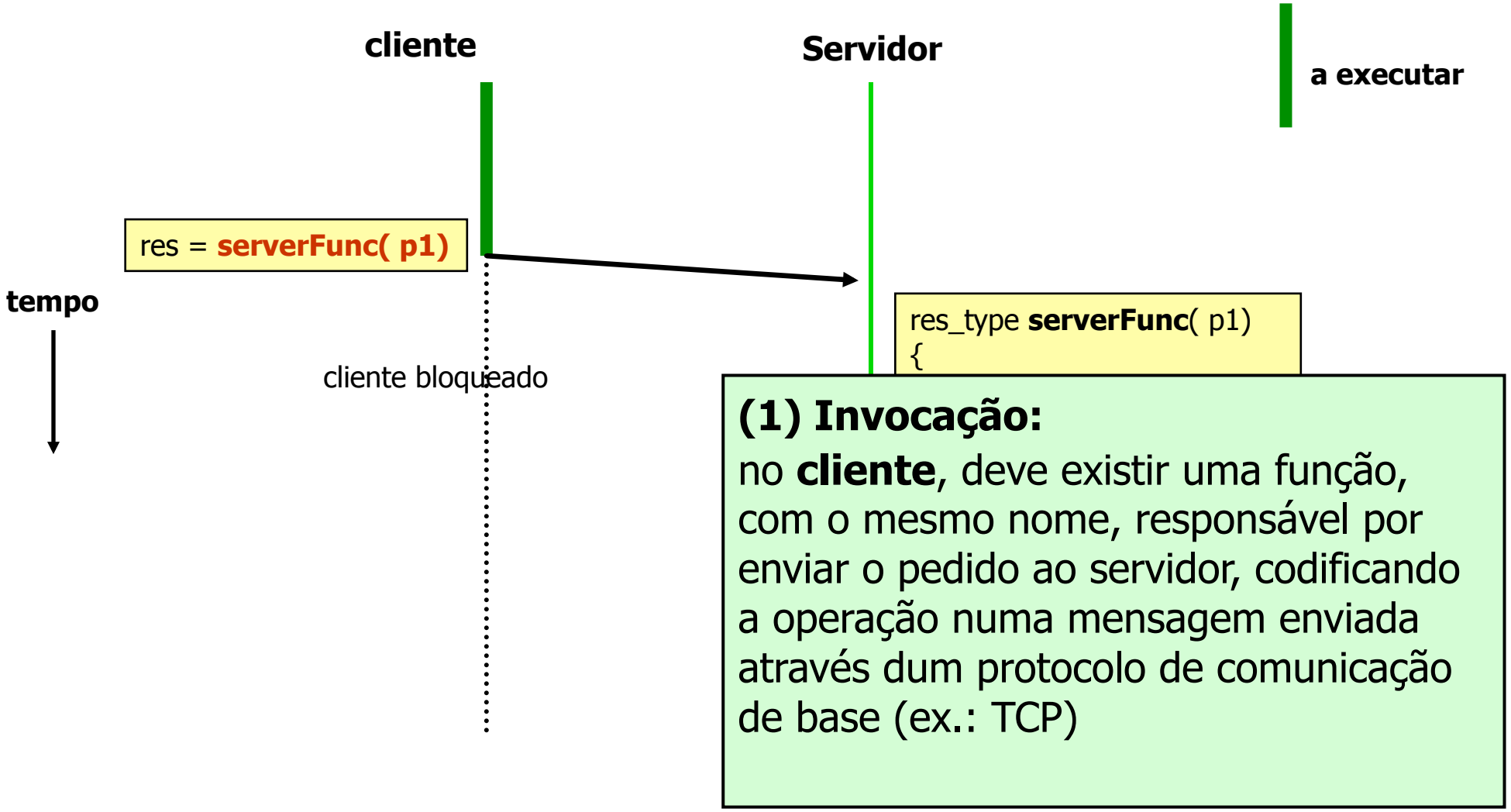
Cliente

```
res = serverFunc( p1)
```

Servidor

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

RPCs: como implementar



(1) Invocação:
no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

Como esconder os detalhe
(sem suporte específico do runtime d

(1) Invocação:
no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

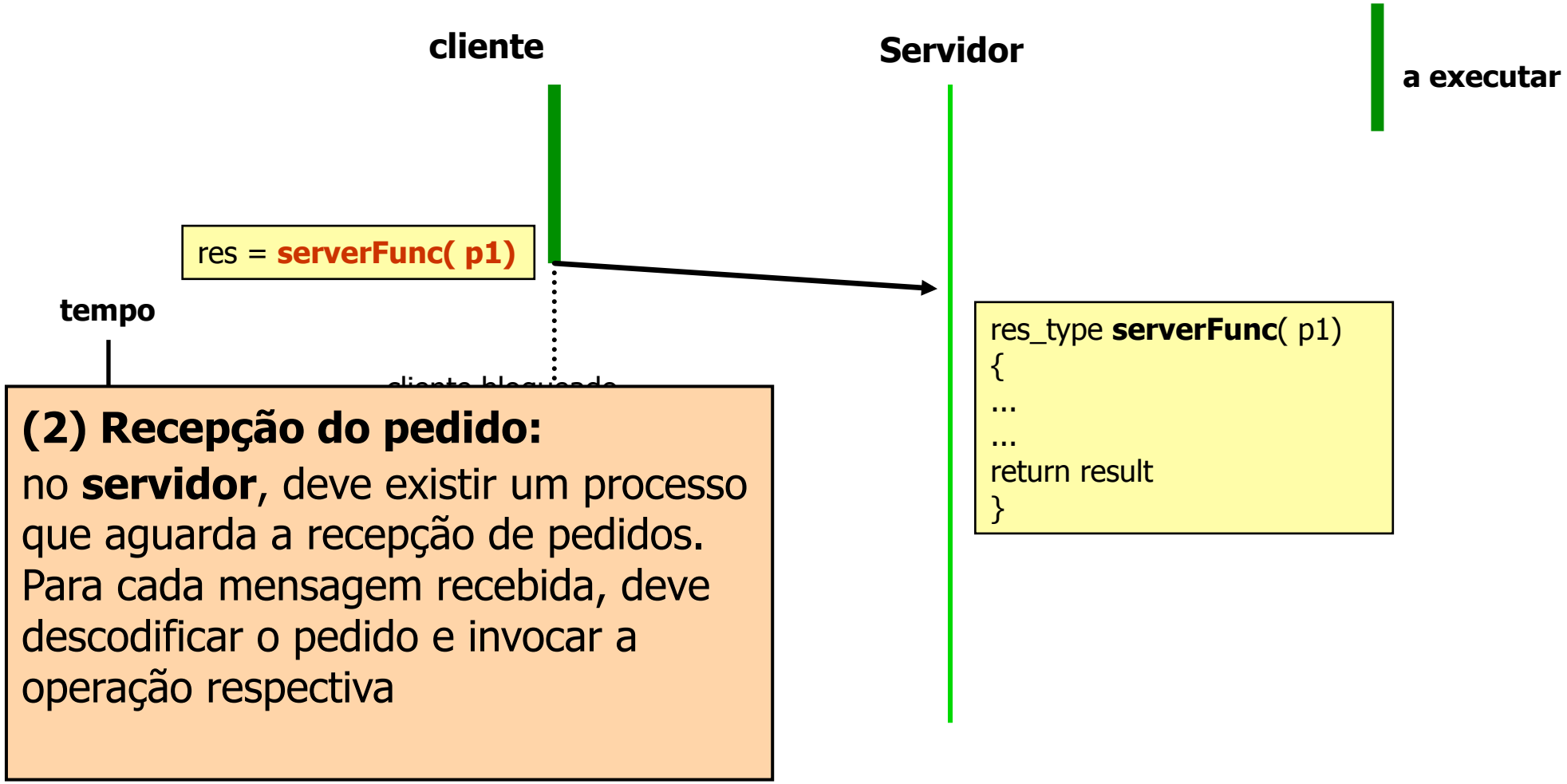
Cliente

```
res = serverFunc( p1)
```

```
}
```

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

RPCs: como implementar



(2) Recepção do pedido:
no **servidor**, deve existir um processo que aguarda a recepção de pedidos. Para cada mensagem recebida, deve descodificar o pedido e invocar a operação respectiva

Como são os detalhes? (da linguagem)

(2) Recepção do pedido:
no **servidor**, deve existir um processo que aguarda a recepção de pedidos. Para cada mensagem recebida, deve descodificar o pedido e invocar a operação respectiva

Servidor

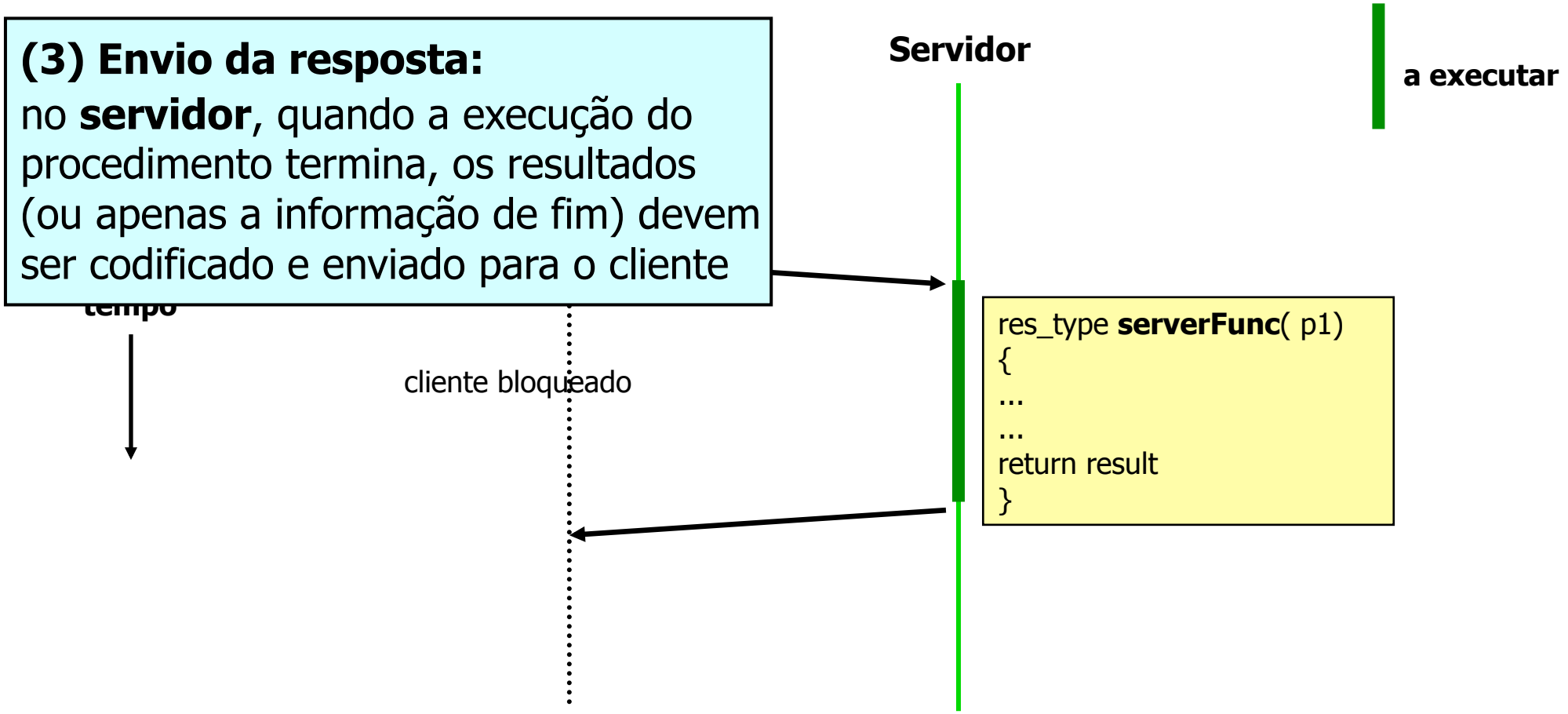
```
res_type serverFunc( T1 p1) {
        ...
        return result
}
```

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

```
s = new ServerSocket
forever
    Socket c = s.accept();
    c.receive( msg( op, params))
    if( op = "serverFunc")
        res = serverFunc( params[0]);
    else if( op = ...)
        ...
```

RPCs: como implementar

(3) Envio da resposta:
no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente



Como esconder os detalhes?
(sem suporte específico do runtime da linguagem)

(3) Envio da resposta:
no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

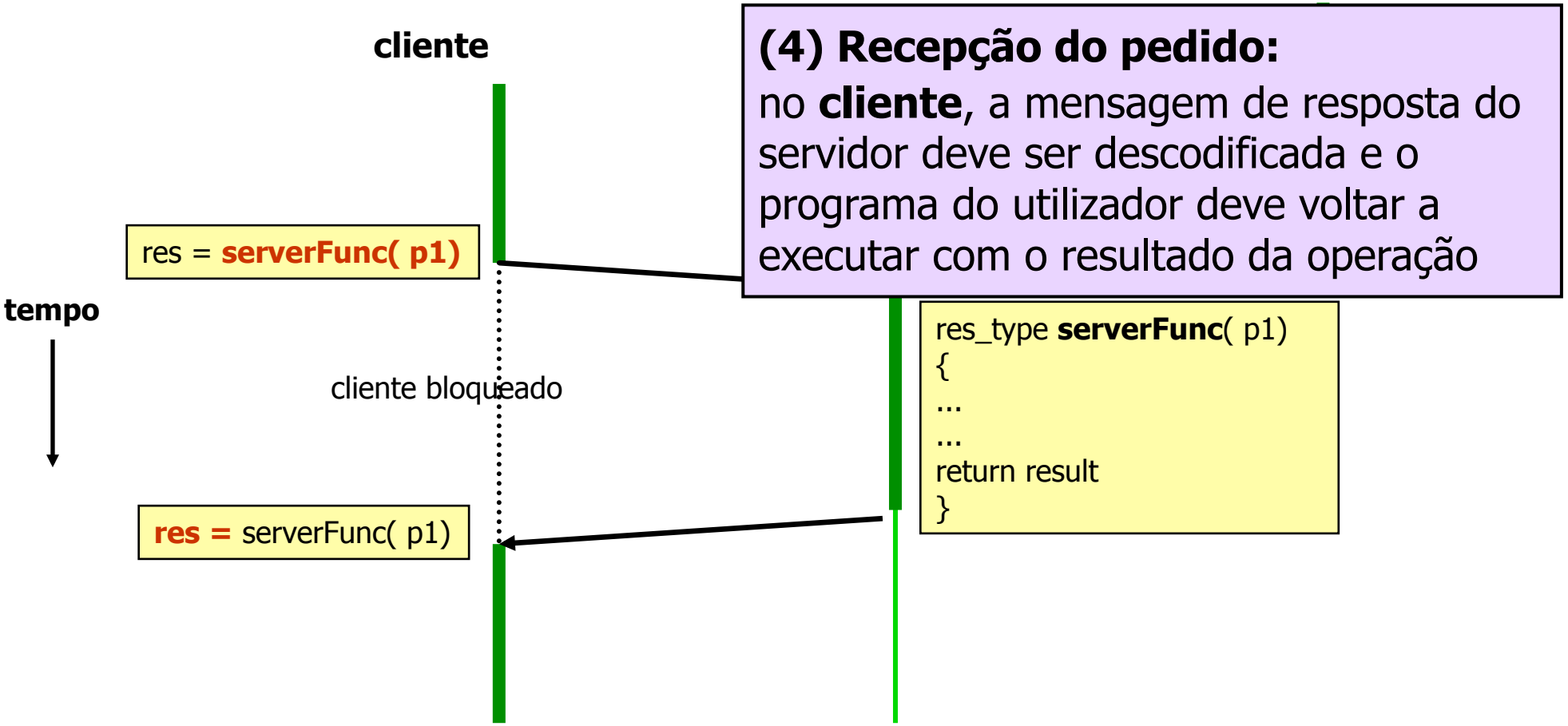
Servidor

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

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

```
s = new ServerSocket
forever
    Socket c = s.accept();
    c.receive( msg( op, params))
    if( op = "serverFunc")
        res = serverFunc( params[0]);
    else if( op = ...)
        ...
    c.send( msg(res))
    c.close
```

RPCs: como implementar



Como esconder os detalhes?
(sem suporte específico do runtime da linguagem)

(4) Recepção do pedido:
no **cliente**, a mensagem de resposta do servidor deve ser descodificada e o programa do utilizador deve voltar a executar com o resultado da operação

Cliente

```
res = serverFunc( p1)
```

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

```
s = new ServerSocket
forever
    Socket c = s.accept();
    c.receive( msg( op, params))
    if( op = "serverFunc")
        res = serverFunc( params[0]);
    else if( op = ...)
        ...
    c.send( msg(res))
    c.close
```

Como esconder os detalhes?
(sem suporte específico do runtime

Cliente

```
res = serverFunc( p1)
```

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

Na prática, sucessivas invocações podem partilhar o mesmo socket…

Stub do cliente ou *proxy* do servidor

(4) Recepção do pedido:
no **cliente**, a mensagem de resposta do servidor deve ser descodificada e o programa do utilizador deve voltar a executar com o resultado da operação

(1) Invocação:
no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

Como esconder os detalhes?
(sem suporte específico do runtime da linguagem)

Stub ou skeleton do servidor

(3) Envio da resposta:
no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

(2) Recepção do pedido:
no **servidor**, deve existir um processo que aguarda a recepção de pedidos. Para cada mensagem recebida, deve descodificar o pedido e invocar a operação respectiva

Servidor

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

```
s = new ServerSocket
forever
    Socket c = s.accept();
    c.receive( msg( op, params))
     if( op = "serverFunc")
         res = serverFunc( params[0]);
     else if( op = ...)
        ...
    c.send( msg(res))
    c.close
```

Automatização do processo - *Stub* ou *proxy compilers*

- Nos sistemas de RPC/RMI pode existir uma ferramenta (*stub compiler*) que gera automaticamente o código que trata da parte de comunicação
  - Este código é gerado com base na (interface do) servidor
  - Stub do cliente inclui funções do cliente com o mesmo nome que efetuam a comunicação com o servidor para efectuar a invocação remota
  - Stub do servidor inclui código de comunicação para esperar invocações e executá-las, devolvendo o resultado
  - Stubs incluem código para codificar os parâmetros e resultado
  - Nos sistemas mais modernos, por vezes, existem stubs genéricos que podem ser usados com todos os tipos (e.g. Java)
  - Algumas linguagens/sistemas incluem suporte para invocação remota, pelo que as comunicações são tratadas pelo *runtime* de suporte (e.g. .NET remoting)

Aspectos a considerar num sistema de RPC/ RMI

- Linguagem de definição de interfaces (IDL - [Network] Interface Definition Language)
- Modelo de passagem de parâmetros, resultados e excepções
  - Formato de trasnmissão dos dados
- Forma de localização do servidor/objecto remoto (*binding*)
- Modelos de falhas e protocolos de interacção entre o cliente e o servidor
- Arquitectura do cliente e do servidor (multiprogramação) e outros aspectos de desempenho
- Segurança (autenticação, controlo de acessos, …)

RPC/RMI Interface Definition Languages (IDL)

- IDLs são linguagens que permitem definir interfaces de servidores/objectos remotos, especificando:
  - Tipos e constantes
  - Interface do serviço - assinatura das funções/procedimentos

- Os IDLs são usados apenas para definir as interfaces, não o código das operações
  - Por vezes, esta distinção é difícil de fazer porque os IDLs estão integrados com linguagem
  - Em certos sistemas (e.g. .NET remoting), a interface pode não ser definida autonomamente

IDLs – aproximações possíveis

- Usar sub-conjunto de uma linguagem já existente
  - Ex.: Java RMI
- Definir linguagem específica para especificar interfaces dos servidores/ objectos remotos
  - Ex.: DCE RPCs, WSDL
  - Geralmente baseado numa linguagem existente
  - Adição de tipos especiais (exemplo: SUN-RPC string, opaque)
  - Necessidade de mapear o IDL e as linguagens de desenvolvimento dos clientes/servidores

Exemplo: DCE - Distributed Computing Environment (adoptado pela Microsoft)

```
[ uuid(b20a1705-3c26-12d8-8ea3-04163a0dcefz) version (1.0)]

interface readwritefs              {
    const long FILE_NAME_SIZE = 16;
    const long BUFFER_SIZE = 1024;
    typedef char FileName [ FILE_NAME_SIZE ];
    typedef char Buffer [ BUFFER_SIZE ];

    void read (       [in] FileName        filename;
                      [in] long    position;
                      [in, out] long       nbytes;
                      [out] Buffer         buffer;     );

    void write (      [in] FileName        filename;
                      [in] long    position;
                      [in, out] long       nbytes;
                      [in] Buffer buffer;  );
}
```

uuid permite identificar univocamente o servidor

podem-se definir constantes e tipos auxiliares

parâmetros podem ser de entrada, saída ou ambos

Exemplo de uma interface remota em CORBA IDL

```
module PersonModule {
    struct Person {
            string name;
            string place;
            long year;
    } ;
    interface PersonList {
            // types
            enum account_kind {checking, saving};
            // excepções
            exception  person_not_found { string name; };
            // atributos
            readonly attribute string listname;
            //operações
            oneway void addPerson(in Person p) ;
            void getPerson(in string name, out Person p)
                    raises (person_not_found);
            long getNumbernumber();
    };
};
```

Podem-se definir tipos auxiliares

Podem-se definir tipos para erros

Para cada atributo são definidas operações para aceder e mudar o valor

oneway: cliente não se bloqueia

Interface remota em Java RMI

```
public interface ContaBancaria
                              extends Remote
 {

    public void depositar ( float quantia )
                              throws RemoteException;

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

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

Interfaces remotos estendem **Remote**

Métodos devem lançar **RemoteException** para tratar erros de comunicação

Interfaces definidos em Java *standard*

Interface definida em C# para .NET Remoting

```
using System;
namespace IRemoting
{
   public interface ContaBancaria
   {
         double SaldoActual
         {
               get;
         }
         void depositar ( float quantia );
         void levantar (float quantia);
   }
}
```

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

Interface definida em C# comum

Interface definida em C# para .NET Remoting

```
using System;
namespace IRemoting
{
    public interface ContaBancaria
    {
            double SaldoActual
            {
                    get;
            }
            void depositar ( float quantia );
            void levantar (float quantia);
    }
}
```

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

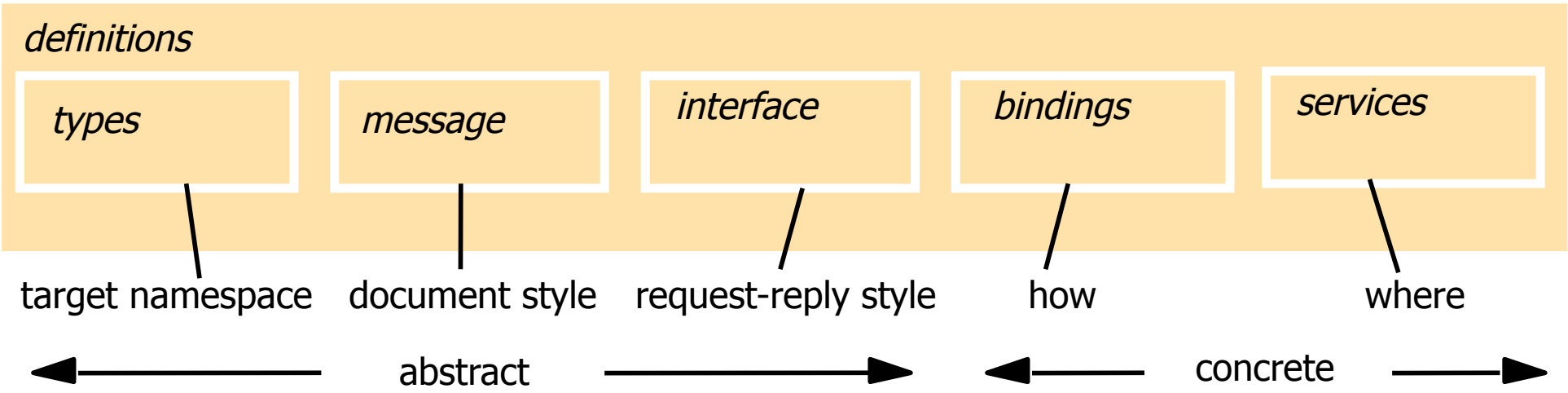
No .NET Remoting não é necessário definir qual a interface remota – esta pode ser inferida a partir da definição do servidor

Um objecto remoto deve estender MarshalByRefObject

WSDL – IDL para *web services*

- Definição da interface em XML
  - WSDL permite definir a interface do serviço, indicando quais as mensagens trocadas na interacção
  - WSDL permite também definir a forma de representação dos dados e a forma de aceder ao serviço

  - Especificação WSDL bastante verbosa – normalmente criada a partir de interface ou código do servidor
    - Ex. JAX-WS tem ferramentas para criar especificação a partir de interfaces Java



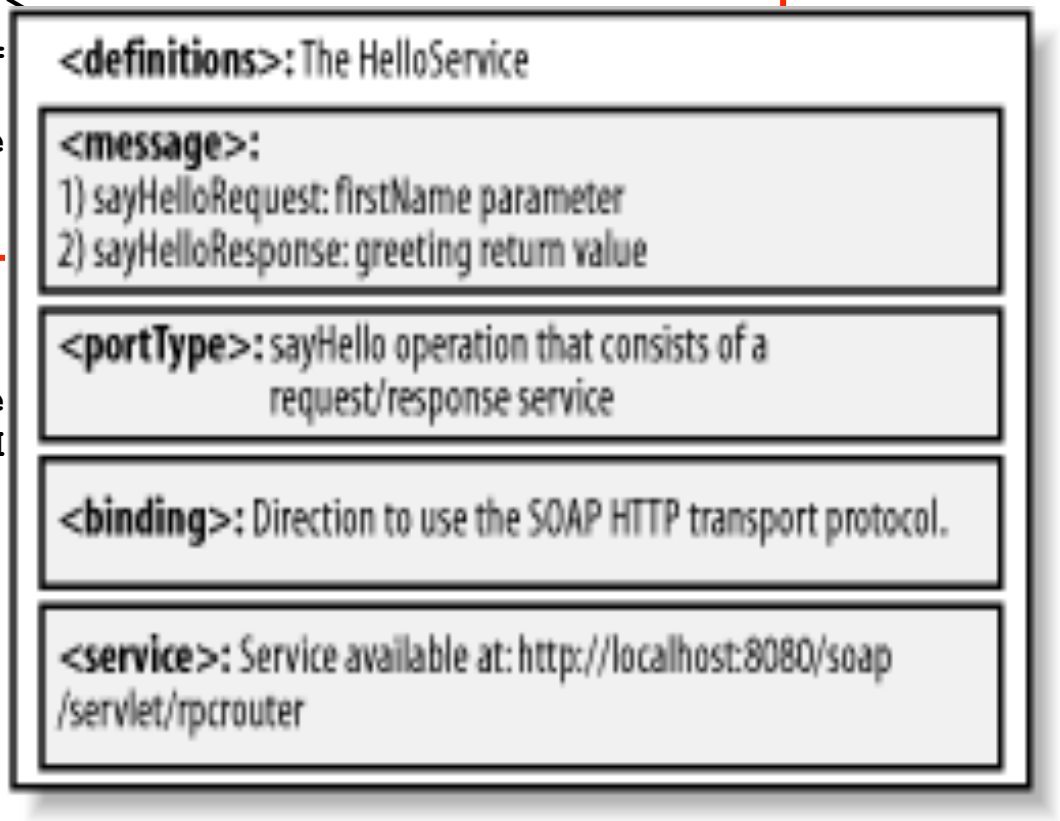
WSDL - exemplo

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
   targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="SayHelloRequest">
      <part name="firstName" type="xsd:string"/>
   </message>
   <message name="SayHelloResponse">
      <part name="greeting" type="xsd:string"/>
   </message>

   <portType name="Hello_PortType">
      <operation name="sayHello">
         <input message="tns:SayHelloRequest"/>
         <output message="tns:SayHelloResponse"/>
      </operation>
   </portType>
```

(exemplo do livro Web Services Essentials, O'Reilly, 2002.)

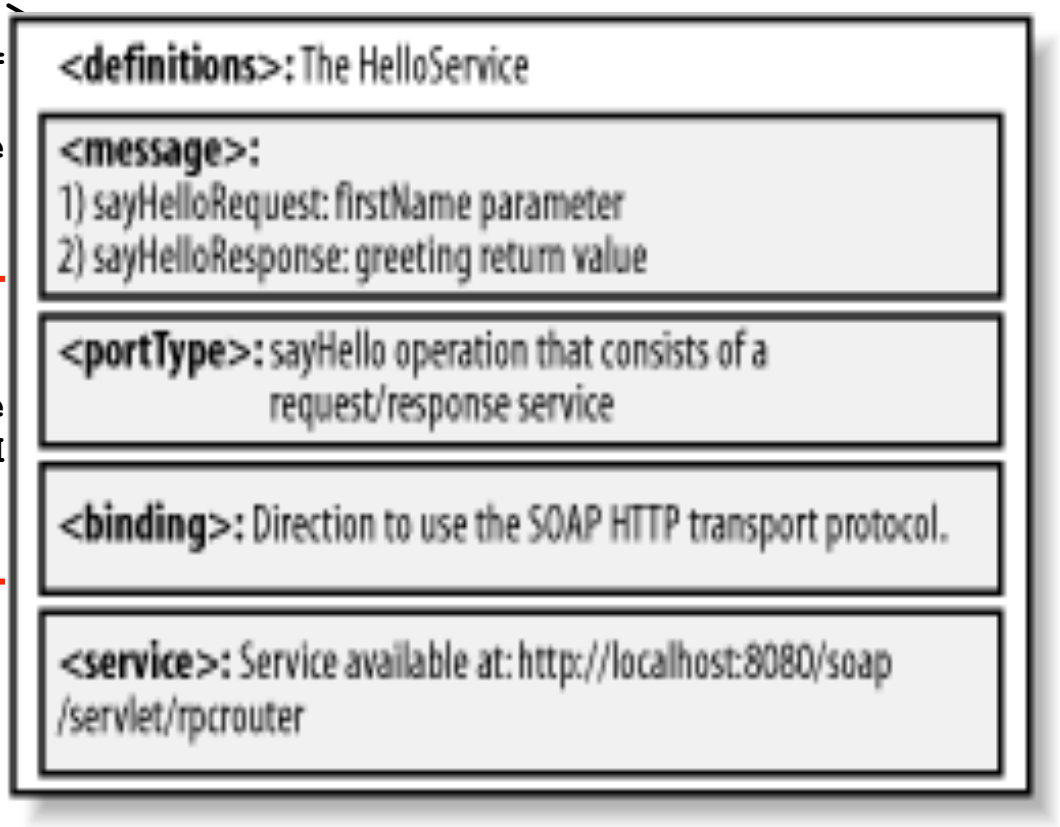
WSDL - exemplo

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
   targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

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

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



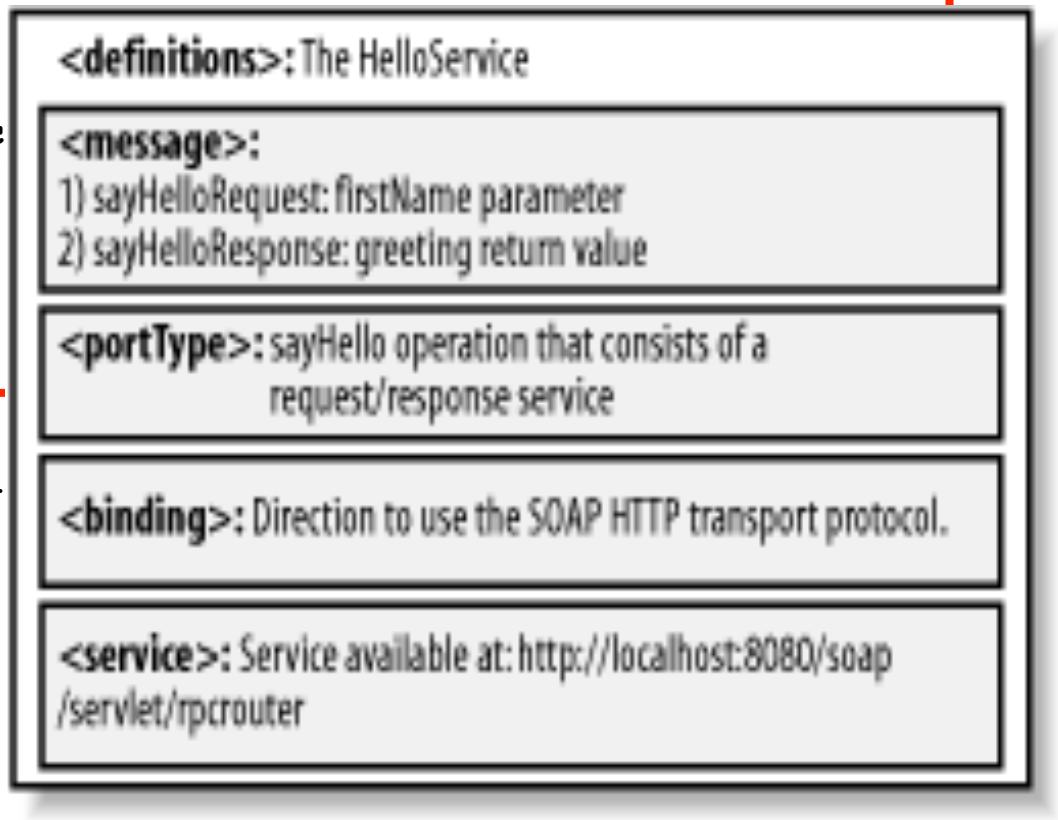
WSDL - exemplo

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
   targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

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

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



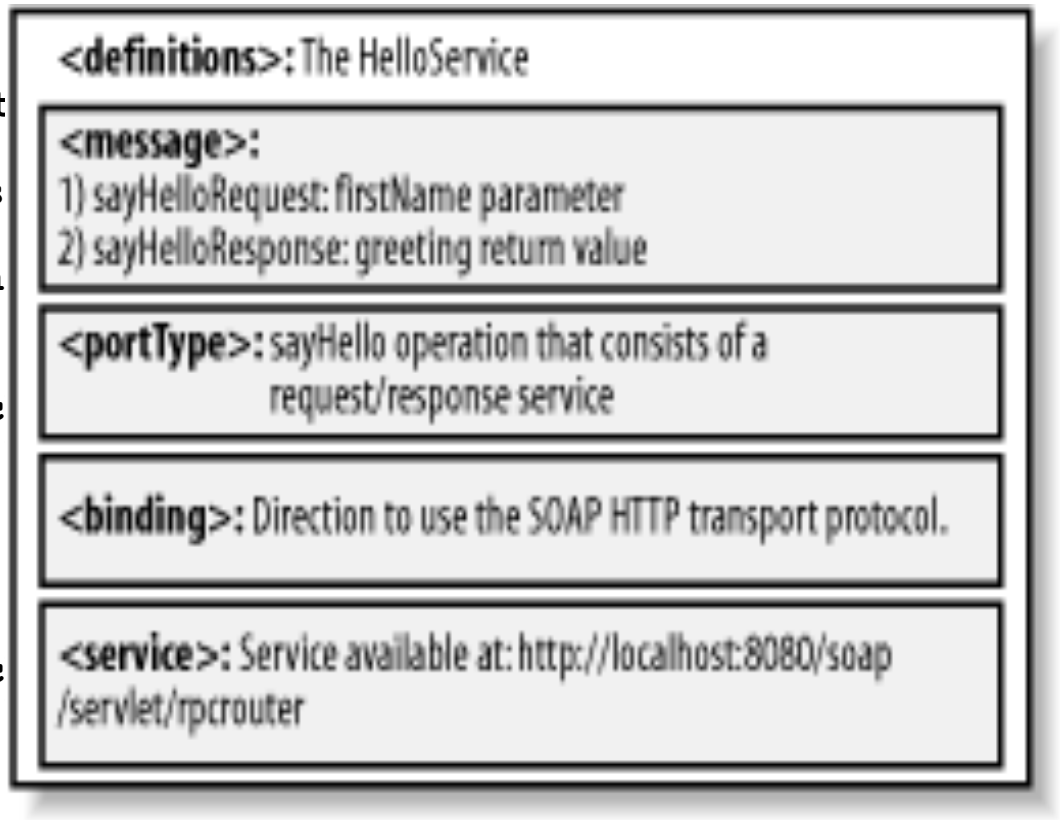
WSDL - exemplo

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

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



WSDL - exemplo

```
<binding name="Hello_Binding" type="t
    <soap:binding style="rpc"
       transport="http://schemas.xmls
    <operation name="sayHello">
       <soap:operation soapAction="sa
       <input>
          <soap:body
             encodingStyle="http://sc
             namespace="urn:examples:
             use="encoded"/>
       </input>
       <output>
          <soap:body
             encodingStyle="http://sc
             namespace="urn:examples:
             use="encoded"/>
       </output>
    </operation>
 </binding>

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



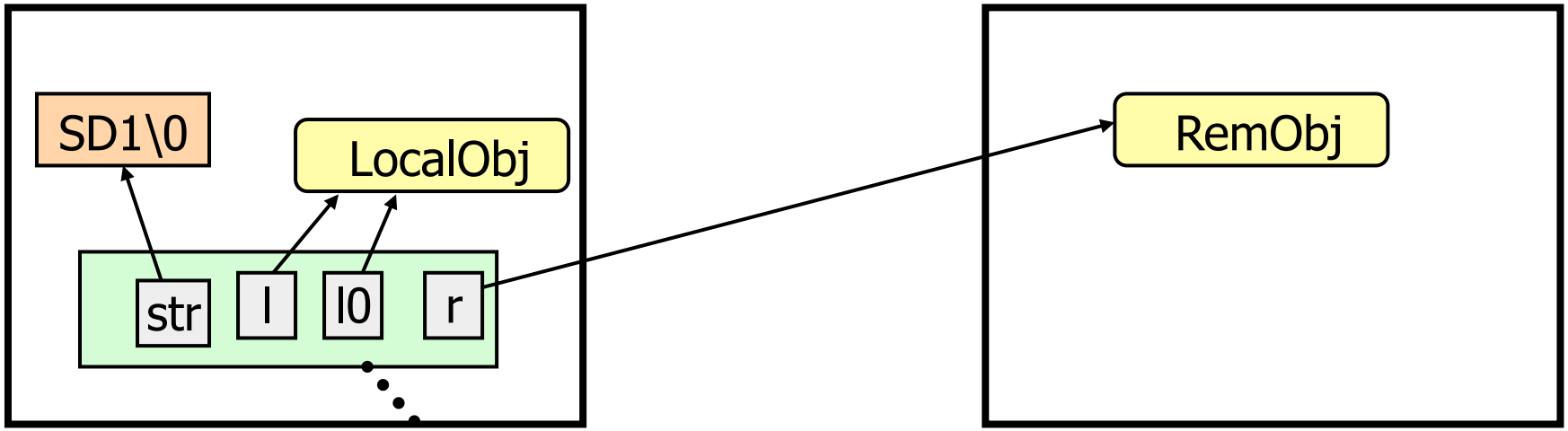
Métodos de passagem de parâmetros

- Independentemente dos tipos dos parâmetros, os mesmos podem ser:
  - parâmetros de entrada (in) : cópia no pedido
  - parâmetros de saída/resultado (out): cópia na resposta
  - parâmetros de entrada/saída (in/out): cópia no pedido e na resposta

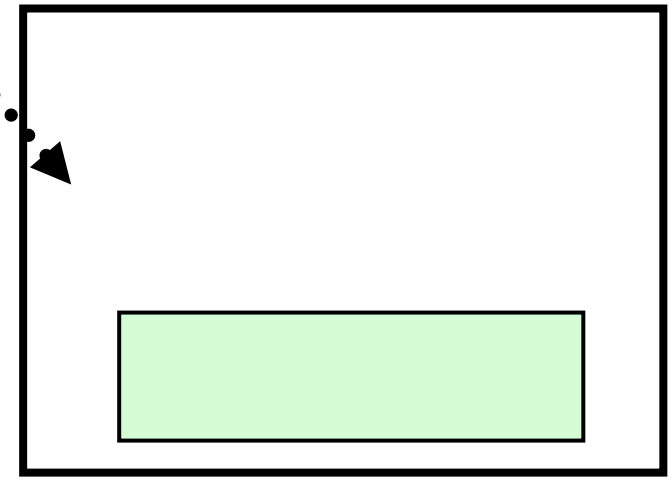
Métodos de passagem de parâmetros (cont.)

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

Métodos de passagem de parâmetros: exemplo



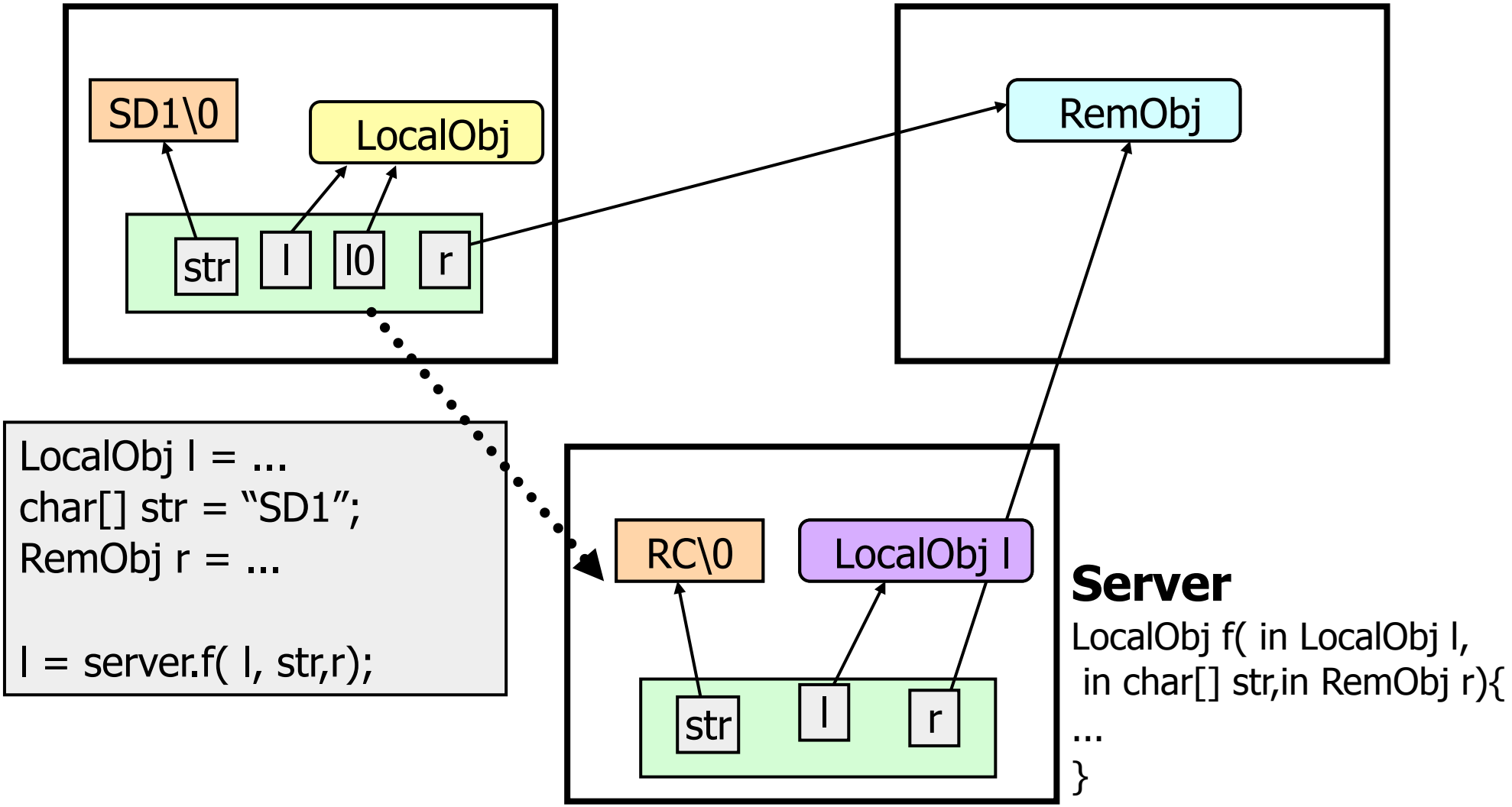

```
LocalObj l = ...
char[] str = "SD1";
RemObj r = ...

l = server.f( l, str,r);
```



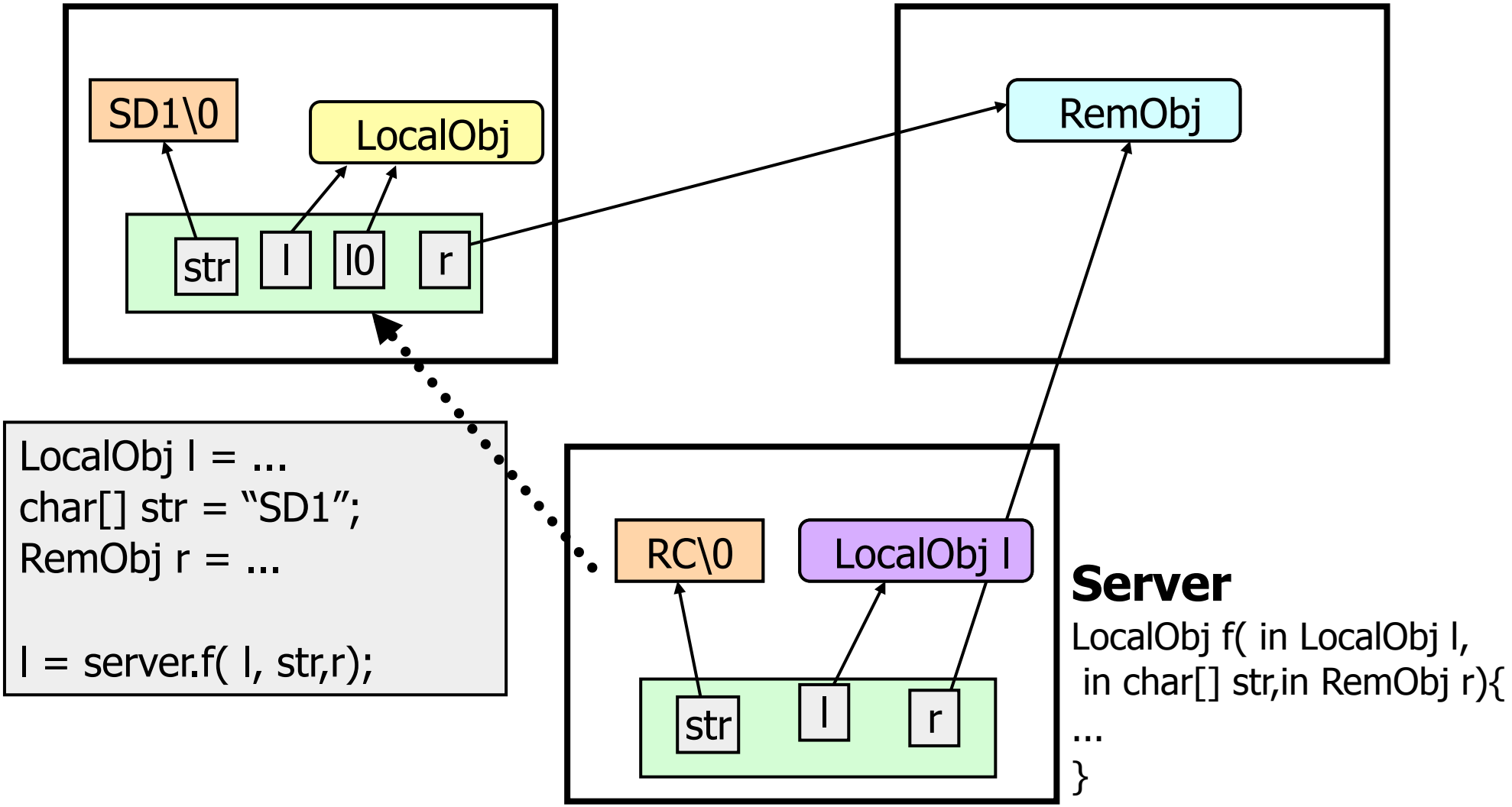
Server

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

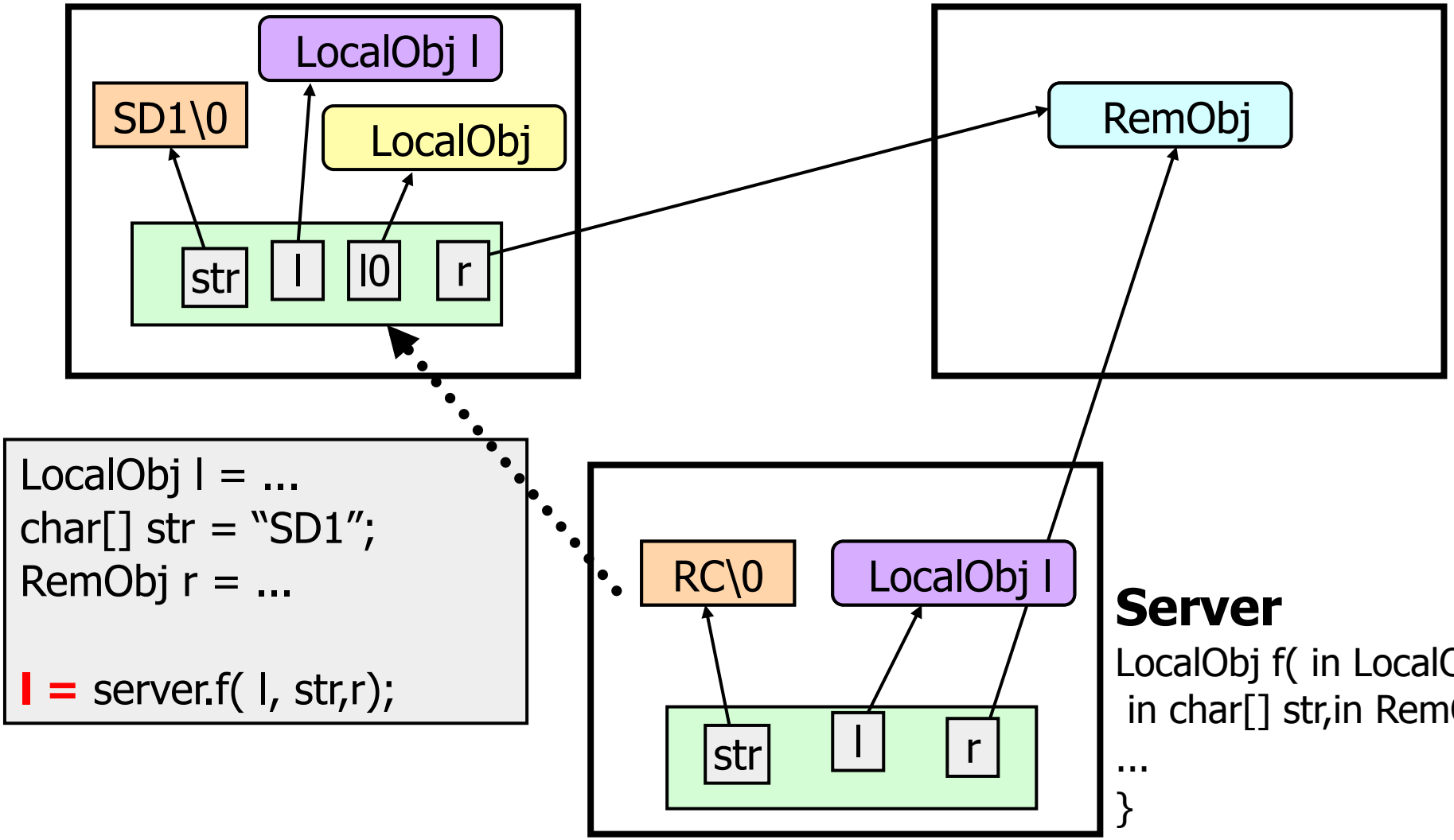
Métodos de passagem de parâmetros: exemplo



Métodos de passagem de parâmetros: exemplo



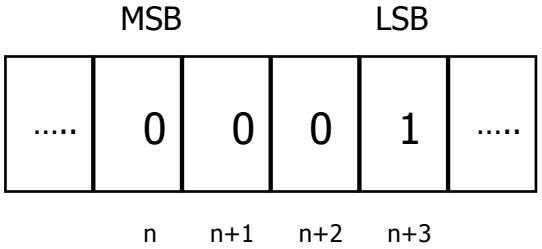
Métodos de passagem de parâmetros: exemplo



Representações dos dados – tipos primitivos

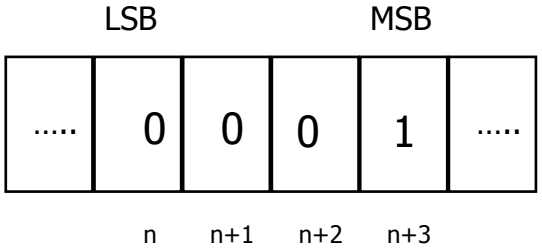
- Diferentes sistemas representam os tipos primitivos de formas diferentes
  - Inteiros armazenados por ordem diferente em memória
  - Diferentes representações para números reais
  - Caracteres com diferentes codificações
- Simples transmissão dos valores armazenados pode levar a resultados errados

"big-endian"



conteúdo da palavra
= 1

"little-endian"



conteúdo da palavra
= 1 x 256 x 256 x 256

Representações dos dados – tipos complexos

- Aplicações manipulam estruturas de dados complexas
  - Ex.: representadas por grafos de objectos
- Mensagens são sequências de bytes
- O que é necessário fazer para propagar estrutura de dados complexa?
  - É necessário convertê-la numa sequência de bytes
  - Por exemplo, para um objecto é necessário:
    - Converter as variáveis *internas*, incluindo outros objectos
    - Necessário lidar com ciclos nas referências

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

Aproximações à codificação dos dados

- Utilização de formato intermédio independente (network standard representation)
  - Emissor converte da representação nativa para a representação da rede
  - O receptor converte da representação da rede para a representação standard
- Utilização do formato do emissor (receiver makes it right)
  - Emissor envia usando a sua representação interna e indicando qual ela é
  - Receptor, ao receber, faz a conversão para a sua representação
- Utilização do formato do receptor (sender makes it right)

- Propriedades:
  - Desempenho ?
    - rep. intermédia tem pior desempenho - exige duas transformações
  - Complexidade (número de transformações a definir) ?
    - rep. intermédia exige apenas que em cada plataforma se saiba converter de/para formato intermédio

CDR - CORBA Common Data Representation

CDR primitive or simple data types

Tipo	Wire size (bytes)	Descrição
short	2	16 bits signed binary integer
unsigned short	2	16 bits unsigned binary integer
long	4	32 bits signed binary integer
unsigned long	4	32 bits unsignedbinary integer
float	4	single precision floating point number
double	8	double precision floating point number
Boolean	4	boolean value (0 or 1)
char	1	ASCII char
octet	1	any 8 bits
any		any type

Inteiros são *big-endian* ou *little-endian* **de acordo com a ordem do emissor** (enviada em cada mensagem).
Floats são IEEE, também *big-endian* ou *little-endian*

CDR - CORBA Common Data Representation

CDR "constructed data types"

Tipo de dados	Representação
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order
array	array elements in order (no length specified because it is fixed)
struct	in order of declaration of components
enumerated	unsigned long
union	type tag followed by the selected number

CDR - CORBA Common Data Representation

CDR "constructed data types"

Tipo de dados	Representação
sequence	length (unsigned long) follo
string	length (unsigned long) follo
array	array elements in order (no length specified beca
struct	in order of declaration of components
enumerated	unsigned long
union	type tag followed by the selected number

```
typedef sequence <Shape, 100> All;
```

```
typedef string<8> SmallString;
```

```
typedef GraphicalObject GO[10][8]
```

```
struct GraphicalObject {
      string type;
      Rectangle enclosing;
      boolean isFilled;
}
```

```
enum Rand
(Exp, Number);
```

```
union Exp switch (Rand)  {
      case Exp: string vote;
      case Number: long n;
}
```

CORBA CDR wire message

index in sequence of bytes ←	4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1934	*unsigned long*

```
struct Person {
    string name;
    string place;
    long year;
};

{'Smith', 'London', 1934}
```

- Dados com dimensão variável são alinhados preenchendo com 0s posições não usadas
- Não é enviada informação sobre os tipos. Como funciona?
  - Assume-se que o emissor e o receptor sabem os tipos que esperam
  - São geradas funções para fazer o *marshalling* e o *unmarshalling*

Java serialization

Serialized values			*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name	java.lang.String place	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

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

```
public class Person
  implements Serializable
{
    private String name;
    private String place;
    private int year;
    ...
}
```

- Assume-se que o processo de *deserialization* não tem informação sobre os objectos serializados
  - Forma serializada inclui informação dos tipos
- Serialização grava estado de um grafo de objectos
  - A cada objecto é atribuído um *handle*. Permite escrever apenas uma vez cada objecto, mesmo quando existem várias referência para o mesmo no grafo de objectos.

Serialização de objectos

- Permite codificar/descoficar grafos de objectos
  - Detecta e preserva ciclos pois incorpora a identidade dos objectos no grafo
- Adaptável em cada classe (os métodos responsáveis podem ser redefinidos)
- Os objectos devem ser serializáveis
  - por omissão não são – porquê?
    - poderia abrir problemas de segurança. Exemplo?
    - Permitia acesso a campos private, por exemplo.
- Os campos static e transient não são serializados
- Usa *reflection* – permite obter informação sobre os tipos em runtime
  - Assim, não necessita de funções especiais de marshalling e unmarshalling

Extensible markup language (XML)

- XML permite descrever estruturas de dados complexas
  - Tags usadas para descrever a estrutura dos dados
  - Permite associar pares atributo/valor com a estrutura lógica
- XML é extensível
  - Novas tags definidas quando necessário
- Num documento XML toda a informação é textual
  - Podem-se codificar valores binários, por exemplo, em base64
- No contexto dos sistemas de RPC/RMI, o XML pode ser usado para:
  - Codificar parâmetros em sistemas de RPC
  - Codificar invocações (SOAP)
  - Etc.

```
<person id="123456789">
    <name>Smith</name>
    <place>London</place>
    <year>1934</year>
    <!-- a comment -->
</person >
```

```
<?xml version="1.0"?>
<methodCall>
    <methodName>inc</methodName>
    <params>
        <param>
            <value><i4>41</i4></value>
        </param>
    </params>
</methodCall>
```

Extensible markup language (XML)

- XML permite descrever estruturas de dados complexas
  - Tags usadas para descrever a estrutura dos dados
  - Codificar invocações (SOAP)
  - Etc.

```
<person id="123456789">
    <name>Smith</name>
    <place>London</place>
```

```
<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/2001/XMLSchema' xmlns:soap='http://
schemas.xmlsoap.org/soap/ envelope/' xmlns:soapenc='http://
schemas.xmlsoap.org/soap/encoding/' soap:encodingStyle='http://
schemas.xmlsoap.org/soap/encoding/'>
<soap:Body>
        <n:sayHello xmlns:n='urn:examples:helloservice'>
                <firstName xsi:type='xsd:string'>World</firstName>
        </n:sayHello>
</soap:Body>
</soap:Envelope>
```

```
        </param>
    </params>
</methodCall>
```

XML Schema / XML namespaces

- Um XML namespace permite criar espaço de nomes para os nomes dos elementos e atributos usados nos documentos XML
- Um XML schema define os elementos e atributos que podem aparecer num documento XML

```
<xsd:schema xmlns:xsd = URL of XML schema definitions >
    <xsd:element name= "person" type ="personType" />
    <xsd:complexType name="personType">
        <xsd:sequence>
           <xsd:element name="name" type="xs:string"/>
           <xsd:element name="place" type="xs:string"/>
           <xsd:element name="year" type="xs:positiveInteger"/>
        </xsd:sequence>
    <xsd:attribute name= "id" type = "xs:positiveInteger"/>
    </xsd:complexType>
</xsd:schema>
```

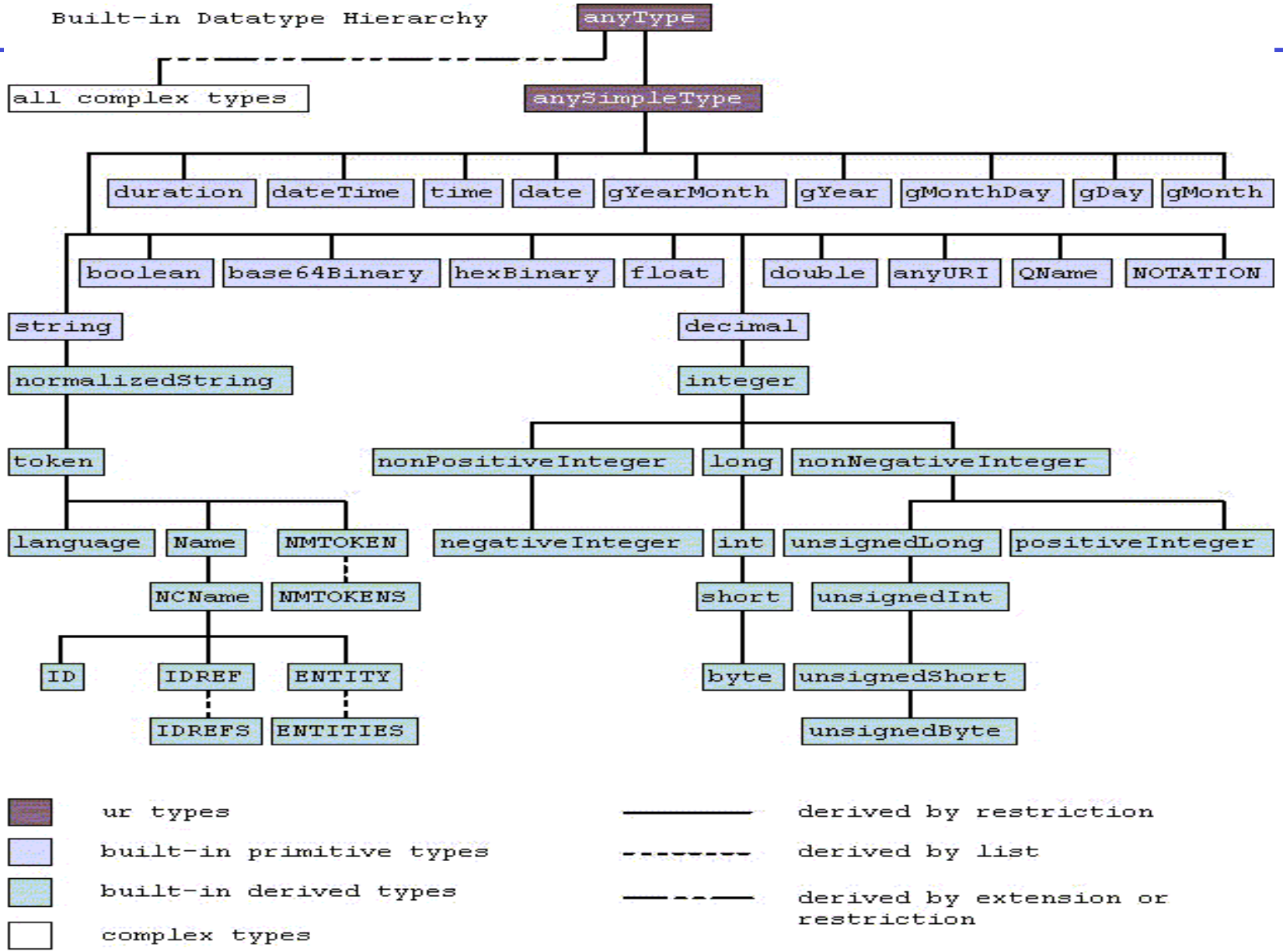
XML Schema / XML namespaces

- Um XML namespace permite criar espaço de nomes para os nomes dos elementos e atributos usados nos documen
- Um XML schema define os elementos e atri num documento XML

```
<person id="123456789">
    <name>Smith</name>
    <place>London</place>
    <year>1934</year>
</person >
```

```
<xsd:schema xmlns:xsd = URL of XML schema definitions >
    <xsd:element name= "person" type ="personType" />
    <xsd:complexType name="personType">
        <xsd:sequence>
            <xsd:element name="name" type="xs:string"/>
            <xsd:element name="place" type="xs:string"/>
            <xsd:element name="year" type="xs:positiveInteger"/>
        </xsd:sequence>
    <xsd:attribute name= "id" type = "xs:positiveInteger"/>
    </xsd:complexType>
</xsd:schema>
```

Tipos XML

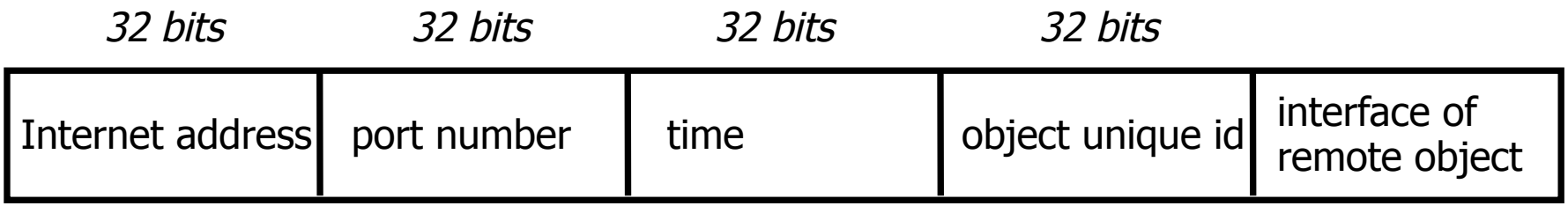

Built-in Datatype Hierarchy
anyType
all complex types
anySimpleType
duration
dateTime
time
date
gYearMonth
gYear
gMonthDay
gDay
gMonth
boolean
base64Binary
hexBinary
float
double
anyURI
QName
NOTATION
string
decimal
normalizedString
integer
token
nonPositiveInteger
long
nonNegativeInteger
language
Name
NMTOKEN
negativeInteger
int
unsignedLong
positiveInteger
NCName
NMTOKENS
short
unsignedInt
ID
IDREF
ENTITY
byte
unsignedShort
IDREFS
ENTITIES
unsignedByte
ur types
built-in primitive types
built-in derived types
complex types
derived by restriction
derived by list
derived by extension or restriction

Representações dos dados: classificação

- Conteúdo da representação
  - Formato binário – Corba, Java
  - Formato de texto – XML
- Integração com linguagem
  - Independente – Corba, XML
  - Integrado – Java
- Informação de tipos
  - Incluída – Java, XML
  - Não incluída – Corba

Passagem de objectos remotos em parâmetro

- Nos sistemas de RMI é, em geral, possível passar (referências para) objectos remotos em parâmetro (ou como resultado de uma operação)
- Em Java RMI pode-se enviar uma referência para um objecto remoto:
  - Passando como parâmetro/resultado uma referência remota – neste caso, uma cópia da referência remota é enviada
  - Passando como parâmetro/resultado o objecto remoto – neste caso, uma referência para o objecto remoto é enviada (e não o próprio objecto) – passagem por referência
- Em CORBA e outros sistemas semelhantes, os objectos remotos são referenciados através de referências para objectos (únicas no tempo e no espaço) como por exemplo a seguinte (não é CORBA):



Passagem de objectos remotos em parâmetro

- Nos sistemas de RMI é, em geral, possível passar (referências para) objectos remotos em parâmetro (ou como resultado de uma operação)

Porquê *time+object unique id*?

Referências únicas no tempo e no espaço, para garantir que o acesso a um objecto que foi apagado dá um erro e não resulta no acesso a outro objecto

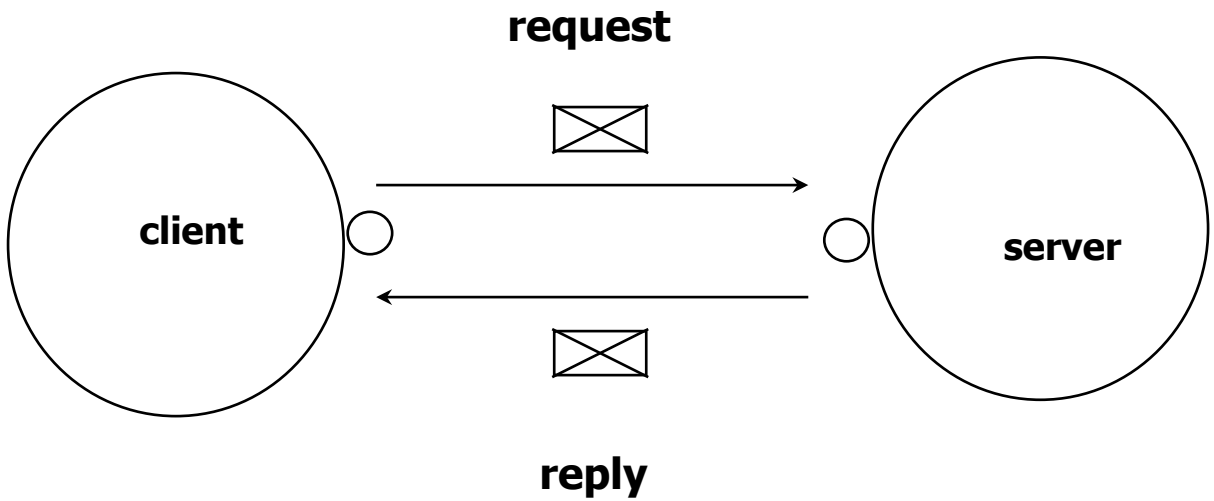
Com esta representação seria fácil mudar a localização do objecto?

Não. Para tal, a referência remota não deve incluir directamente a localização do objecto.

- Em CORBA e outros sistemas semelhantes, os objectos remotos são referenciados através de referências para objectos (únicas no tempo e no espaço) como por exemplo a seguinte (não é CORBA):

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object unique id	interface of remote object

Ligação do cliente ao servidor (*binding*)



- Para poder invocar o servidor, o cliente tem de obter uma referência para o servidor
  - Nos sistemas de RPC, referência corresponde ao communication end-point do servidor – endereço IP + porta + ... (número único da interface)
  - Nos sistemas de RMI, referência remota corresponde geralmente a um proxy com o mesmo interface do servidor (que internamente inclui informação de localização do servidor)
- Como obter essa referência?

Como obter referência para o servidor?

- Por configuração directa
  - Ex.: Web services, .NET remoting
- Servidor de nomes regista associação entre nome e referência remota
  - Ex.: Java RMI, Corba
- Servidor de nomes e directório regista informação sobre servidores
  - Ex. Universal Directory and Discovery Service – UDDI (web services)
  - Além de permitir obter servidor dado o nome, permite procurar servidor pelos seus atributos
- Cliente procura servidor usando multicast/broadcast
  - Alguns sistemas de objectos distribuídos usavam esta aproximação

Por configuração directa

- No código do cliente, para obter uma referência para o servidor indica-se explicitamente a sua localização

.NET remoting

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

Web services

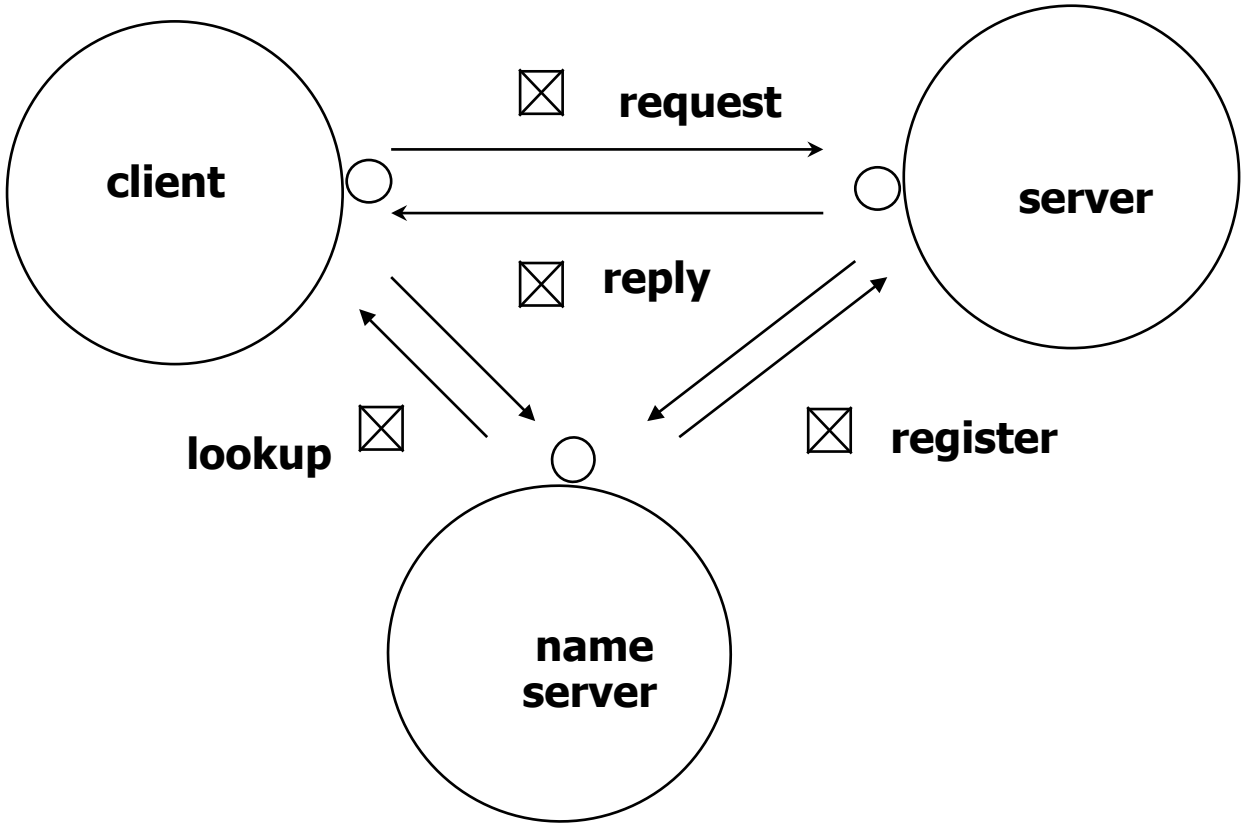
```
NetServerImplWSService service = new NetServerImplWSService(
            new java.net.URL("http://host/xpto"));
```

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

Web services

```
NetServerImplWSService service = new NetServerImplWSService();
```

Usar um serviço (*Binding, Naming or Trading*)



A interface da *Registry* Java RMI

- **void rebind (String name, Remote obj)**
  - This method is used by a server to register the identifier of a remote object by name.
- **void bind (String name, Remote obj)**
  - This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.
- **void unbind (String name, Remote obj)**
  - This method removes a binding.
- **Remote lookup(String name)**
  - This method is used by clients to look up a remote object by name. A remote object reference is returned. The code of the remote reference may be downloaded, if necessary. It encodes the protocol to be used.
- **String [] list()**
  - This method returns an array of Strings containing the names bound in the registry.

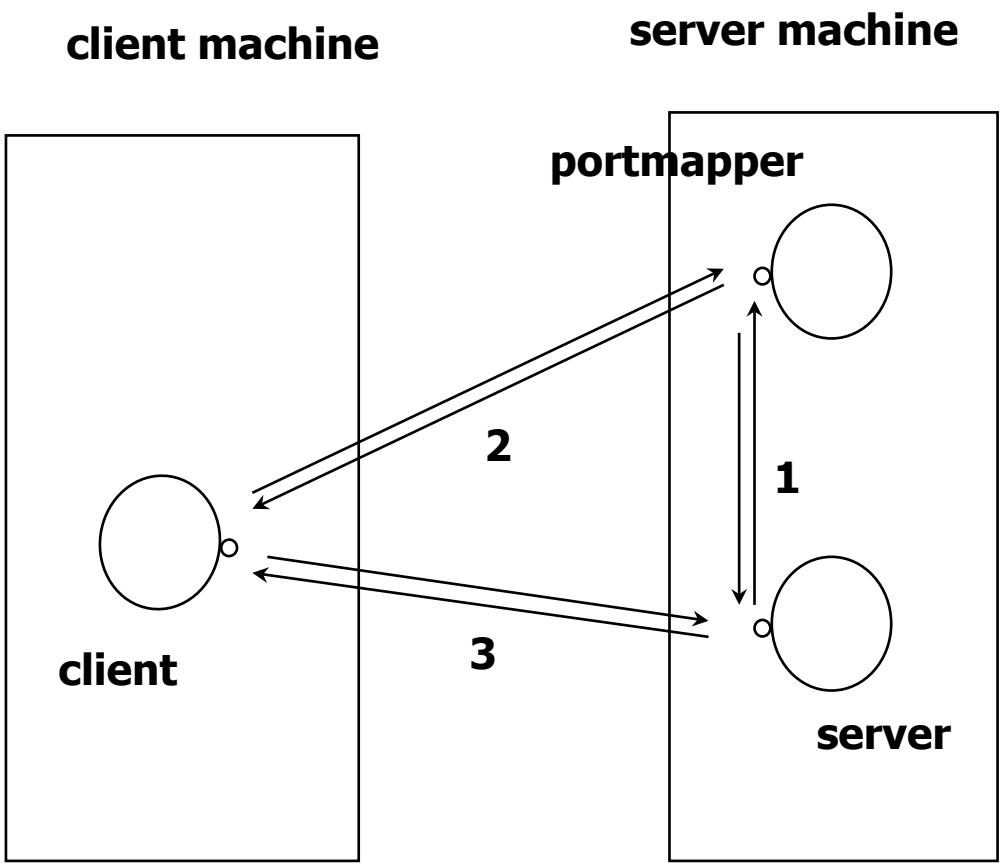
Problemas quando se usa um servidor de nomes?

- Como encontrar o name server ?
  - Nome do objecto indica máquina em que está o name server
  - Name server descoberto por multicast/broadcast

- O name server pode ser único ?
  - O sistema CORBA define um serviço de nomes em que o espaço de nomes é único e global. A implementação depende do sistema.

- Problemas de segurança – como evitar que haja serviços / objectos impostores ou que atacantes impeçam o acesso ao serviço?

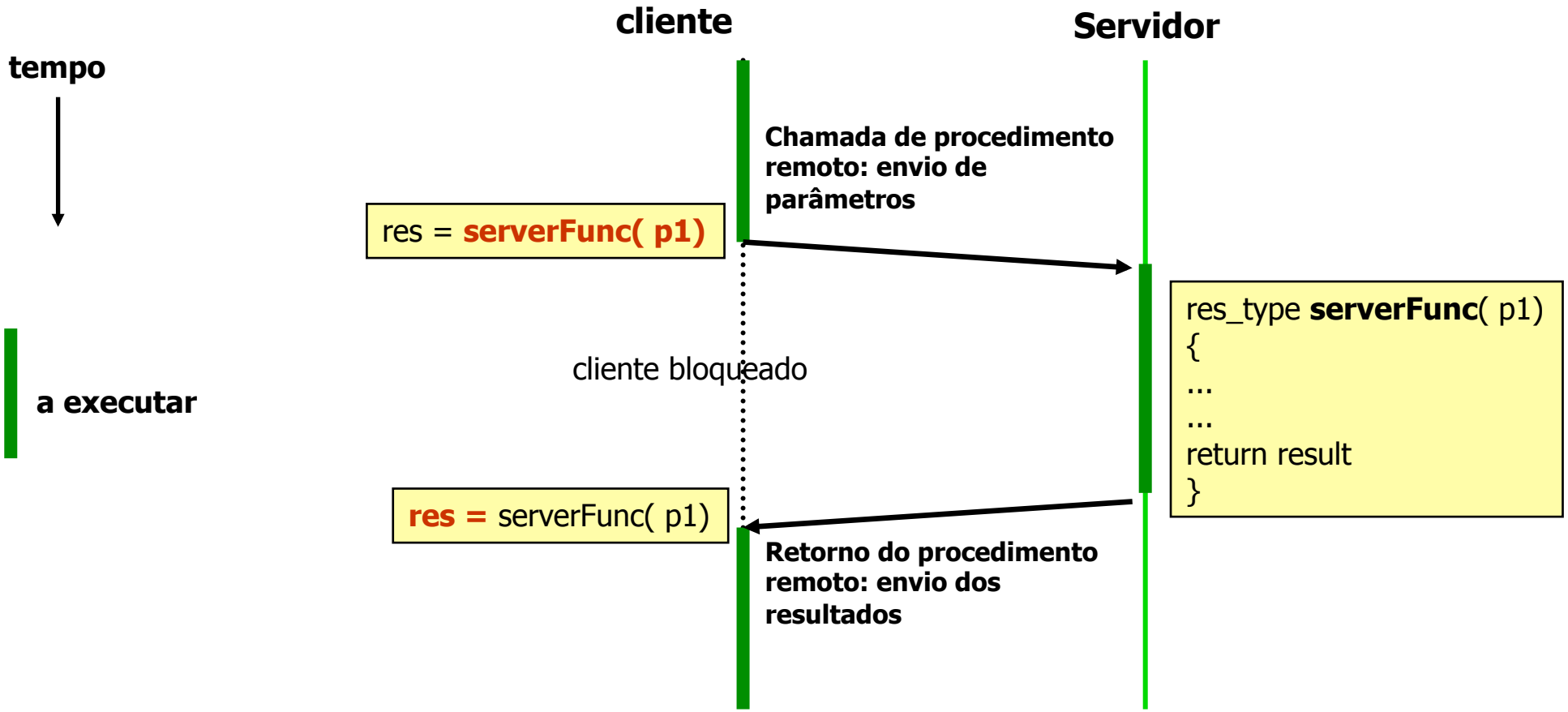
Solução pragmática do Java RMI



- Em cada máquina existe um RMI registry. O cliente tem de saber em que máquina está o serviço/objecto remoto em que está interessado.
  - Em cada máquina, só pode estar associado uma instância de uma interface/classe a cada nome
  - Pode existir mais do que uma instância da mesma interface/classe associados a nomes diferentes

- Porque é que esta solução diminui os problemas de segurança?

Modelo de falhas e semântica da invocação remota (RPC/RMI ou qualquer interacção do tipo pedido/resposta)



- Admitindo que o canal não introduz falhas arbitrárias, podemos ter:
  - **Canal**: falhas de omissão e temporização
  - **Cliente e servidor**: crash-failures e falhas de temporização

Anomalias possíveis durante uma invocação remota

- Anomalias a considerar:
  - a mensagem com o pedido pode perder-se
  - a mensagem com a resposta pode perder-se
  - o servidor está muito lento e aparentemente não responde
  - o servidor falha e não responde
    - o servidor falha e recupera mais tarde (quando ?)
  - o cliente falha e recupera

- Algumas destas situações podem ser facilmente detectáveis, outras não.

Protocolo connection-less vs. connection-oriented

- Motivações para o uso do UDP:
  - Estabelecer uma conexão tem um peso que se pode revelar demasiado pesado (para pedidos pontuais e de pequenas dimensões)
  - Resposta à invocação remota funciona como ACK (não é necessário suportar peso dos mecanismos de fiabilidade incluídos no TCP)

- Motivações para o uso de TCP (ou HTTP):
  - Dimensão dos parâmetros/resultados não influencia complexidade da implementação
  - No caso de usar UDP pode ser necessário enviar um pedido/resposta em mais do que uma mensagem

Protocolo *Connection-oriented*

- **Hipótese**: o cliente usa o protocolo TCP para contactar o servidor e enquanto não receber resposta a um pedido, ou não decidir abandonar um pedido, não avança com outro pedido
  - Situação semelhante quando se usa protocolo HTTP

- 1) O cliente fez um pedido e recebeu a resposta:
  - O cliente tem a certeza que o procedimento executou (uma e uma só vez)
- 2) O cliente fez o pedido e não recebeu resposta nenhuma até um certo *timeout*. Decidiu então abandonar o pedido
  - Que se passou ? Não se sabe se a operação foi executada ou não.
- 3) O cliente fez o pedido e recebeu uma notificação de que a conexão foi quebrada (por falha na comunicação ou por *crash* no servidor)
  - Que se passou ? Não se sabe se a operação foi executada ou não.

Protocolo *connection-less* não fiável (e.g. UDP)

- Hipótese: o cliente usa o protocolo UDP para contactar o servidor e enquanto não receber resposta a um pedido, ou não decidir abandonar um pedido, não avança com outro pedido

- 1) O cliente enviou uma só vez o pedido e recebeu a resposta
  - O cliente tem a certeza que o procedimento se executou (uma e uma só vez… assumindo que não existe duplicação de pacotes)
- 2) O cliente fez o pedido e não recebeu resposta nenhuma até um certo timeout
  - Que se passou ? Não se sabe se a operação foi executada ou não.

Introdução de re-emissões

- Nas situações em que o cliente não recebe resposta até um dado timeout poder-se-ia introduzir um mecanismo de re-emissão do pedido.
  - No caso *connection-less*, reenviando a mensagem
  - No caso *connection-oriented*, se a conexão se quebrou, criando uma nova conexão

- Quantas vezes se re-emite ?

Em caso de anomalia, após retransmissão, o que se passou?

- Caso o cliente não receba resposta ao seu pedido até um timeout após, pelo menos, uma retransmissão, e deseje abandonar, podem ter acontecido duas situações:
  - A anomalia ocorreu antes de executar a operação em todas as retransmissões
  - A anomalia ocorreu após a execução da operação em uma ou mais retransmissões
- Sabe-se que: a operação foi executada zero, uma ou mais vezes.

- Caso o cliente receba resposta ao seu pedido após, pelo menos, uma retransmissão, podem ter acontecido duas situações:
  - Na tentativa de transmissão anterior, a anomalia ocorreu após a execução da operação
  - Na tentativa de transmissão anterior, a anomalia ocorreu antes de executar a operação
- Sabe-se que: a operação foi executada uma vez ou mais vezes.

Filtragem de duplicados

- Problema: como evitar que um pedido seja executado mais do que uma vez no servidor?
- O cliente numera com um número de sequência único cada pedido que faz. Re-emite enquanto não receber resposta até um número limite de vezes.
- O servidor quando recebe um pedido novo, executa a operação e guarda numa "cache" a resposta dada. Caso receba uma re-emissão devolve o resultado previamente calculado ("cached") pois a resposta pode ter-se perdido.
  - Problemas?
  - Para tolerar crashs e recuperações do servidor, a cache com resultados deve ser mantida em memória estável

Filtragem de duplicados (cont.)

- Problema: quando é que o servidor pode remover a informação sobre resultados de pedidos antigos?
- O servidor pode remover informação quando souber que já não vai ser necessária:
  - Cliente pode informar o servidor que recebeu o resultado (ack)
  - Cliente pode iniciar um novo pedido (equivalente a já ter recebido resultado do anterior)
- Caso não inicie um novo pedido e o ack da resposta se perder, o servidor pode anular a informação ao fim de um tempo alargado.

Semântica da invocação remota

- **May be (talvez)**: método pode ter sido executado uma vez ou nenhuma vez
  - usa-se quando não se esperam resultados; não tem garantias. O interesse é muito limitado.

  - Protocolo?
  - Implementado por protocolo de envio simples de mensagem (sem resposta nem retransmissões)

Semântica da invocação remota (cont.)

- **At least once (uma ou mais vezes ou "pelo menos uma vez"):**
  - 1. se cliente recebeu a resposta, o procedimento foi executado uma ou mais vezes
  - 2. se o cliente não recebeu a resposta, o procedimento foi executado zero ou mais vezes

  - Protocolo ?
  - Implementado por protocolo com re-emissões e sem filtragem de duplicados

Semântica da invocação remota (cont.)

- **At most once (zero ou uma vez ou "no máximo uma vez")**:
  - 1. Se cliente recebeu a resposta, o procedimento foi executado uma só vez
  - 2. Se o cliente não recebeu a resposta, o procedimento foi executado zero ou uma vezes

  - Protocolo ?
  - Protocolo sem re-emissões ou protocolo com re-emissões e filtragem de duplicados

Semântica da invocação remota (cont.)

- **Exactly once (exactamente uma vez)**:
  - É necessário registar em memória estável o estado do cliente e servidor para:
    - Quando o cliente recupera de uma falha, deve retomar a execução da invocação
    - Quando o servidor recupera de uma falha, deve recuperar o seu estado (incluindo informação sobre as operações executadas)
  - Nota: a indisponibilidade do serviço durante a falha não é mascarada
  - Protocolo ?
  - Protocolo com re-emissões, filtragem de duplicados e manutençao do estado em memória estável

Protocolos de invocação remota (resumo)

- Com transporte connection-oriented (TCP)
  - Request / Reply protocol (RR): semântica?
    - Implementa facilmente a política *no máximo uma vez*. Como?

- Com transporte connectionless (UDP)
  - Request protocol (R): semântica?
    - Implementa a semântica *maybe*
  - Request / Reply protocol (RR): semântica?
    - Implementa facilmente semântica *pelo menos uma vez*. Como?
  - Request / Reply / Acknowledge protocol (RRA): semântica?
    - usado para melhorar RR no caso de de se estar a implementar a semântica no máximo uma vez. Porquê?

Operações idempotentes

- Em geral, a execução repetida devido às re-emissões conduz a um estado errado do servidor. Nestes casos a semântica "pelo menos uma vez" é inadequada.
- Uma operação é **idempotente** se a sua execução repetida não altera o efeito produzido (deixando o servidor no mesmo estado ou num estado aplicacionalmente aceitável como equivalente e produzindo o mesmo resultado).
- Exemplos de operações idempotentes:
  - em geral todas as operações que não mudam o estado
  - reescrever os primeiros 512 bytes de um ficheiro se se ignorar o problema da concorrência de acessos ao ficheiro
- Exemplos de operações não idempotentes:
  - acrescentar 512 bytes a um ficheiro
  - transferir dinheiro entre contas
  - As operações idempotentes podem ser usadas com um protocolo de invocação remota que faça re-emissões sem filtrar duplicados.

Operações idempo

Nota: Podem existir operações que deixem o servidor no mesmo estado e não sejam idempotentes.
Exemplo ???
Exemplo: uma operação de criação de um ficheiro que devolva como resultado um booleano que indique se o ficheiro foi criado ou não (caso já existisse).

- Em geral, a execução rep
do servidor. Nestes casos
- Uma operação é **idempotente** se a sua execução repetida não altera o efeito produzido (deixando o servidor no mesmo estado ou num estado aplicacionalmente aceitável como equivalente e produzindo o mesmo resultado).
- Exemplos de operações idempotentes:
  - em geral todas as operações que não mudam o estado
  - reescrever os primeiros 512 bytes de um ficheiro se se ignorar o problema da concorrência de acessos ao ficheiro
- Exemplos de operações não idempotentes:
  - acrescentar 512 bytes a um ficheiro
  - transferir dinheiro entre contas
  - As operações idempotentes podem ser usadas com um protocolo de invocação remota que faça re-emissões sem filtrar duplicados.

Que protocolos se usam?

- Protocolos connection-oriented
  - Levam a stubs com implementação mais simples
  - Apropriados quando:
    - Erros nas comunicações podem ser elevados (ex.: redes de longa distância)
    - Cliente inter-actuando continuamente com um dado servidor
    - Parâmetros de grande dimensão
  - Tendência geral para utilização deste tipo de protocolo
    - Java RMI: TCP
    - .NET Remoting: TCP e HTTP
    - Web-services: HTTP
- Protocolos connection-less
  - Usados quando se pretende optimizar as comunicações num ambiente com pouco erros nas comunicações e mensagens de dimensão reduzida
  - Usado em:
    - SunRPC/NFS
    - Sistemas de operação distribuídos com suporte nativo (no núcleo) de protocolos de RPC (Amoeba, Chorus, V-Kernel, ...)

Soluções geralmente adoptadas

- Java RMI
  - "At most once" sobre TCP
- Corba
  - "At most once" geralmente sobre TCP, "maybe"
- .NET Remoting
  - "At most once" sobre TCP, HTTP, pipes
- Web Services
  - "At most once" sobre HTTP (SMTP, etc.)
  - WS-Reliability: suporte para "at least once", "exactly-once"

- Sistemas *antigos*
  - DCE RPC permite utilizar as semânticas "at-most once" e "at-least once" a partir de directivas introduzidas na definição da interface. Estão disponíveis implementações sobre UDP e sobre TCP.
  - SUN/RPC permite utilizar as semânticas "maybe", "at-most once" e "at least once" quer sobre UDP quer sobre TCP. O programador pode também manipular os valores dos timeouts e o número de repetições.

Números únicos e de sequência

- Os sistemas de RPC / RMI necessitam de utilizar números únicos não reutilizáveis ou números de sequência (inconfundíveis num certo período) para diversos efeitos.
  - Números únicos de serviço / interfaces, números únicos de objectos
  - Números únicos de invocações
  - Números únicos de incarnação dos clientes e servidores se necessário

- Ideia para geração de números únicos
  - Concatenar identificador único num dado contexto com um contador

- Números únicos numa máquina
  - Exemplo de prefixo único: pid + tempo
- Números únicos num sistema
  - Exemplo de prefixo único: IP (+tempo: quando IP pode variar ou se quer garantir unicidade após falha e recuperação)

Organização dos servidores

- Activação dos servidores
  - Servidor a executar continuamente
  - Servidor activado quando necessário

- Organização interna
  - Iterativo vs. concorrente

Activação de objectos remotos

- Motivação: num sistema pode haver um número muito elevado de objectos remotos cujo estado se quer que persista durante tempo ilimitado, mas que não estão em uso durante grande parte do tempo
- Solução: activam-se os objectos remotos apenas quando necessário
  - Quando um método é invocado ou quando uma referência remota é obtida
- *Activator*: servidor responsável por:
  - Manter informação sobre os objectos activáveis
  - Activar os objectos remotos quando solicitado por um cliente
  - Manter informação sobre localização dos objectos activados
- Objecto remoto *passivo* (quando não activado)
  - Código
  - Estado do objecto *marshalled*
- Referência remota mantém informação necessária para solicitar a activação do objecto

Organização dos servidores: activação

- Quando é necessário activar um servidor (objecto remoto), pode-se criar:
  - Um servidor para atender todos os clientes
    - Aproximação mais comum
  - Um servidor para atender cada cliente
    - E.g. .NET remoting: servidor *SingleCall*

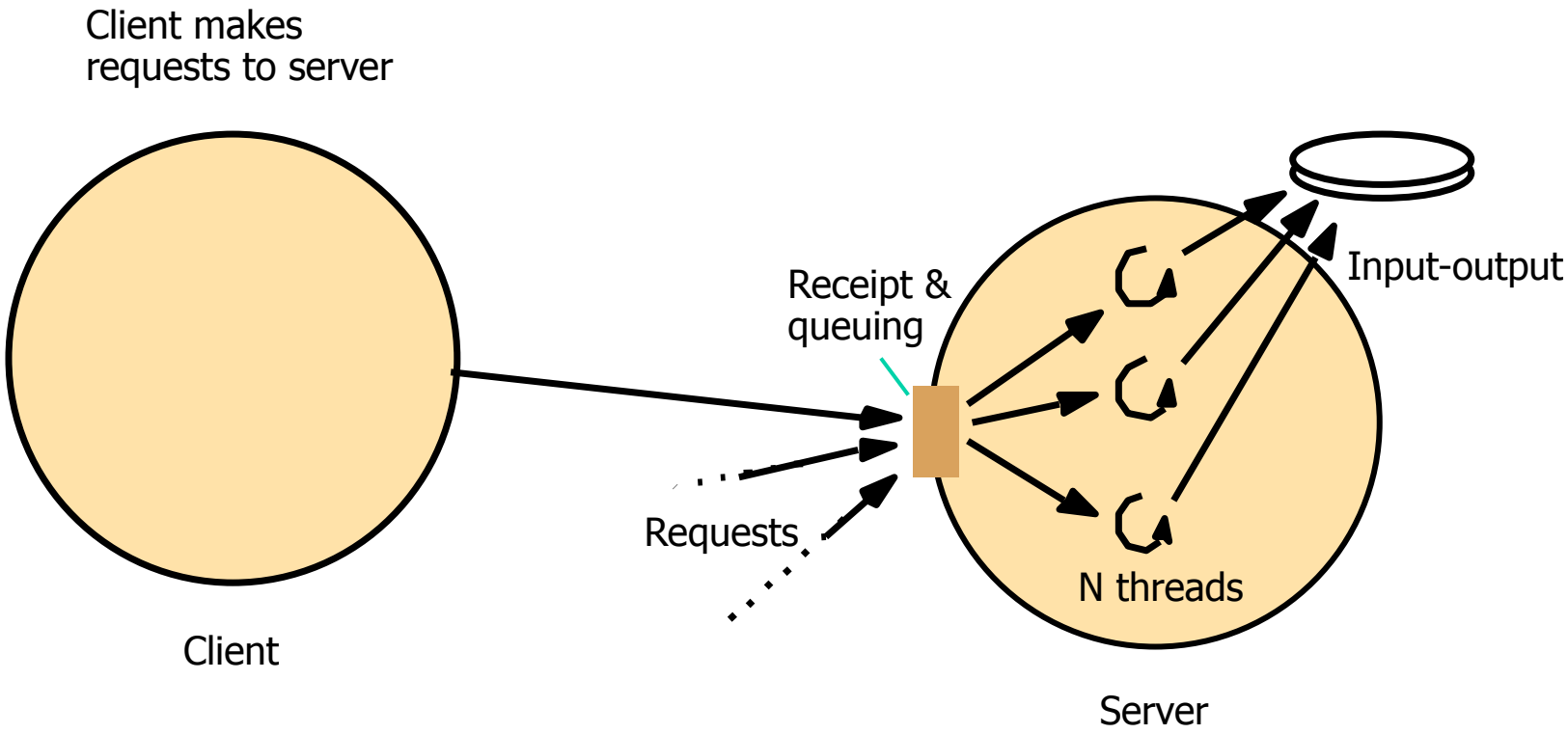
Organização dos servidores: *threads*

- Servidor iterativo: o servidor executa os pedidos de forma sequencial, executando um de cada vez
  - Modelo simples e claro

- Para alguns tipos de serviços, esta aproximação pode ter um desempenho inadequado
  - Exemplos: servidores de bases de dados, de ficheiros, etc. Porquê?
  - Exemplo: serviços que chamam outros serviços em ambos os sentidos (A→B e B →A). Porquê?

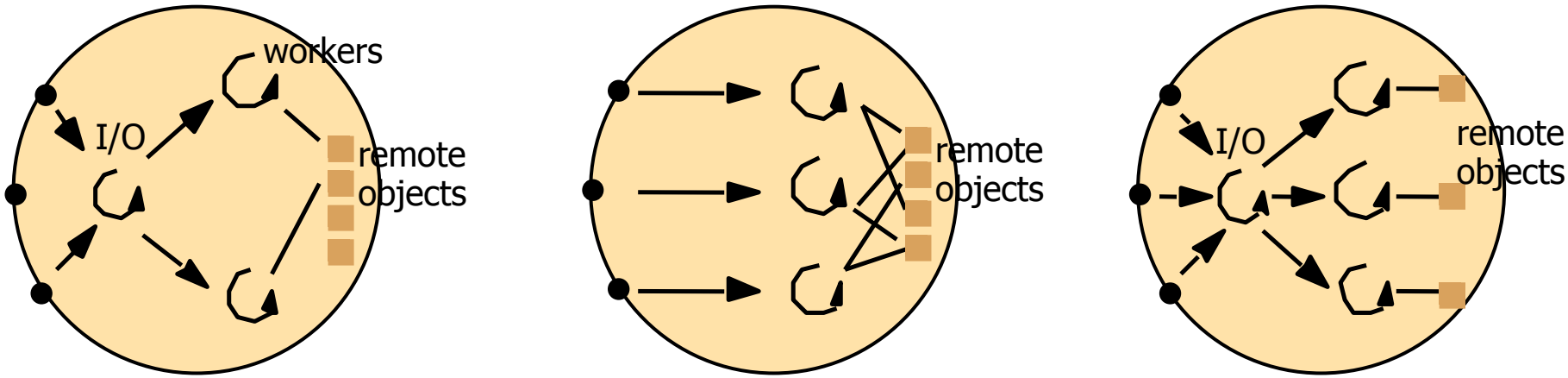
- Em geral, quando a execução de uma operação remota pode ser longa é interessante introduzir concorrência no servidor. Porquê?

Utilização de threads num servidor



- A ter em atenção:
  - Possíveis problemas de concorrência: necessidade de sincronizar execução dos vários *threads*.
    - *Distributed deadlock*: quando um servidor A chama outro servidor B e o servidor B chama o servidor A
  - Como é que os threads se organizam e se relacionam com os pedidos?

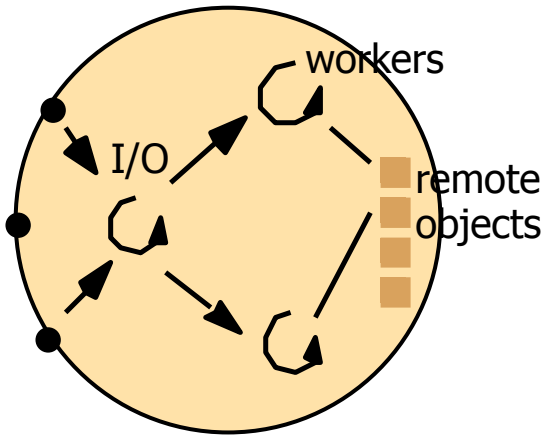
Hipóteses de relação invocações / *threads*



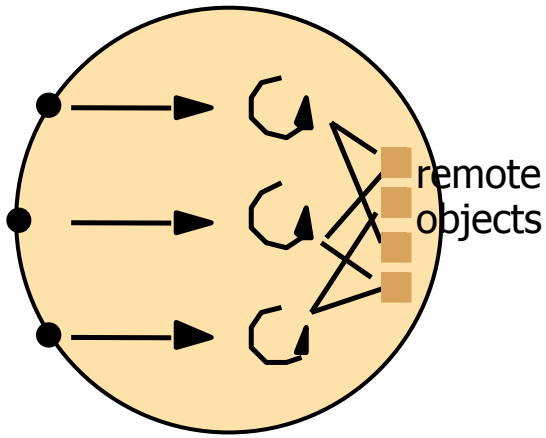
a. Thread-per-request b. Thread-per-connection c. Thread-per-object

- *Thread-per-request*: cada invocação é executada por um *thread*
- *Thread-per-connection*: as invocações de uma conexão são executadas todas pelo mesmo *thread*
- *Thread-per-object*: as invocações para um objecto são executadas todas pelo mesmo *thread*

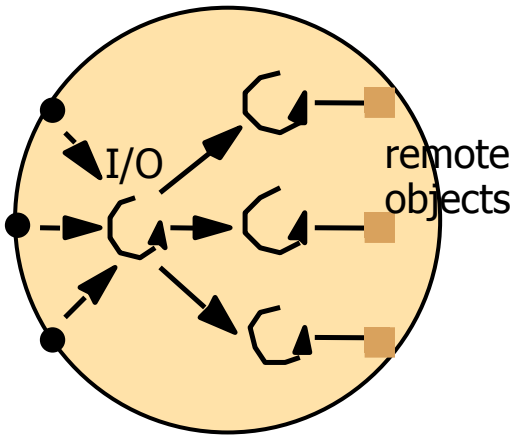
Hipóteses de relação invocações / *threads*



a. Thread-per-request



b. Thread-per-connection



c. Thread-per-object

- *Thread-per-request*: cada
- *Thread-per-connection*: as
  pelo mesmo *thread*
- *Thread-per-object*: as invo
  mesmo *thread*

Quais os problemas de concorrência entre os threads nos vários modelos?

Nos modelos *thread-per-request* e *thread-per-connection* podem existir vários *threads* a aceder aos mesmos dados.

No modelo *thread-per-object* pode existir um problema de deadlock distribuído se a execução de um método puder levar à invocação local de um método noutro objecto

Organização dos *threads*

- Nos sistemas que usam múltiplos *threads* é comum:
  - Existir um *thread* responsável por distribuir as invocações e existir um conjunto de *threads* responsáveis por executar as invocações, sendo reutilizados em sucessivas invocações
  - *Pools* de *threads*
    - Em cada momento, o sistema mantém informação sobre os *threads* que não estão a processar nenhuma operação, os quais se encontram a *dormir*
    - Quando uma nova invocação é recebida, a informação sobre a mesma é passada para um *thread* da *pool*, o qual fica responsável por processar o pedido
    - No fim de processar o pedido, o *thread* volta à *pool*

Outros aspectos

- Tipo de invocação em RMI
  - Invocação estática – operação a invocar definida em tempo de compilação
    - Ex: server.someOp(param1, param2)
  - Invocação dinâmica – operação a invocar pode ser definida em tempo de execução
    - Ex. RMISystem.invoke(server, "someOp", param1, param2)

- Outros modelos de RPC/RMI
  - RPCs/RMI assíncronos – neste caso, o servidor envia um ACK quando recebe o pedido. O cliente prossegue assim que recebe o ACK.
    - Cliente pode aceder ao resultado assincronamente
    - E.g. Futuros: Future<ResultType> res = server.execOpAsync(someParams);
if(res.isReady()) result = res.getResult();
  - RPCs/RMIs *one-way* – neste caso, o cliente prossegue imediatamente a seguir a enviar o pedido, não esperando pelo ACK

Apresentação de Java RMI

- A estrutura da apresentação é semelhante e utiliza algumas figuras e exemplos de um tutorial sobre Java RMI proferido por Jim Waldo no

  - USENIX COOTS'98 – "The Fourth Conference on Object-Oriented Technologies and Systems (COOTS'98)", Santa Fé, New Mexico, USA, April 1998

Java RMI

- Ao assumir-se uma única linguagem, portável entre diferentes plataformas, é possível explorar algumas vantagens impossíveis noutros contextos, nomeadamente:
  - Uma plataforma de distribuição homogénea mesmo quando as máquinas e sistemas são heterogéneos
  - Um modelo de objectos único e global
  - Um único sistema de tipos (locais e remotos)
  - Objectos em parâmetro, passados de forma polimórfica

  - Carrega dinamicamente código se necessário
  - Baseia-se na segurança do sistema Java

Relevância da linguagem Java para o RMI

- Características particularmente relevantes do Java no contexto da invocação remota:
  - Heterogeneidade:
  - Neutralidade em relação a diferentes arquitecturas hardware: o código é interpretado (byte Codes), através de JVMs (Java Virtual Machines). Todos os tipos de dados são definidos rigorosamente.

  - Acesso a servidores (eventualmente desconhecidos):
  - Ligação dinâmica do código das classes: o código só é carregado quando é necessário e tal operação é realizada dinamicamente. A JVM que carrega dinamicamente as classes, conhece a noção de interface e realiza controlo de tipos em tempo de carregamento. O carregamento de código pode fazer-se remotamente através de URLs.

Carregamento de código e segurança

- Não há apontadores e não é possível escrever código que viole as interfaces das classes. Se um programa (uma JVM) importar o código de uma classe, o processo de compilação e ligação dinâmica de código impede que um objecto aceda ao código de uma classe ou aos dados dos outros objectos.
- O código das classes a carregar dinamicamente pode ser assinado para autenticação e é possível especificar os seus direitos de acesso (criação de *sockets*, acesso a ficheiros, etc.)
- Desta forma, é possível carregar dinamicamente código com segurança. Por exemplo, um cliente pode carregar dinamicamente um *stub*, ou um objecto pode ser passado por valor, sendo copiado de uma JVM para outra. Se necessário, o código da sua classe é carregado dinamicamente pela JVM receptora do objecto.

Os objectos remotos são objectos Java

- Os objectos remotos são objectos Java.
- Os objectos remotos são referenciados via interfaces que herdam da interface *Remote*
- Os objectos *stub* implementam a mesma interface que o objecto remoto
  - pode fazer-se *casting*
  - pode usar-se *instanceof* para testar o tipo das interfaces
- Os problemas de comunicação e outros são comunicados através de excepções

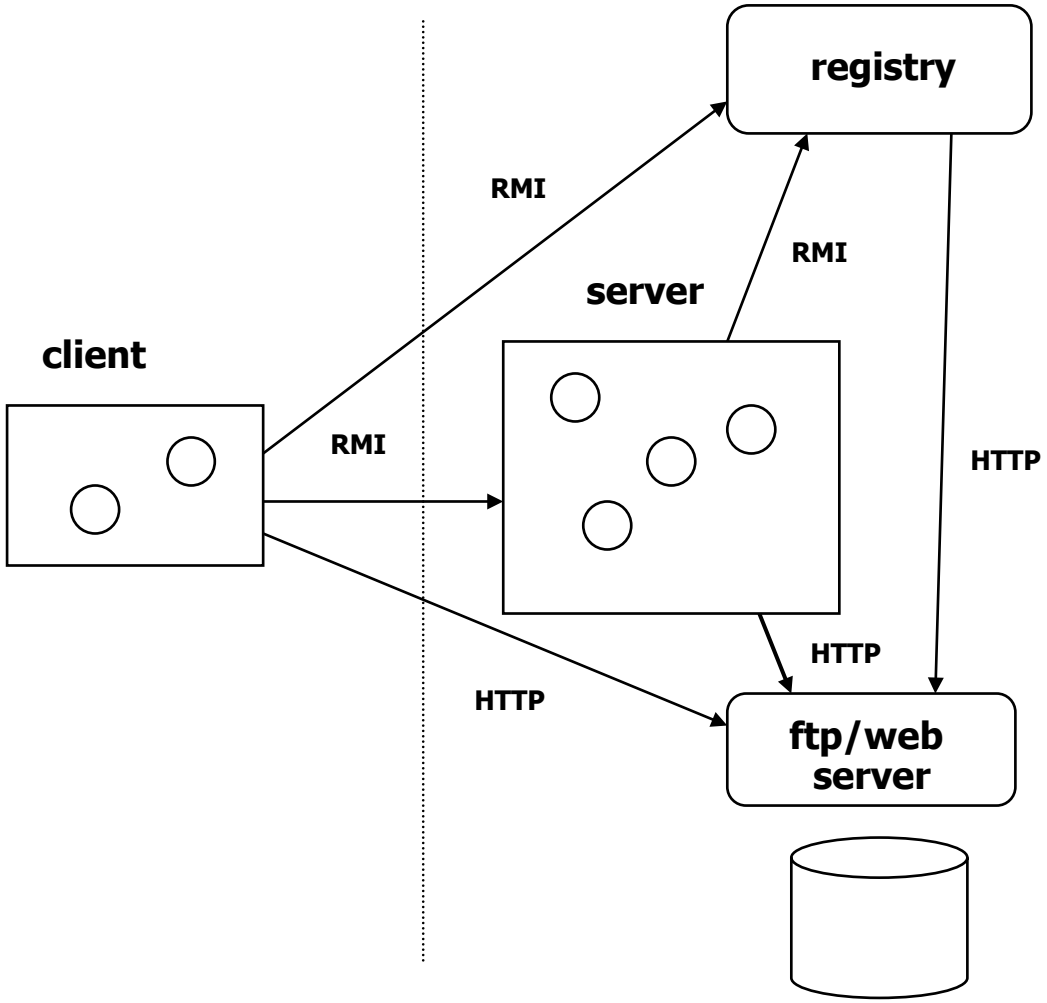
Tipos dos argumentos e do valor de retorno

- Os argumentos e o valor de retorno podem ser de qualquer tipo desde que seja derivado de *Serializable* ou *Externalizable* (or *RemoteObject,* como o *UnicastRemoteObject)*
- Os objectos do tipo *Serializable* e *Externalizable* são copiados através da serialização de objectos
  - os *stubs* (referências remotas) são serializáveis
  - dá o efeito da passagem de um objecto por valor (duplicação do objecto)
- Os objectos do tipo *RemoteObject* passados em parâmetro são substituídos por *stubs*
  - o *stub* tem embebida a referência remota (a serialização é especial)
  - dá o efeito da passagem por referência
- O código das classes é carregado a pedido

Carregamento de classes

- RMI carrega o código das classes se o mesmo não estiver disponível
- Isto aplica-se às classes do objecto remoto, do *stub*, do esqueleto, dos parâmetros e do valor de retorno dos métodos
- O carregamento faz-se remotamente se necessário
- Os URLs das classes ficam anotados nas referências remotas para se poderem localizar as mesmas
- As classes a carregar são sujeitas ao controlo do gestor de segurança instalado

Uma aplicação distribuída em tempo de execução



Passos do desenvolvimento

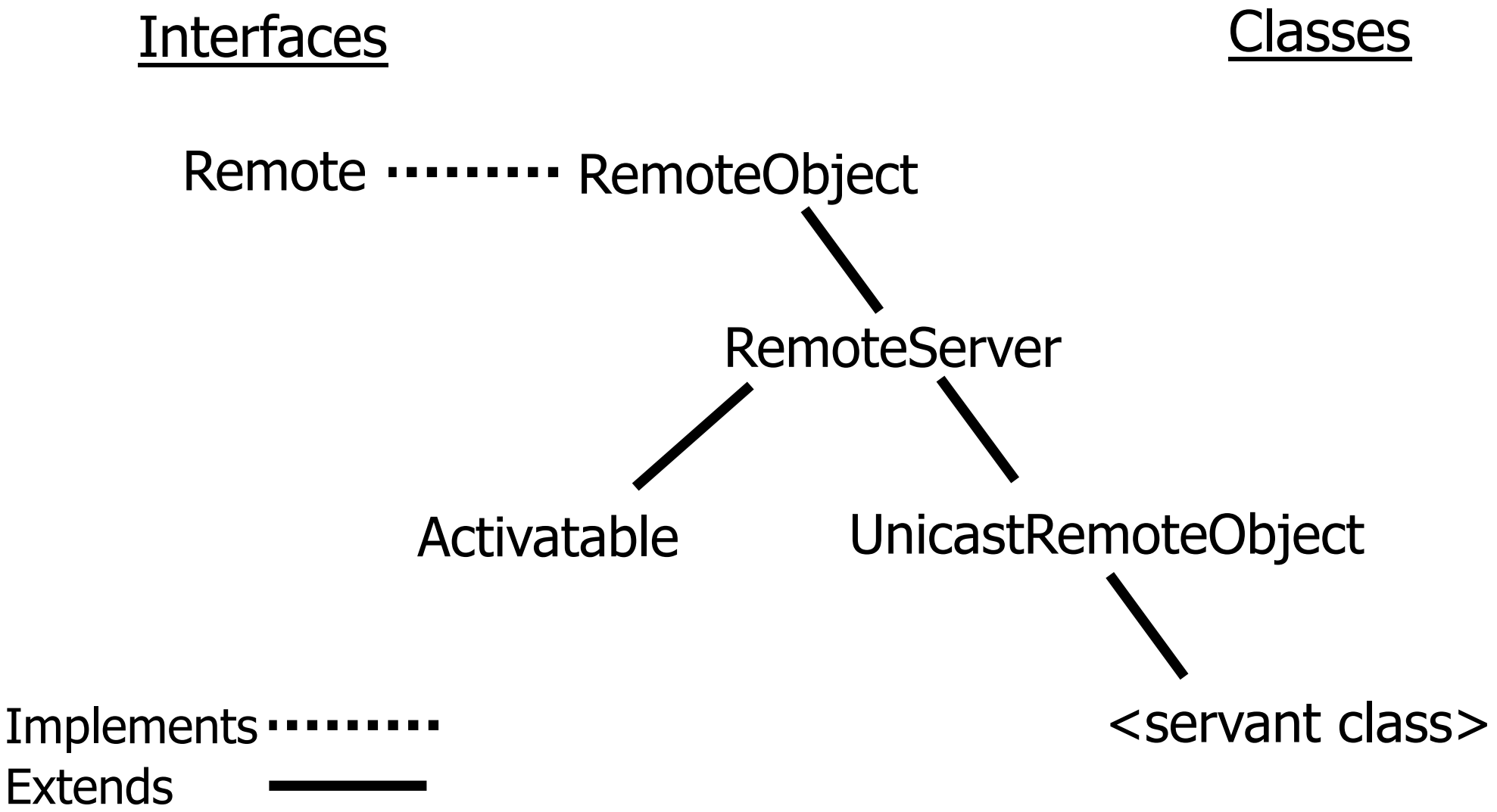
- Definir as interfaces dos objectos remotos
- Implementar os objectos remotos
- Compilar com *rmic* as classes de implementação dos objectos remotos (opcional)
- Tornar o código acessível na rede

Exemplo 1

- A ideia é desenvolver um servidor de anedotas (jokes).
- Primeira passo: criar o interface do servidor

```
public interface JokeServer extends java.rmi.Remote
{
   public String getJoke ()
                  throws java.rmi.RemoteException
}
```

Java RMI interfaces e classes



Joke Server Implementation

```
import java.rmi.*
import java.rmi.server.*
import java.util.Random;

public class JokeServerImpl
   extends UnicastRemoteObject
   implements JokeServer
{
   private String[] jokes;
   private Random randgen
         = new Random ();
```

```
public JokeServerImpl ()
      throws RemoteException
{
   super();
   // initialize joke array
}

public String getJoke ()
      throws RemoteException
{
   int i = random.nextInt();
   i = i % jokes.lenght;
   return jokes[i];
}
```

Joke server Implementation (cont.)

```
public static void main (String [] args) {
   try {
      if (System.getSecurityManager() == null)
                  System.setSecurityManager ( new RMISecurityManager () );

      JokeServerImpl js = new JokeServerImpl() ;
      Naming.rebind ( “Jokes”, js );
      System.out.println (“Joke server ready” );
   } catch (Exception e) {
      System.out.println (“Can’t make a joke: ”  + e.getMessage());
      e.printStackTrace ();
   }
}// main

} // JokeServerImpl
```

Implementação do cliente

- Um cliente terá de localizar o servidor e obter uma referência para o mesmo. Isto pode ser feito usando o *registry service* do RMI
  - Os servidores são designados por URLs (Uniform Resource Locator)
  - É necessário invocar um servidor de nomes através de RMI, passando em parâmetro o URL do servidor e obtendo como resposta uma referência ao servidor
  - De que tipo é a referência obtida (servidor: JokeServerImpl implements JokeServer)?
    A referência remota é um objecto local – stub (do tipo JokeServerImpl_Stub implements JokeServer caso tenha sido usado o *rmic*) que actua como proxy do objecto remoto

- Invocar o método getJoke do objecto remoto invocando o objecto local (stub)
- Imprimir o Joke
- Tratar as eventuais excepções

O cliente

```
class JokeClient {
   static public void main (String[] args ) {
      try {
          JokeServer joker = ( JokeServer )
                              Naming.lookup (“//JokeServerHostName/Jokes” );
          String newJoke = joker.getJoke();
          System.out.println ( newJoke ) ;
      } catch ( Exception e ) {
         System.out.println(“can’t get a joke: ” + e.getMessage());
         e.printStackTrace ();
      }
    } // main

} // JokeClient
```

Exemplo 2: polimorfismo no servidor

- Objectivo: criar um servidor que possa executar cálculos arbitrários

- O cliente envia o objecto capaz de realizar o cálculo para o servidor
  - Podíamos chamar a este objecto um agente (simples)
- O servidor executa o cálculo e retorna o objecto resultado, por cópia ou por referência
  - O sistema carrega dinamicamente no servidor o código do objecto cálculo

Interfaces

- ***Task*** - define uma task arbitrária para ser executada remotamente

```
public interface Task extends Serializable {
   Object run () ;
}
```

- ***Compute*** - define o serviço do servidor de computação

```
import java.rmi.*;
public interface Compute extends Remote {
   Object runTask ( Task t ) throws RemoteException;
}
```

O servidor de computação

```
import java.rmi.*
import java.rmi.server.*

public class ComputeServer extends UnicastRemoteObject implements Compute {
  public ComputeServer () throws RemoteException { /* initializations*/ }

  public Object runTask (Task t) throws RemoteException {
    return t.run () ;
  }
  public static void main (String [] args) {
    // SecurityManager installation
    try {
        ComputeServer cs = new ComputeServer () ;
        Naming.rebind ( “ComputeServer”, cs );
    } catch (Exception e) {
        System.out.println (“Can’t compute” + e.getMessage() );
        e.printStackTrace ();
    }
  }
}
```

Notas sobre a *interface Compute*

- O método *runTask* retorna qualquer objecto do tipo *Object*. Duas alternativas são possíveis.
  - A *Task* passada em parâmetro de *runTask* pode criar um objecto serializável no servidor e retorná-lo por cópia
    - O original será *garbage collected* no servidor quando a task terminar
  - A *Task* passada em parâmetro de *runTask* pode criar um objecto do tipo *Remote* no servidor e retornar uma referência para ele
    - O objecto remoto será *garbage collected* distribuidamente quando essa referência deixar de ser usada
  - Vantagens de cada método?

- Se a task passada em parâmetro de *runTask* criar um objecto que não é nem *Remote* nem serializável, é gerada uma excepção de *marshalling*

- Os tipos primitivos (byte, short, int, ...) têm de ser passados através dos seus equivalentes objectos, eventualmente por coversão automática (box/unbox)

O que o cliente tem que fazer

- Cria a *task* a ser executada remotamente
- Obtém uma referência para o serviço de computação remoto
- Envia a *task* a calcular para o servidor de cálculo
- Obtém o objecto resultado
- Imprime o resultado

Task que calcula Pi

```
public class Pi implements Task {
   private int places;

   public Pi ( int places ) {
      this.places = places;
   }

   public Object run ( ) {
      // compute Pi
      return result ;
   }
}
```

Task que calcula uma função

```
public class F implements Task {
   private F args ....;

   public F ( args .... ) {
      // initializes args
   }

   public Object run () {
      // compute F
      return result ;
   }
}
```

O cliente

```
Compute comp = (Compute ) Naming.Lookup
                        (“//SuperProcessorHost/ComputeServer” );

// create the tasks to be evaluated remotely
Pi pi = new Pi ( 100 );
F f = new F ( args... );

// make the remote caclculations

Object piResult = comp.runTask ( pi );
Object fResults = comp.runTask ( f );

// print results
```

Garbage-collection distribuído

- Numa máquina virtual, o sistema Java usa um mecanismo de *grabage-collection* para remover (apagar) os objectos para os quais já não há referências
  - Por isso, ao contrário do C++, não é necessário fazer *delete* sobre os objectos que já não são necessário
- Entre máquinas virtuais, o sistema Java executa um algoritmo de *garbage-collection* distribuído para remover (apagar) objectos remotos para os quais não existem referências
  - Cada máquina virtual (VM) contabiliza as referências que existem para cada objecto remoto e informa a VM do objecto
    - Quando aparece a primeira referência, a VM do objecto remoto é informada
    - Quando é removida a última referência, a VM do objecto remoto é informada
  - Um objecto remoto pode ser removido (apagado) quando não existem referências em nenhuma VM

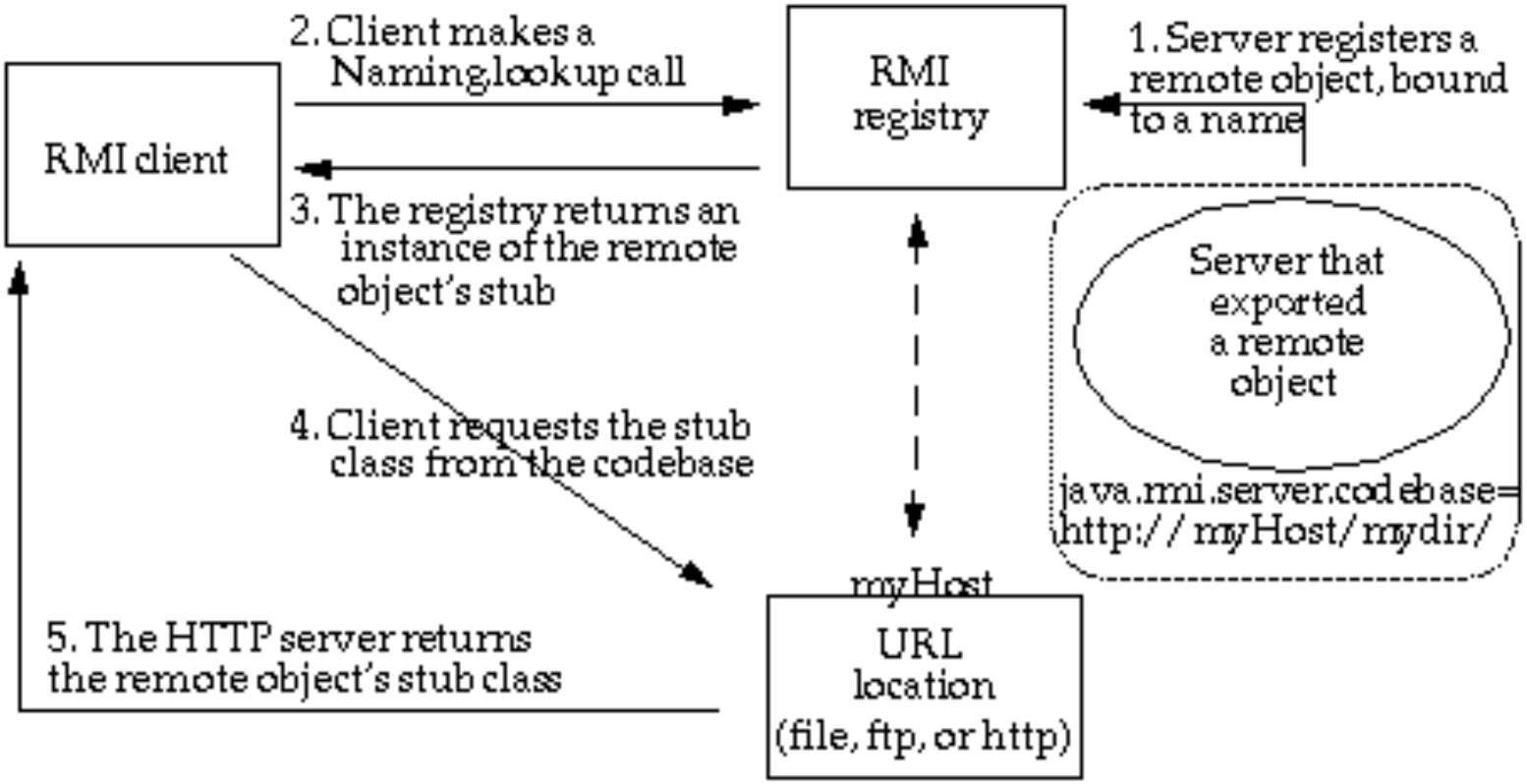
Questão dum exame

- Como sabe, o sistema Java RMI usa um mecanismo de garbage collection distribuído para recolher (remover) os objectos remotos que já não são necessários. Explique o que acontece a um objecto remoto no caso de existir uma falha temporária da rede que impeça a comunicação entre a máquina virtual em que executa um objecto remoto e a máquina virtual do único programa que mantém uma referência para esse objecto remoto.

Segurança do carregamento do código

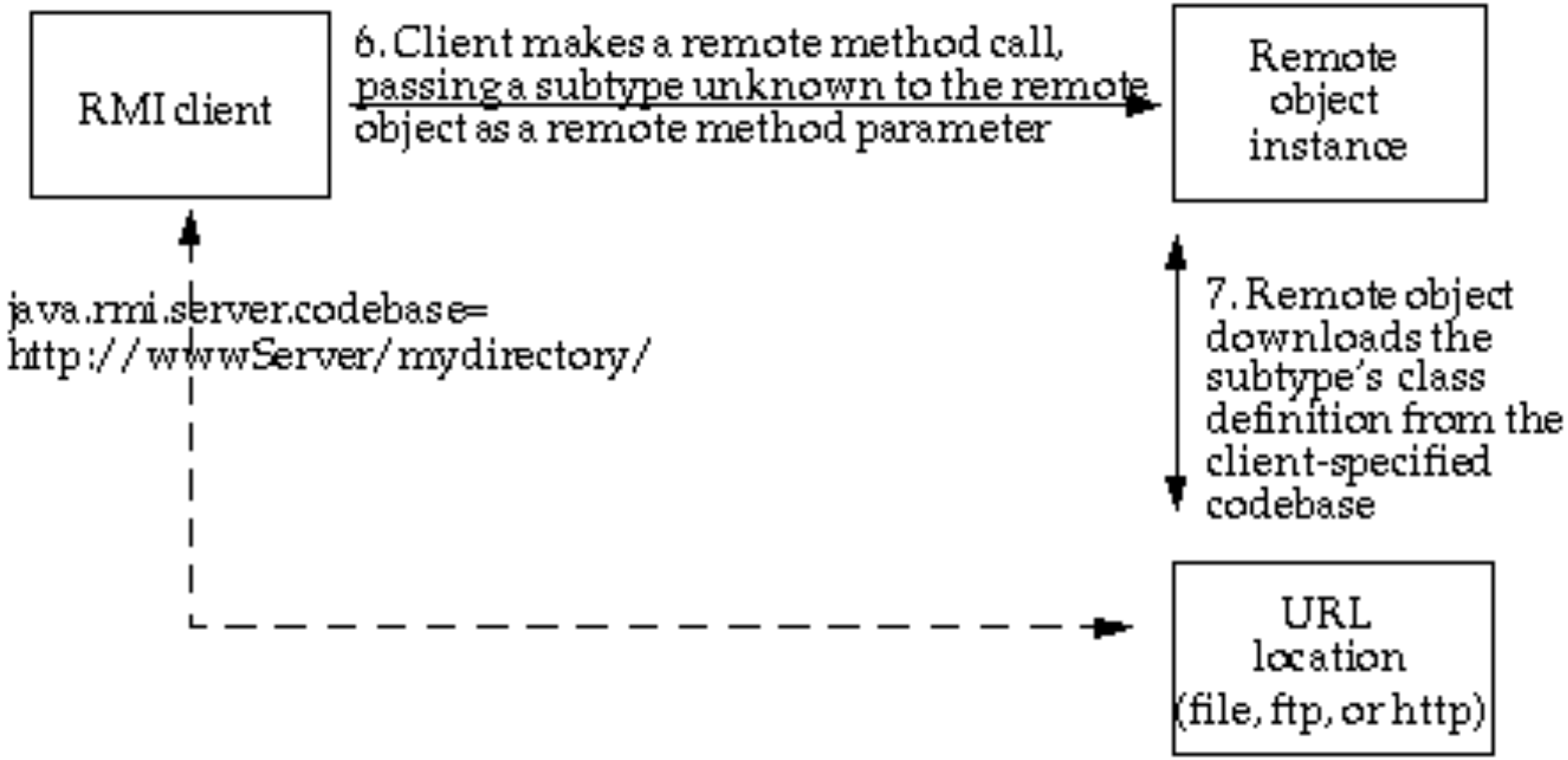
- O RMI usa os mecanismos de segurança da linguagem Java
- Qualquer código a carregar tem de o ser através de um carregador de classes (*class loader*)
- Um gestor de segurança (*security manager*) tem de estar presente para que o carregamento de código seja possível
- As aplicações podem fornecer o seu próprio gestor de segurança
- A omissão de um gestor de segurança não abre buracos de protecção

Carregamento de código no cliente



Os objectos anotam o seu *codebase*, o que permite obter o código das classes remotamente sem configuração adicional.

Carregamento de código no servidor



Exemplo 3: polimorfismo no cliente

- O servidor conterá um objecto "política de notas de despesa aceitáveis" actualizado
- O cliente tem uma interface através da qual são preenchidas as notas de despesa
- Como permitir actualização das *políticas* sem ter de criar um novo cliente e não sobrecarrgando o servidor?
- A política de aceitabilidade das mesmas será carregada dinamicamente no cliente para verificar se a nota de despesa é aceitável
- Os clientes são actualizados dinamicamente conforme necessário se a política de notas de despesa mudar
- O **teste de aceitabilidade pode executar no cliente** e o servidor não é sobrecarregado o que é preferível também para o utilizador

O servidor de notas de despesa

```
import java.rmi.*

public interface ExpenseServer extends Remote
{
   ExpPolicy getExpPolicy ( ) throws RemoteException;

   void submit ( ExpenseList report )
                  throws RemoteException, InvalidReportException ;
}
```

A interface Policy

```
import java.io.Serializable ;

public interface ExpPolicy extends Serializable
{
   void checkEntry ( ExpenseEntry entry )
                        throws InvalidExpenseException ;
}
```

Notas sobre a solução

- A interface para a submissão de notas de despesa utiliza um objecto *ExpPolicy* para saber se um item de despesa é válido (entry)
  - Se uma entry é válida não há problema
  - Senão desencadeia uma excepção *InvalidEntryException*
- A política de aceitação é testada localmente pelo cliente
  - evita sobrecarregar o servidor
  - dá *feedback* imediato ao utilizador
- Novas políticas de aceitação das despesas são carregadas dinamicamente sempre que a interface é carregada, através de um browser por exemplo

ExpPolicy e ExpenseReport

```
ExpenseEntry entry;

// start new expense report and show GUI to user
ExpPolicy currentPolicy = server.getPolicy();
ExpenseList report = new ExpenseList();
while ( notDone ) {
   entry = GUI.getEntry () ;
   try {
      currentPolicy.checkEntry ( entry );
      report.addEntry ( entry );
   } catch (InvalidEntryException e) {
      GUI.errorDisplay ( e.getReason() );
   }
}
server.submit ( report );
```

O código do objecto *ExpPolicy* está instalado no servidor do objecto *ExpenseServer*

Para saber mais

- G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems - Concepts and Design, Addison-Wesley, 4th Edition, 2005
  - RMI/RPCs - capítulo 5.
  - Representação de dados e protocolos - capítulo 4.
  - Web services – capítulo 19.1-19.4
  - Corba – capítulo 20 (interessante para quem queira saber mais).

Sistemas Distribuídos

Capítulo 5

Web services e modelos alternativos de interacção cliente/ servidor na internet

Nota prévia

- A apresentação utiliza algumas das figuras do livro de base do curso

  G. Coulouris, J. Dollimore and T. Kindberg,
  Distributed Systems - Concepts and Design,
  Addison-Wesley, 4th Edition, 2005

- Para saber mais:
  - Web services – capítulo 19.1-19.4
  - REST: http://en.wikipedia.org/wiki/Representational_State_Transfer
  - GWT: http://code.google.com/webtoolkit/

Organização do capítulo

- Web services SOAP
- Web services REST
- Invocação remotas assíncronas
  - Ajax
  - GWT

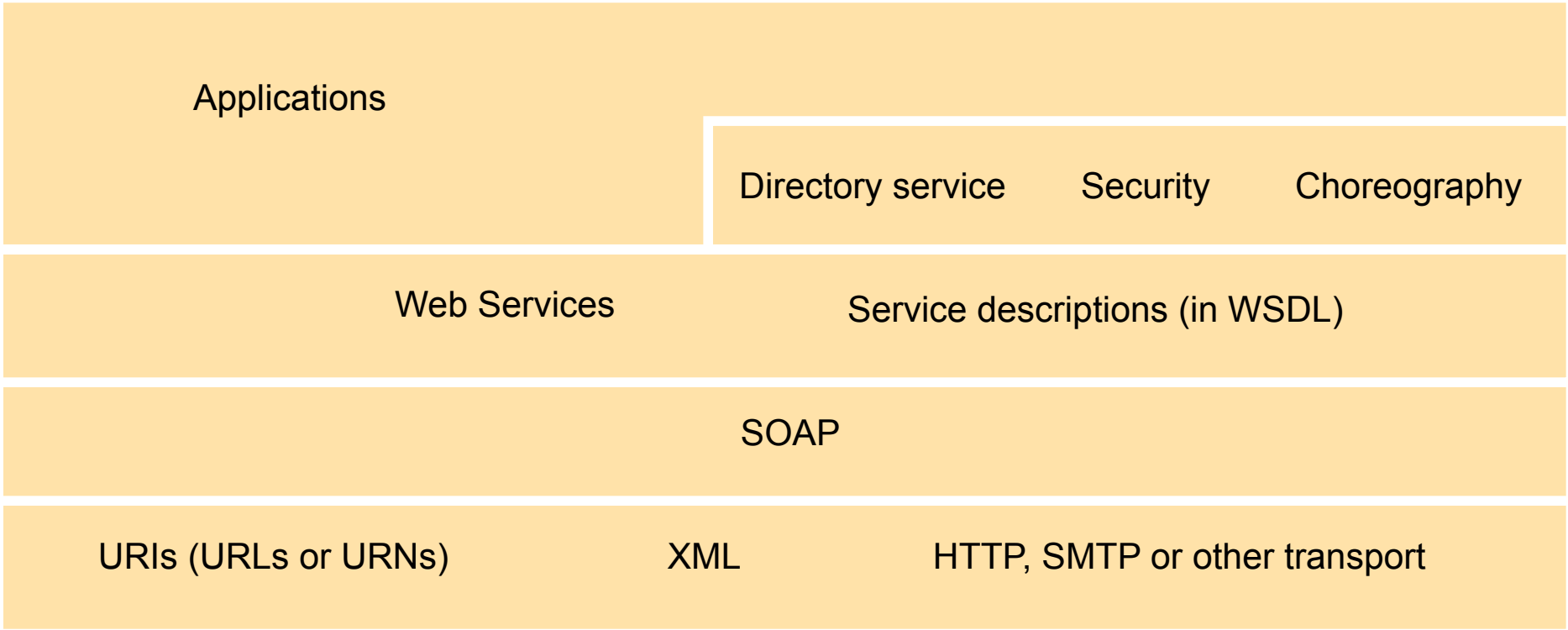
Web Services

- Definição W3C:
  - *A **Web service** is a software system designed to support interoperable machine-to-machine interaction over a network. It has an **interface** described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using **SOAP messages**, typically conveyed using **HTTP** with an **XML serialization** in conjunction with other Web-related standards.*

Web services: motivação

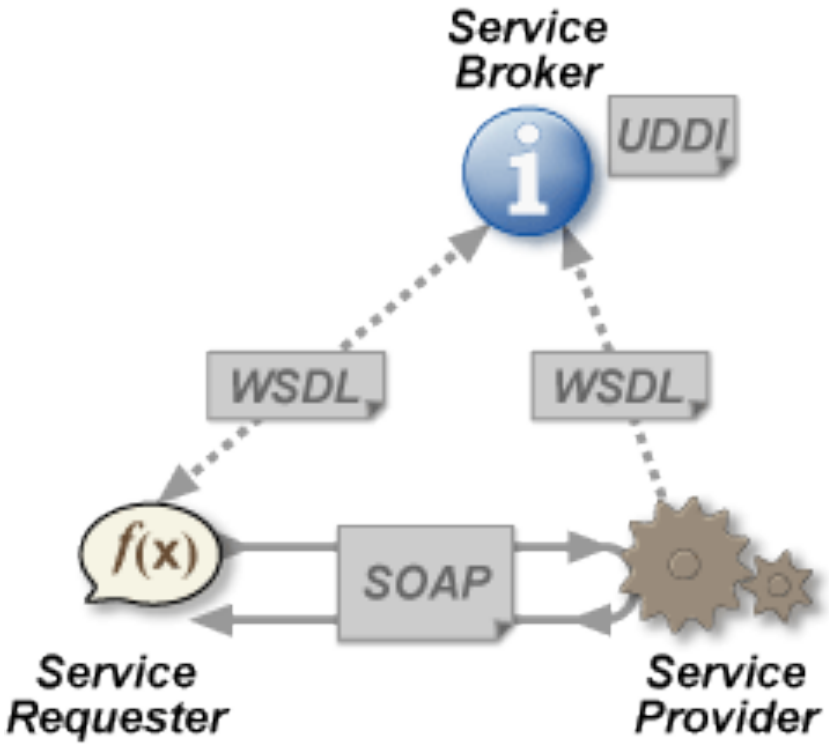
- Garantir inter-operabilidade
  - Usar protocolos web (HTTP e HTTPS) para transporte
    - Também se pode usar SMTP !!!
  - Usar URLs e URIs como referências para serviços remotos
  - Tratar problemas de heterogeneidade dos dados usando XML para representar os dados

Componentes dos web services



Componentes básicos

- SOAP: protocolo de invocação remota
- WSDL: linguagem de especificação de serviços
- UDDI: serviço de registo

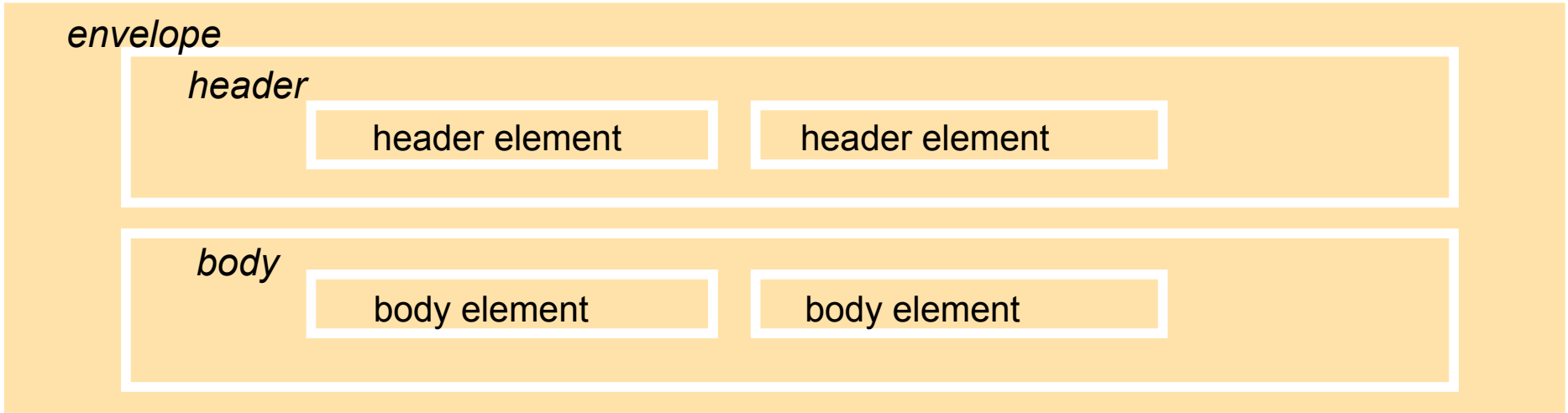


SOAP

- Protocolo para trocar mensagens XML
  - Inclui definição do formato das mensagens a trocar
  - Inclui um mecanismo de ligação das mensagens SOAP com o protocolo de transporte usado: HTTP ou HTTPS (ou SMTP, …)
  - Inclui mecanismo para tratar falhas

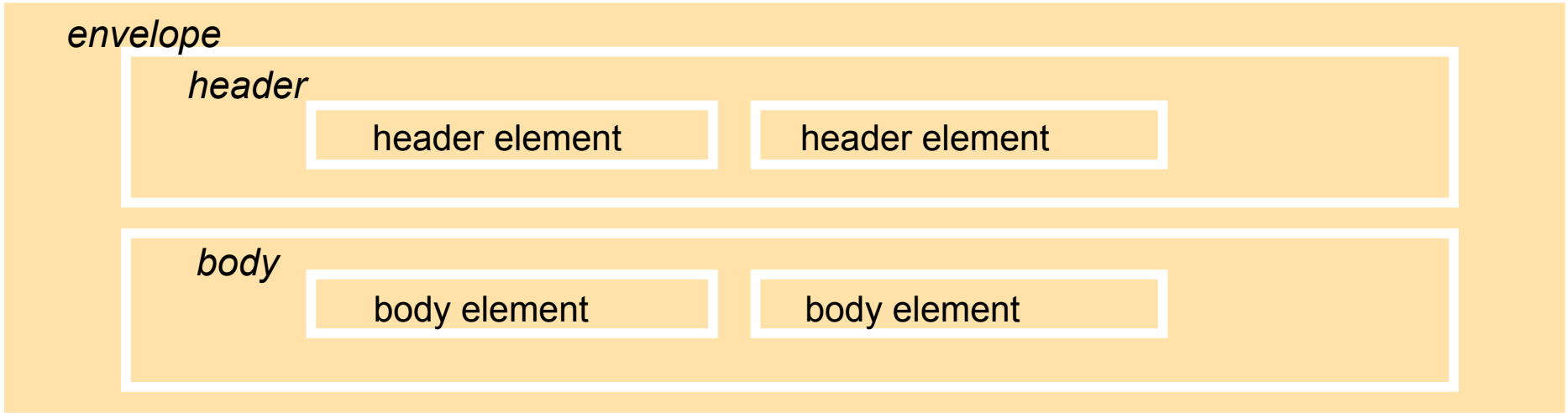
SOAP: mensagens

- No SOAP toda a informação está incluída no envelope da mensagem
- O envelope inclui elementos de cabeçalho e de corpo



SOAP: mensagens

```
<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/2001/XMLSchema' xmlns:soap='http://
schemas.xmlsoap.org/soap/envelope/' xmlns:soapenc='http://
schemas.xmlsoap.org/soap/encoding/' soap:encodingStyle='http://
schemas.xmlsoap.org/soap/encoding/'>
<soap:Body>
        <n:sayHello xmlns:n='urn:examples:helloservice'>
                <firstName xsi:type='xsd:string'>World</firstName>
        </n:sayHello>
</soap:Body>
</soap:Envelope>
```



SOAP: interacções

- **Oneway**: mensagem undireccional do cliente para o servidor
- **Pedido-resposta**: interacção cliente-servidor-cliente
- **Notificação**: interacção unidireccional servidor-cliente
  - E.g. callback, notificação
- **Notificação-resposta**: interacção servidor-cliente-servidor

WSDL – IDL para *web services*

- Definição da interface em XML
  - WSDL permite definir a interface do serviço, indicando quais as operações disponíveis
  - WSDL define as mensagens trocadas na interacção (e.g. na invocação duma operação, quais as mensagens trocadas)
  - WSDL permite também definir a forma de representação dos dados e a forma de aceder ao serviço

  - Especificação WSDL bastante verbosa – normalmente criada a partir de interface ou código do servidor
    - Em Java e .NET existem ferramentas para criar especificação a partir de interfaces Java
    - Sistemas de desenvolvimento possuem *wizards* que simplificam tarefa

WSDL: elementos

- No seu conjunto, um documento WSDL comporta os seguintes elementos:
  - **Types** - definições de tipos de dados, num dado sistema (e.g., XSD)
  - **Messages** - definição abstracta dos dados trocados nas operações
  - **Operation** - definição abstracta das acções suportadas pelo serviço
  - **Port Type** - conjunto abstracto das operações suportadas por um ou mais *endpoints*
  - **Binding** - uma especificação concreta do protocolo e formato de dados usados por um *port type.*
  - **Port** - um *endpoint* concreto que combina um endereço e um *binding* particulares.
  - **Serviço** - um conjunto de endpoints/ports relacionados.

WSDL a partir do Java

Anota-se uma classe usando: WebService e WebMethod

```
import javax.jws.*;
import java.util.*;

@WebService()
public class SimpleWSServer {
        private Map<String, MyInfo> infos;

        public SimpleWSServer() {
                infos = new HashMap<String, MyInfo>();
        }
        @WebMethod()
        public void addInfo( String name, int age) {
                infos.put( name, new MyInfo( name, age));
        }
        @WebMethod()
        public MyInfo getInfo( String name) throws InfoNotFoundException {
                if( infos.containsKey( name))
                        return infos.get( name);
                else
                        throw new InfoNotFoundException();
        }
```

Para gerar WSDL:
wsgen -cp . aulaWS.SimpleWSServer -wsdl -servicename {http://basename}/MyServer

Código gerado: SimpleWSServer.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://aulaWS/" xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://
aulaWS/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://aulaWS/" schemaLocation="SimpleWSServer_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="getInfo">
    <part name="parameters" element="tns:getInfo"/>
  </message>
  <message name="getInfoResponse">
    <part name="parameters" element="tns:getInfoResponse"/>
  </message>
  <message name="InfoNotFoundException">
    <part name="fault" element="tns:InfoNotFoundException"/>
  </message>
  ...
  <portType name="SimpleWSServer">
    <operation name="getInfo">
      <input message="tns:getInfo"/>
      <output message="tns:getInfoResponse"/>
      <fault name="InfoNotFoundException" message="tns:InfoNotFoundException"/>
    </operation>
  ...
  </portType>
</definitions>
```

Código gerado: SimpleWSServer.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://aulaWS/" xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://
aulaWS/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://aulaWS/" schemaLocation="SimpleWSServer_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="getInfo">
    <part name="parameters" element="tns:getInfo"/>
  </message>
  <message name="getInfoResponse">
    <part name="parameters" element="tns:getInfoRespons
  </message>
  <message name="InfoNotFoundException">
    <part name="fault" element="tns:InfoNotFoundException"/>
  </message>
  ...
  <portType name="SimpleWSServer">
    <operation name="getInfo">
      <input message="tns:getInfo"/>
      <output message="tns:getInfoResponse"/>
      <fault name="InfoNotFoundException" message="tns:InfoNotFoundException"/>
    </operation>
  ...
  </portType>
</definitions>
```

portType

Define os serviços do Web Service

Mensagens permitem assinatura dos serviços

Código gerado: MyServer.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="http://a
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="InfoNotFoundException" type="t
 ...
  <xs:element name="getInfo" type="tns:getInfo"/>
  <xs:element name="getInfoResponse" type="tns:getInfoResponse"/>
  <xs:complexType name="getInfo">
    <xs:sequence>
      <xs:element name="arg0" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="getInfoResponse">
    <xs:sequence>
      <xs:element name="return" type="tns:myInfo" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="myInfo"/>
  <xs:complexType name="InfoNotFoundException">
    <xs:sequence>
      <xs:element name="message" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

tipos

Tipos usados nas mensagens são definidos em XML Schema

Código gerado: SimpleWSServer.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://aulaWS/" xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://
aulaWS/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://aulaWS/" schemaLocation="SimpleWSServer_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="getInfo">
    <part name="parameters" element="tns:getInfo"
  </message>
  <message name="getInfoResponse">
    <part name="parameters" element="tns:getInfoResponse"/>
  </message>
  <message name="InfoNotFoundException">
    <part name="fault" element="tns:InfoNotFoundException"/>
  </message>
  ...
  <portType name="SimpleWSServer">
    <operation name="getInfo">
      <input message="tns:getInfo"/>
      <output message="tns:getInfoResponse"/>
      <fault name="InfoNotFoundException" message="tns:InfoNotFoundException"/>
    </operation>
  ...
  </portType>
</definitions>
```

message

As mensagens podem ser de input, output ou assinalar erros

Código gerado: MyServer.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://localhost/WS/" name="MyServer" ... xmlns:tns="http://localhost/WS/" ... >
  <import namespace="http://aulaWS/" location="SimpleWSServer.wsdl"/>
  <binding name="SimpleWSServerPortBinding" type="ns1:SimpleWSServer" xmlns:ns1="http://aulaWS/">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="getInfo">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
      <fault name="InfoNotFoundException">
        <soap:fault name="InfoNotFoundException" use="literal"/>
      </fault>
    </operation>
    ...
  </binding>
  <service name="MyServer">
    <port name="SimpleWSServerPort" binding="tns:SimpleWSServerPortBinding">
      <soap:address location="http://webserver/WS/MyServer"/>
    </port>
  </service>
</definitions>
```

binding

Concretiza o serviço definindo como se processam as interacções

Código gerado: MyServer.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://localhost/WS/" name="MyServer" ... xmlns:tns="http://localhost/WS/" ... >
  <import namespace="http://aulaWS/" location="SimpleWSServer.wsdl"/>
  <binding name="SimpleWSServerPortBinding" type="ns1:SimpleWSServer" xmlns:ns1="http://aulaWS/">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="getInfo">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
      <fault name="InfoNotFoundException">
        <soap:fault name="InfoNotFoundException" use="literal"/>
      </fault>
    </operation>
    ...
  </binding>
  <service name="MyServer">
    <port name="SimpleWSServerPort" binding="tns:SimpleWSServerPortBinding">
      <soap:address location="http://webserver/WS/MyServer"/>
    </port>
  </service>
</definitions>
```

port

Define o endereço de rede onde o web service é disponibilizado

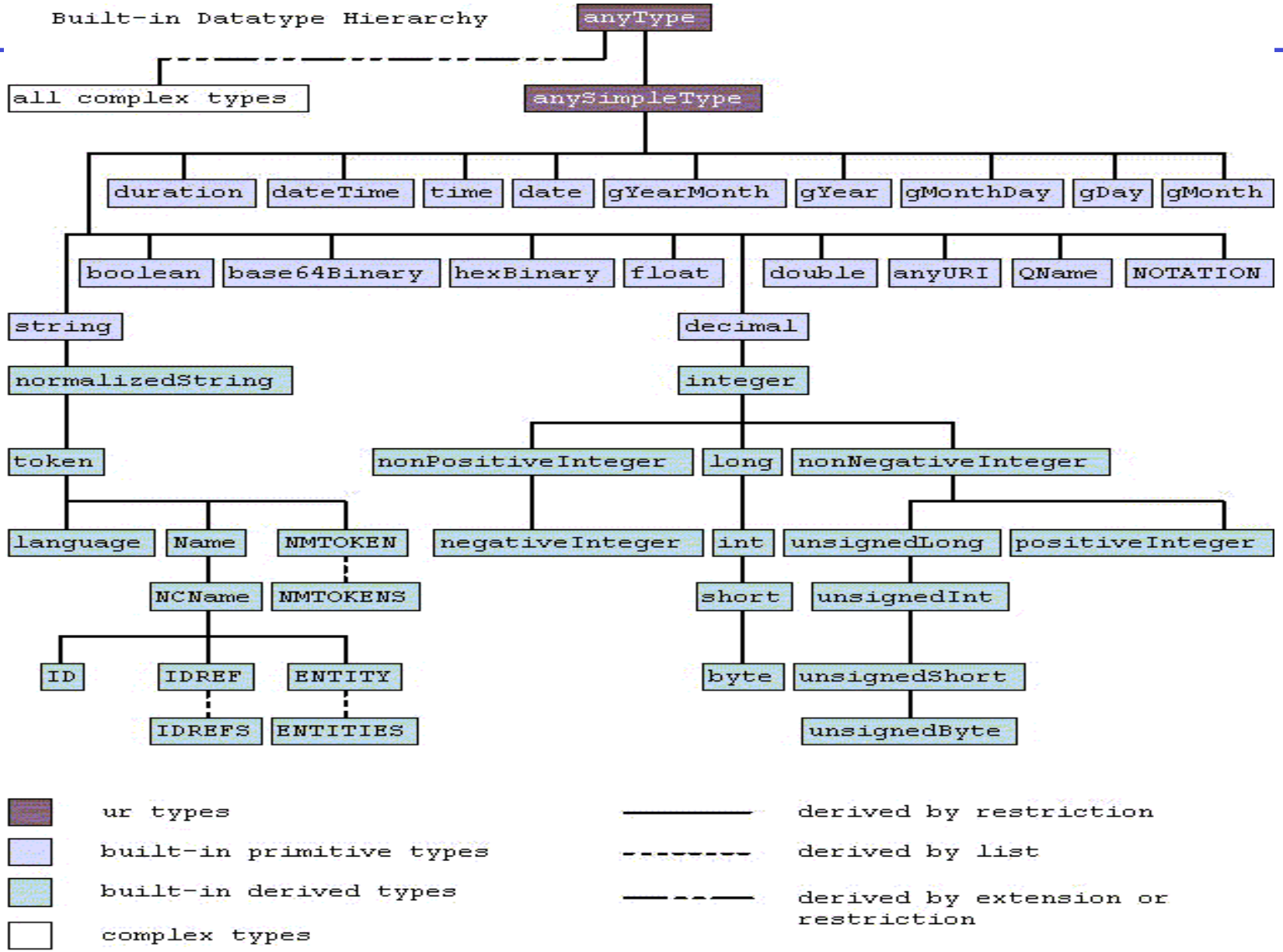
WSDL a partir do .NET

```
using System;
using System.WebServices;
using System.Xml.Serialization;

[WebService(Namespace="http://127.0.0.1:8088/PingImpl")]
public class PingImpl : System.WebServices.WebService
{
    [WebMethod]
    public string ping( string v )
    {
        return v.ToUpper() ;
    }
}
```

Web Services: tipos

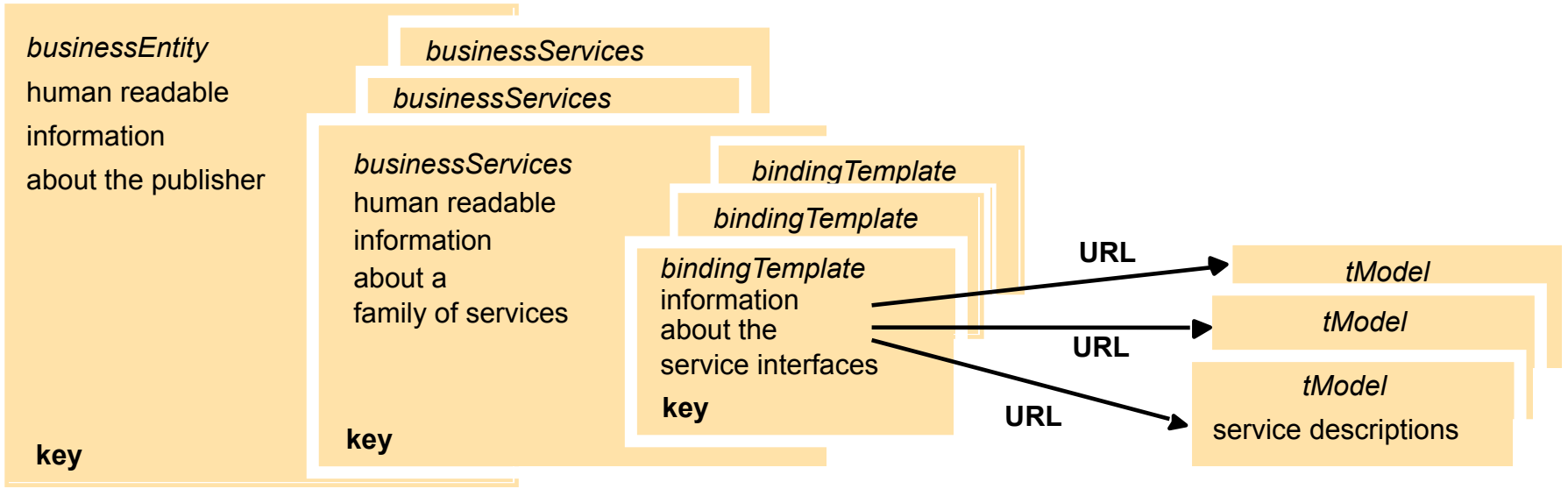
- Noção de tipos simples e tipos complexos.
  - Os tipos simples (*built-in*) XSD são os usuais, mas inclui diversos outros derivados: *decimal, integer, positiveinteger, normalizedstring, token, etc.*
  - Os tipos complexos correspondem a novos tipos derivados a partir de outros tipos já definidos.
    - A **diferença principal** entre tipos simples e complexos é que os tipos simples **não podem ter** sub-elementos ou atributos.
- Suporta a noção de hierarquia de tipos por derivação a partir de **um tipo base** (*any*)
  - Há 4 regras de derivação:
    - **restriction** (limita o intervalo de valores permitido no tipo base)
    - **extension** (aumenta a gama de valores permitido no tipo base)
    - **list** (define uma colecção de valores de um dado tipo)
    - **union** (define tipos análogos aos registos com variante de Pascal)

Tipos XML

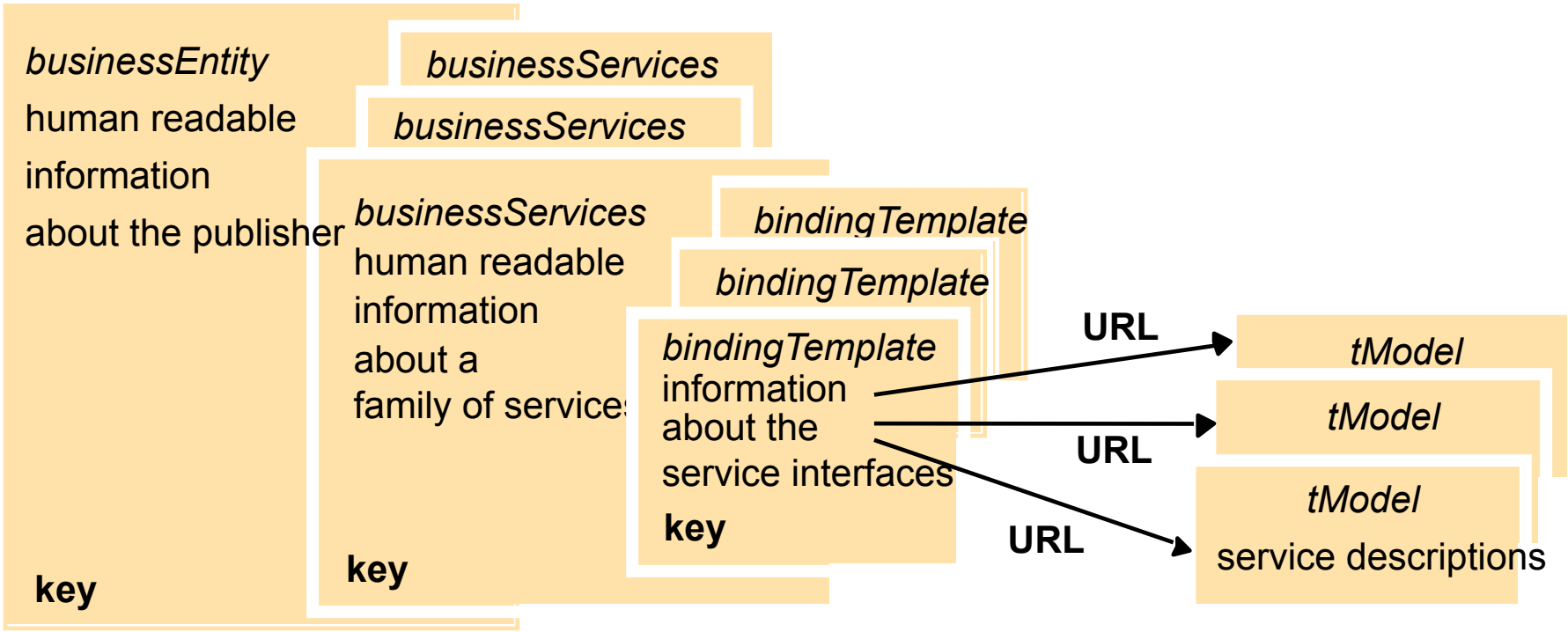

Built-in Datatype Hierarchy
anyType
all complex types
anySimpleType
duration
dateTime
time
date
gYearMonth
gYear
gMonthDay
gDay
gMonth
boolean
base64Binary
hexBinary
float
double
anyURI
QName
NOTATION
string
decimal
normalizedString
integer
token
nonPositiveInteger
long
nonNegativeInteger
language
Name
NMTOKEN
negativeInteger
int
unsignedLong
positiveInteger
NCName
NMTOKENS
short
unsignedInt
ID
IDREF
ENTITY
byte
unsignedShort
IDREFS
ENTITIES
unsignedByte
ur types
built-in primitive types
built-in derived types
complex types
derived by restriction
derived by list
derived by extension or restriction

UDDI

- Protocolo de ligação ou binding entre o cliente e o servidor
- Os fornecedores dos serviços publicam a respectiva interface
- O protocolo de inspecção permite verificar se uma dado serviço existe baseado na sua identificação
- O UDDI permite encontrar o serviço baseado na sua definição – capability lookup



Estruturas de dados UDDI



Clientes

- O cliente pode ter um **static proxy** criado antes da execução (static stub) que é compilado a partir do WSDL

- O cliente pode também usar um **dynamic proxy** uma classe que é criada durante a execução a partir do WSDL

Cliente em Java

```
import javax.xml.ws.WebServiceRef;

public class SimpleClient {
  @WebServiceRef(wsdlLocation="http://localhost:8080/WS/SimpleServer?wsdl")
  static MyServer service;

  public static void main(String[] args) {
     new SimpleClient().doTest(args);
   }

  public void doTest(String[] args) {
    try {
      SimpleWSServer port = service.getSimpleWSServerPort();

       MyInfo response = port.getInfo(args[0]);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Criar servidor/cliente usando Java

- Uma solução consiste em utilizar os *wizards* disponíveis nas ferramentas de desenvolvimento de software

- No Eclipse existem tutoriais que mostram como criar um servidor ou um cliente usando os wizards disponibilizados

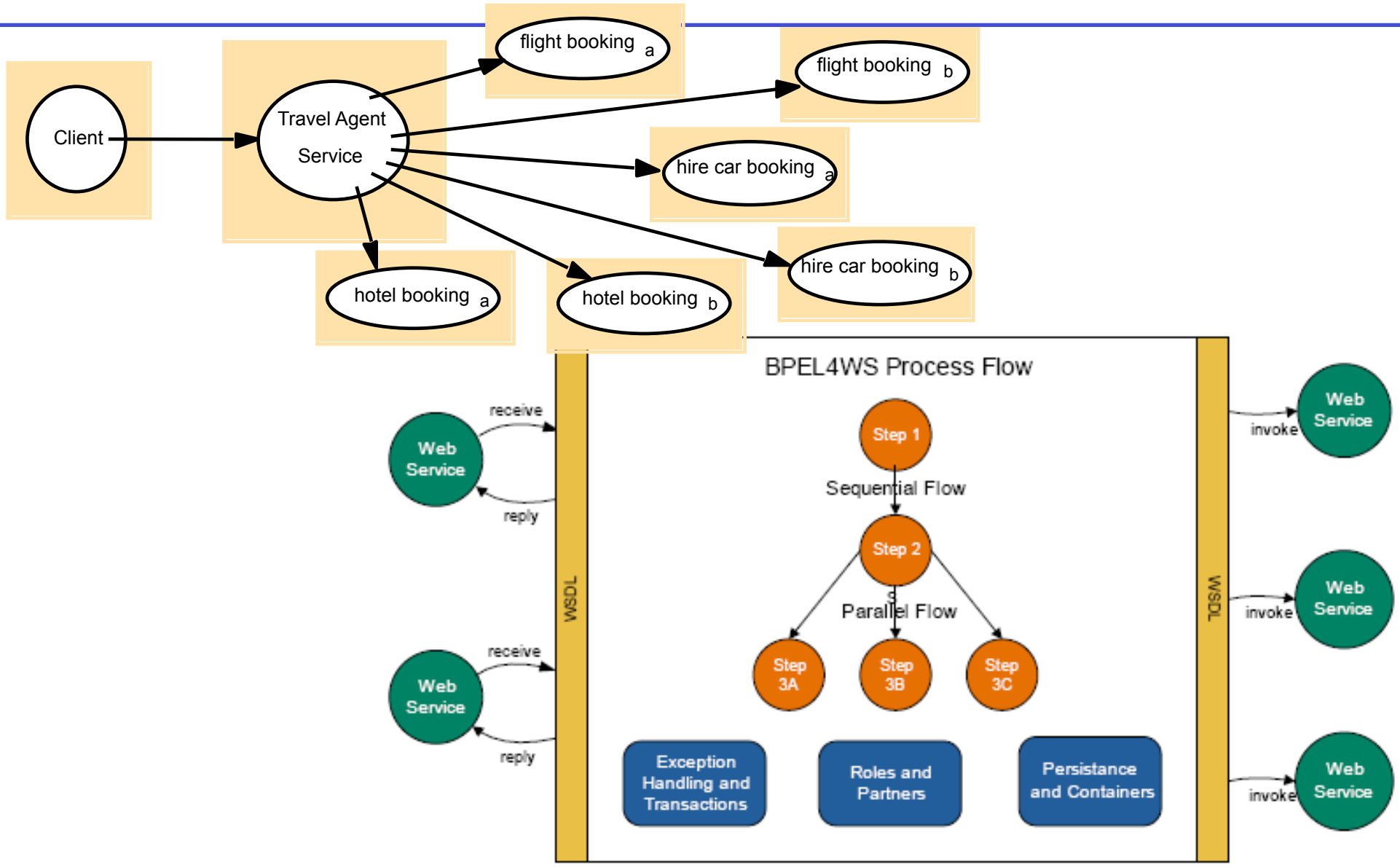
- http://www.eclipse.org/webtools/community/communityresources.html#tutorials

- Nas aulas práticas apresentou-se como criar cliente e servidor usando ferramentas disponíveis no Java 6

WS-*

- WS-Reliability: especificação que permite introduzir fiabilidade nas invocações remotas (com as diferentes semânticas)
  - WS-ReliableMessaging
- WS-Security: define como efectuar invocações seguras
- WS-Coordination: fornece enquadramento para coordenar as acções de aplicações distribuídas (coreografia de web services)
  - WS-AtomicTransaction: forncece coordenação transaccional (distribuída) entre múltiplos web services
  - E.g.: BPEL4WS, WSBPEL linguagem de orquestração

WS-Coordination



Web Services: resumo

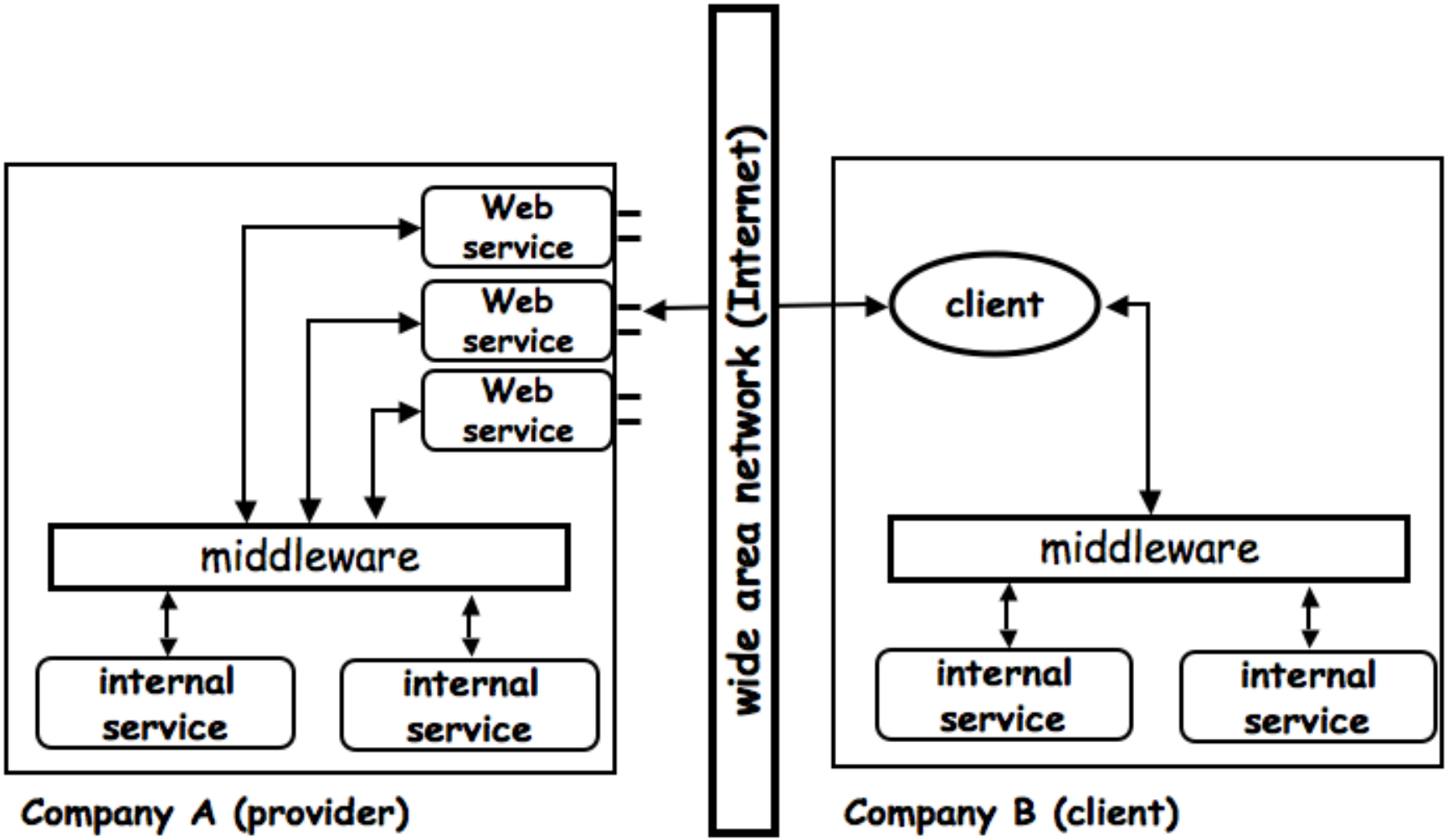
- Tem ganho popularidade na indústria porque apresenta uma solução para integrar diferentes aplicações
  - RMI – apenas Java
  - Corba/IIOP – suporte mais complexo
- Permite:
  - Utilização de standards
    - HTTP, XML
  - Reutilização de serviços (arquitecturas orientadas para os serviços)
    - WS-Coordination
  - Modificação parcial/evolução independente dos serviços
    - WSDL
  - Disponibilidade, assincronismo
    - WS-Eventing

Web services: utilizações possíveis

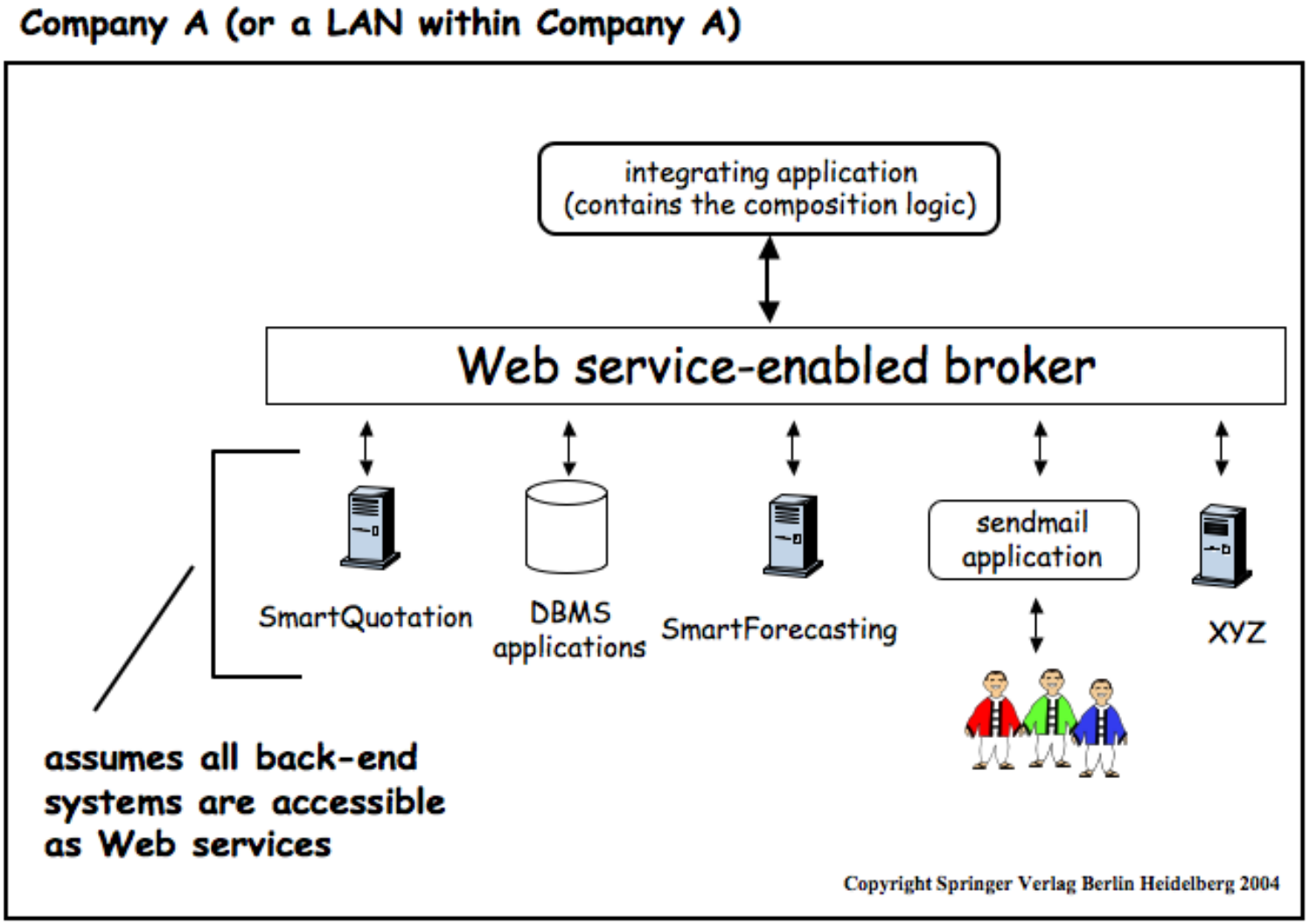
- Inter-operação entre instituições
  - Utilização tende a ser limitada
  - Promessas de todos os serviços disponíveis para todos acabaram por não se concretizar

- Inter-operação entre sistemas duma mesma instituição
  - Utilização com forte implantação

Web services: entre organizações



Web services: dentro de uma organização



Web services: problemas

- Desempenho:
  - Complexidade do XML
  - Difícil fazer *caching*: noção de operação + estado
- Necessário cuidado para fazer evoluir o servidor
- Complexidade
  - Necessário utilização de ferramentas de suporte

REST: Representational State Transfer

- Aproximação: vê uma aplicação como uma colecção de recursos
  - Um recuso é identificado por um URI/URL
  - Um URL devolve um documento com a representação do recurso
  - Podem-se fazer referências a outros recursos usando *ligações (links)*

- Estilo arquitectural, não um sistema de desenvolvimento

- Aproximação proposta por Roy Fielding na sua tese de doutroamento
  - Não como uma alternativa aos web services, mas como uma forma de aceder a informação

REST: princípios de desenho

- Protocolo cliente/servidor stateless: cada pedido contém toda a informação necessária para ser processado – objectivo: tornar o sistema simples.
- Recursos – no sistema tudo são recursos, identificados por um URI/URL.
  - Recursos tipicamente armazenados num formato estruturado que suporta hiper-ligações (e.g. XML)
- Interface uniforme: todos os recursos são acedidos por um conjunto de operações bem-definidas. Em HTTP: POST, GET, PUT, DELETE (equiv. criar, ler, actualizar, remover).
- Cache: para melhorar desempenho, respostas podem ser etiquetadas como permitindo ou não *caching*.

Mais detalhes e exemplo

- Usa HTTP GET, POST, etc., mas
  - Usa dados bem-tipados (usando schemas XML, json)
- Um exemplo (de Roger L. Costello)
  - Uma empresa pretende disponibilizar web services REST para permitir aos seus clientes:
    - Obter uma lista de peças
    - Obter informação detalhada sobre uma peça
    - Submeter uma ordem de compra

Exemplo

- Tornar uma lista de peças disponível como um recurso
  - http://www.parts-depot.com/parts
  - Enviar um pedido HTTP GET para o recurso devolve

```
<?xml version="1.0"?>
   <p:Parts xmlns:p="http://www.parts-depot.com"
            xmlns:xlink="http://www.w3.org/1999/xlink">
        <Part id="00345" xlink:href="http://www.parts-depot.com/parts/00345"/>
        <Part id="00346" xlink:href="http://www.parts-depot.com/parts/00346"/>
        <Part id="00348" xlink:href="http://www.parts-depot.com/parts/00348"/>
   </p:Parts>
```

Exemplo (cont.)

- Tornar uma lista de peças disponível como um recurso
  - http://www.parts-depot.com/parts?query=hammer
    - Ou http://www.parts-depot.com/parts/hammer
  - Enviar um pedido HTTP GET para o recurso devolve

```
<?xml version="1.0"?>
   <p:Parts xmlns:p="http://www.parts-depot.com"
            xmlns:xlink="http://www.w3.org/1999/xlink">
        <Part id="00345" xlink:href="http://www.parts-depot.com/parts/00345"/>
   </p:Parts>
```

Exemplo (cont.)

- Obter detalhes sobre uma peça
  - http://www.parts-depot.com/parts/00345?flavor=xml
  - Enviar um pedido HTTP GET para o recurso devolve

```
<?xml version="1.0"?>
    <p:Part xmlns:p="http://www.parts-depot.com"
            xmlns:xlink="http://www.w3.org/1999/xlink">
          <Part-ID>00345</Part-ID>
          <Name>Widget-A</Name>
          <Description>This part is used within the frap assembly</Description>
          <Specification xlink:href="http://www.parts-depot.com/parts/00345/
   specification"/>
          <UnitCost currency="USD">0.10</UnitCost>
          <Quantity>10</Quantity>
    </p:Part>
```

Exemplo (cont.)

- Submeter uma ordem de compra
  - Cliente deve saber ou obter informação de como representar o pedido num documento
  - A seguir efectua um HTTP POST http://www.parts-depot.com/order

```
<?xml version="1.0"?>
    <p:Order xmlns:p="http://www.parts-depot.com"
             xmlns:xlink="http://www.w3.org/1999/xlink">
          <Part-ID>00345</Part-ID>
          <UnitCost currency="USD">0.10</UnitCost>
          <Quantity>10</Quantity>
    </p:Part>
```

REST vs. RPCs

- Nos sistemas de RPCs a ênfase é nas operações que podem ser invocadas
- Nos sistemas REST, o ênfase é nos recusos, na sua representação e em como estes são afectados por um conjunto de métodos standard

Outro exemplo: sistema sobre utilizadores

(baseado em documento REST na wikipedia)

- Ideia: sistema que mantém informação sobre utilizadores

- RPCs:
  - getUser()
  - addUser()
  - removeUser()
  - updateUser()
  - listUsers()
  - findUser()

- REST:
  - http://example.com/users/{user}

  (GET, POST, DELETE, PUT)

  - http://example.com/users/
  - http://example.com/find/Nuno+Preguica

REST: notas soltas

- Tem ganho popularidade
  - Muitos serviços web disponibilizados segundo este modelo
  - Grandes vantagens: simplicidade e desempenho

- Serviços disponibilizados num modelo REST nem sempre aderem completamente aos princípios REST
  - Difícil disponibilizar número elevado de métodos

  - http://api.flickr.com/services/rest/?method=flickr.photos.search&appid=sdfjkshf

REST: suporte Java

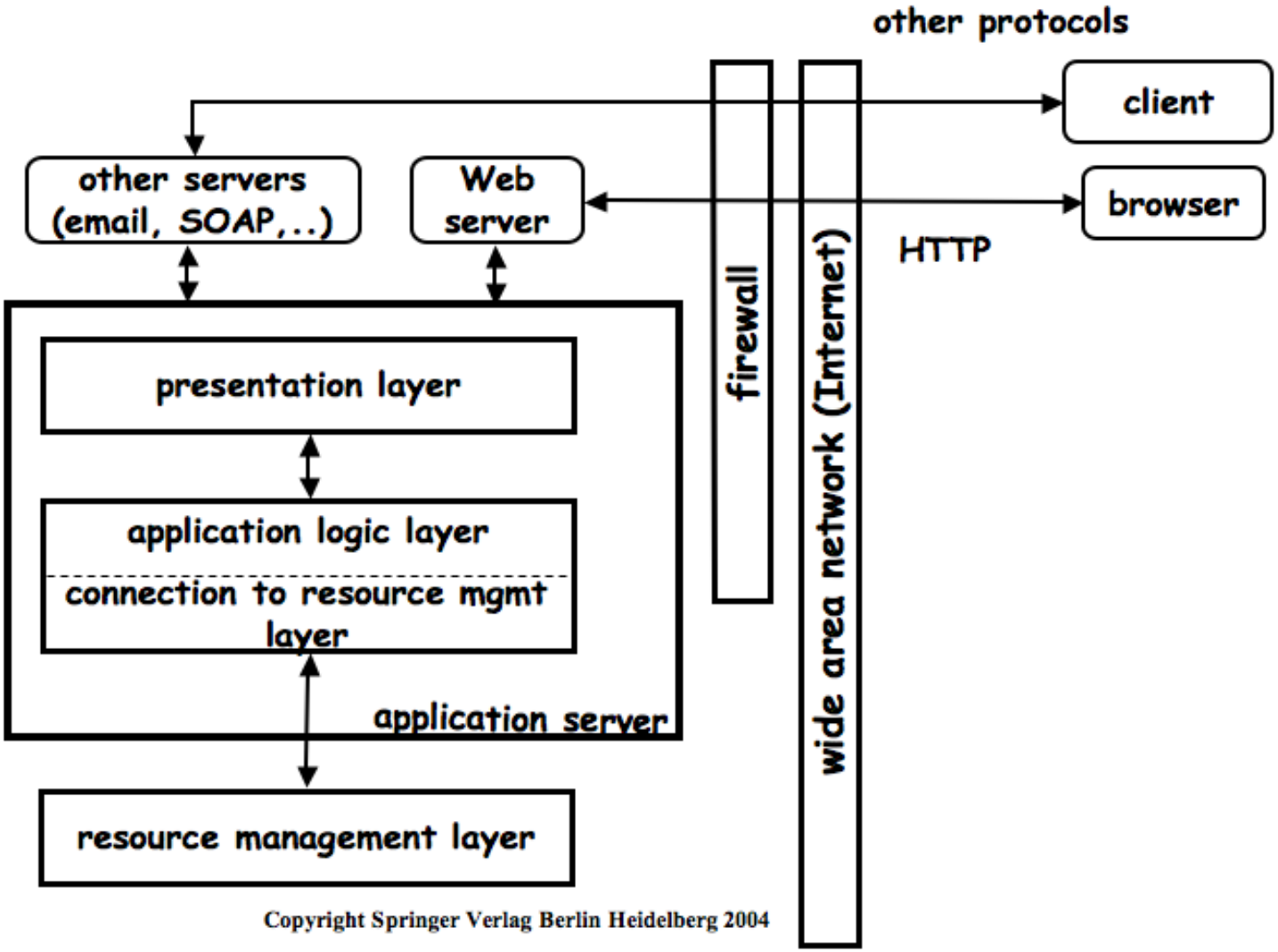
- Definido em JAX-RS (JSR 311)
- Suporte linguístico baseado na utilização de anotações
  - Permite definir que um dado URL leva à execução dum dado método
  - Permite definir o modo de codificação da resposta
    - XML – usando mecanismo standard de codificação de cobjectos java em XML fornecido pelo JAXB
    - JSON – mecanismo leve de codificação de tipos (RFC 4627)

```
{
    "firstName": "John",
    "lastName": "Smith",
    "address": {
        "streetAddress": "21 2nd Street",
       "postalCode": 10021
    },
    "phoneNumbers": [
        "212 555-1234",
        "646 555-4567"
    ]
}
```

REST: suporte Java (exemplo)

```
@Path("/customerservice/")
@ProduceMime("application/xml")
public class CustomerService {
   @GET
   public Customers getCustomers() {   ......   }

   @GET
   @Path("/customers/{id}")
   @Produces("application/json")
   public Customer getCustomer(@PathParam("id") String id) {......   }

   @PUT
   @Path("/customers/{id}")
   @Consumes("application/xml")
   public Response updateCustomer(@PathParam("id") Long id, Customer customer) {   ......   }

   @POST
   @Path("/customers")
   public Response addCustomer(Customer customer) {   ......   }

   @DELETE
   @Path("/customers/{id}/")
   public Response deleteCustomer(@PathParam("id") String id) {   ......   }

   @Path("/orders/{orderId}/")
   public Order getOrder(@PathParam("orderId") String orderId) {   ......   }
}
```

Web services e REST: arquitectura de suporte



Web services ou REST?

- Questão em aberto
- REST:
  - Mais simples
  - Mais eficiente
  - Complicado implementar serviços complexos
- Web services
  - Mais complexo
  - Grande suporte da indústria
- E.g.:
  - http://www.oreillynet.com/pub/wlg/3005

AJAX: Asynchronous JavaScript with XML

- Ideias chave:
  - Apresentação baseada em standards: XHTML e CSS;
  - Apresentação e interação dinâmica usando o Document Object Model;
  - Troca e manipulação de dados usando XML e XSLT;
  - Obtenção assíncrona de dados usando XMLHttpRequest;
  - Utilização de JavaScript na implementação destes princípios.
- Objectivo: permitir ao cliente duma aplicação, a executar num browser, apresentar uma interface *rica* e interactiva
  - Riqueza: Muitos widgets disponíveis
  - Interactividade
    - Possibilidade de modificar a página actual: modificando a árvore DOM
    - Possibilidade de enviar/receber pedidos sem se bloquear: operações XMLHttpRequest não bloqueante

Ajax vs. web clássico

- Web:
  - Utilizador clica em links ou forms
  - Servidor obtém dados dum form e computa nova página
  - Browser apresenta nova página completa
- Ajax:
  - Acção do utilizador desencadeia modificação rápida da interface
  - E.g. Gmail, Google Maps, etc.

Ajax (cont.)

- Baseado em Javascript
  - Javascript é uma linguagem de scripting suportada pela generalidade dos browsers
  - Algumas características:
    - Não é Java !!!
    - Não tipada
    - Sem threads
    - Suporte para eventos. Porquê?
    - XMLHttpRequest permite efectuar pedido HTTP, de forma assíncrona
- Desenvolvimento
  - Usando frameworks Ajax
    - E.g. Prototype, Dojo, Script.aculo.us, GWT

GWT: Google Web Toolkit

- Permite desenvolver aplicações Ajax de forma simples
- Desenvolvimento em Java
  - Compilador transforma Java em Javascript
- E.g. Gmail, Google Maps, etc.

- Suporte para invocação remota de procedimentos (semelhante ao RMI), mas suportando RPCs assíncronos
  - Mecanismo de serialização própria, permite serializar grafos de objectos

GWT: interface com método assíncrono

```
public interface MyService extends RemoteService {
    Shape[] getShapes(String databaseName) throws ShapeException;
}
public interface MyServiceAsync {
    void getShapes(String databaseName, AsyncCallback callback);
}

public class MyServiceImpl extends RemoteServiceServlet implements MyService {
    Shape[] getShapes(String databaseName) throws ShapeException {
            …
    }
}
```

GWT: interface com método assíncrono

```
service.getShapes(dbName, new AsyncCallback() {
   public void onSuccess(Object result) {
     Shape[] shapes = (Shape[]) result;
     controller.processShapes(shapes);
   }

   public void onFailure(Throwable caught) {
     try {
       throw caught;
     } catch (IncompatibleRemoteServiceException e) {
       // this client is not compatible with the server; cleanup and refresh the browser
     } catch (InvocationException e) {
       // the call didn't complete cleanly
     } catch (ShapeException e) {
       // one of the 'throws' from the original method
    } catch (Throwable e) {
       // last resort -- a very unexpected exception
     }
   }
 });
```

Para saber mais

- G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems - Concepts and Design, Addison-Wesley, 4th Edition, 2005
  - Web services – capítulo 19.1-19.4
- REST: http://en.wikipedia.org/wiki/Representational_State_Transfer
- GWT: http://code.google.com/webtoolkit/

Sistemas Distribuídos I

Capítulo 6

Segurança em sistemas distribuídos

Nota prévia

- A estrutura da apresentação é semelhante e utiliza algumas das figuras livro de base do curso

  G. Coulouris, J. Dollimore and T. Kindberg,
  Distributed Systems - Concepts and Design,
  Addison-Wesley, 4th Edition, 2005

  - Capítulo 7

- e também do capítulo de segurança do livro

  James F. Kurose and Keith W. Ross, "Computer Networking - A Top-Down Approach Featuring the Internet," Addison Wesley Longman, Inc., Second Edition, 2003

Segurança em sistemas distribuídos

- Este capítulo apresenta uma introdução às técnicas de segurança em sistemas distribuídos centrada na problemática da segurança das comunicações e da autenticação.

Organização do capítulo

- Apresentação do problema
  - Diferenças essenciais entre os sistemas centralizados e os sistemas distribuídos
  - Canais seguros e seus requisitos
- Criptografia simétrica
  - Utilização da criptografia simétrica na comunicação e autenticação
- Criptografia assimétrica
  - Utilização da criptografia assimétrica na comunicação e autenticação
- Assinaturas digitais
- Aplicações seguras mais comuns: SSL, PGP, ...

Da necessidade da segurança

- Dados e operações de um sistema necessitam de ser protegidas de ataques
- Sistema necessitam de fornecer mecanismos de segurança e implementar uma política de segurança para proteger dados e operações
  - Mecanismos de segurança fornecem protecção
  - Política de segurança define como mecanismos são usados
- Sistemas distribuídos são susceptíveis a novos ataques
- Sistemas distribuídos requerem mecanismos de segurança mais sofisticados face a um sistema isolado

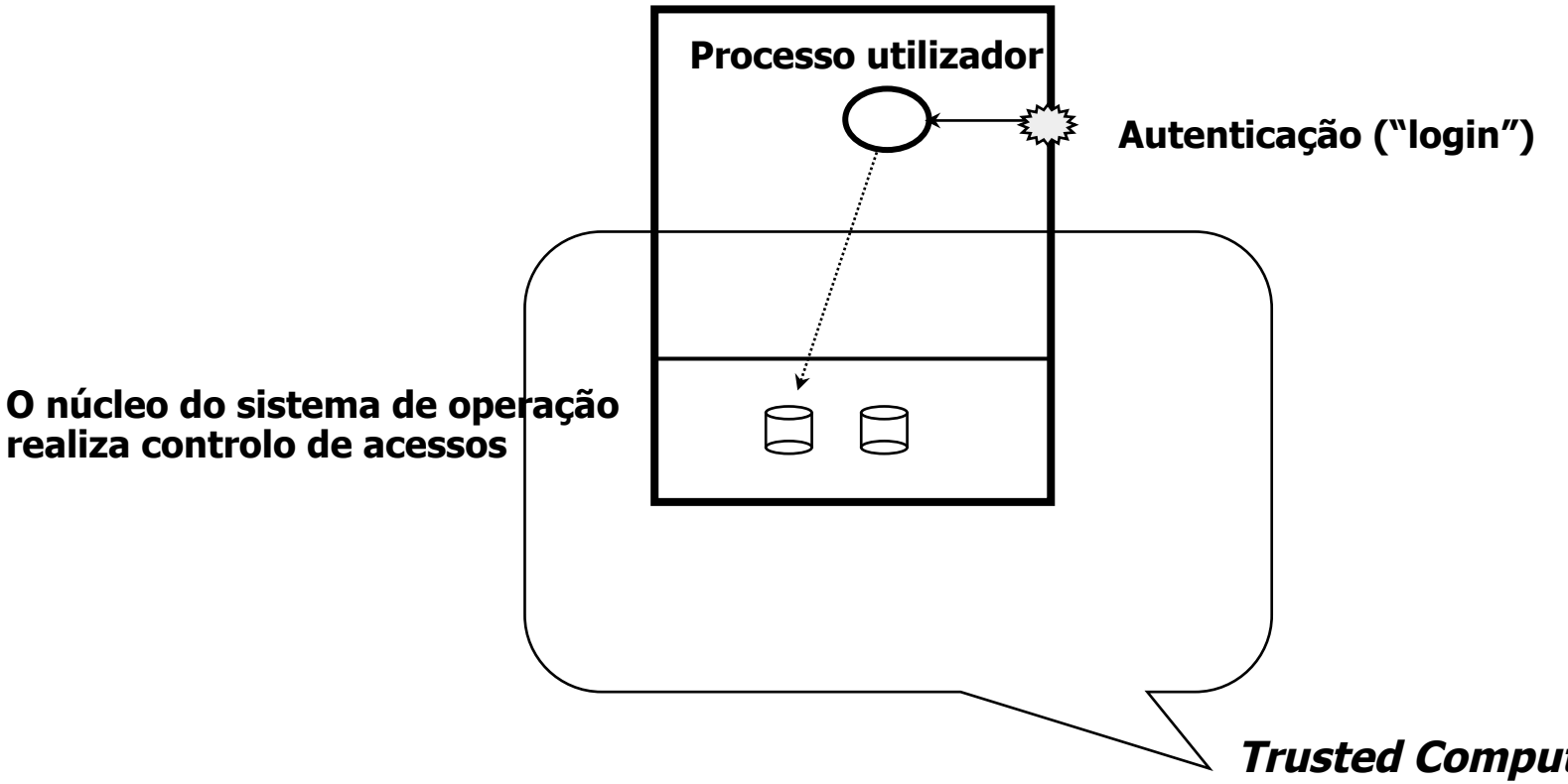
Elementos do modelo de segurança

- **Principal** - uma entidade (pessoa, processo, servidor, cliente, …) que é singular do ponto de vista dos direitos no sistema.

- **Controlo de acessos** - dada uma operação *Op* sobre um objecto *O*, é necessário decidir se o principal *P* pode aplicar *Op* a *O*.
  - Normalmente utilizam-se ACLs (Acess Control Lists) ou Capacidades (Tickets).

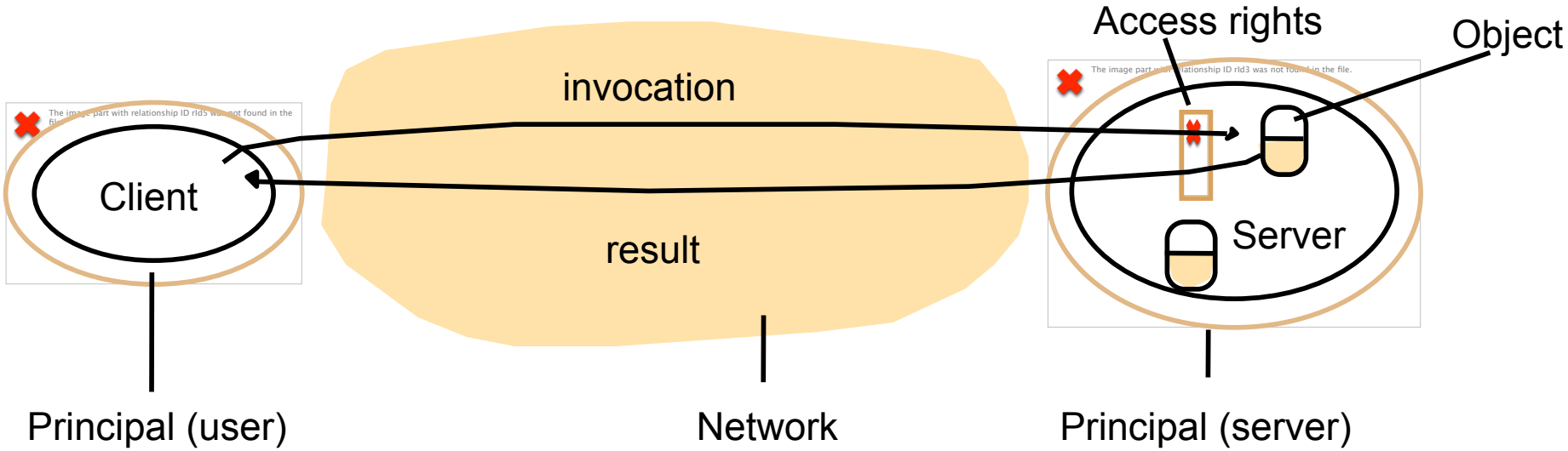
- **Autenticação** – processo de verificar que um principal tem a identidade que diz ter – em geral, deve ser capaz de o provar.
  - Geralmente utiliza-se um método lógico do tipo segredo partilhado entre P e quem o autentica (de que uma palavra chave é o exemplo mais conhecido), mas também se pode basear na verificação de atributos físicos (identificação da voz, impressões digitais ou da retina por exemplo) ou na posse de algo que só P pode possuir (um cartão magnético por exemplo).

Um sistema centralizado clássico



Trusted Computer Base
Mecanismos (software) nos quais se confia para implementar a política de segurança

Principais, objectos, direitos de acesso e canais



Onde está a *Trusted Computing Base* ?

"End-to-end arguments" em segurança

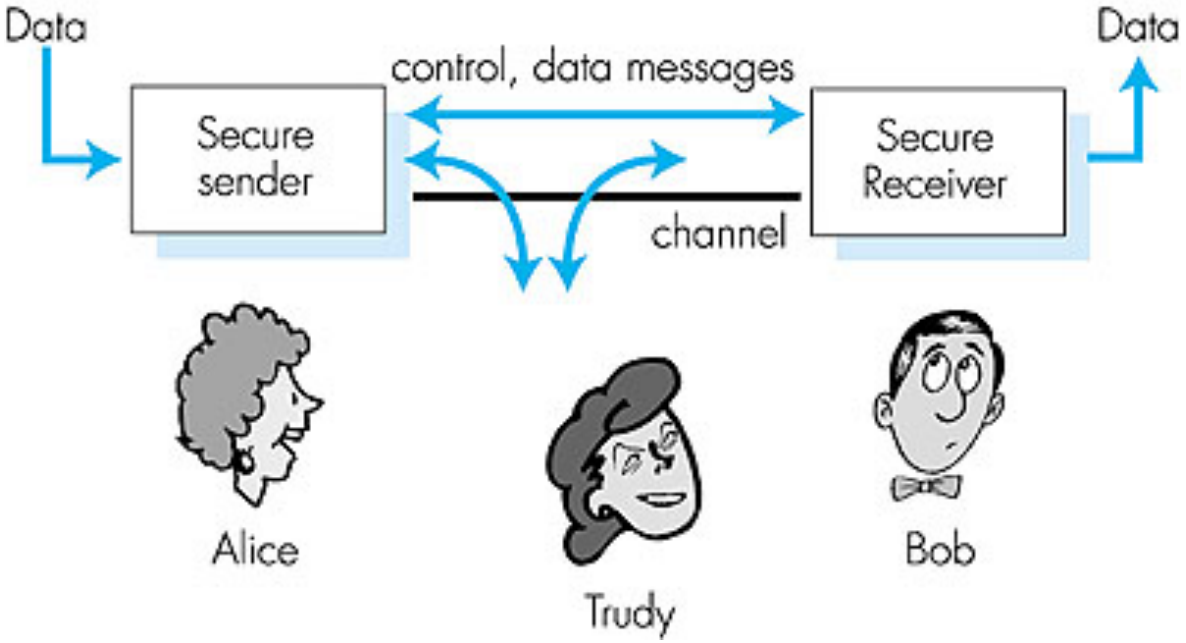
- A segurança deve depender de um conjunto de mecanismos de segurança e de uma *Trusted Computing Base* mínima, porque assim:
  - é mais fácil compreender os mecanismos de segurança,
  - é mais fácil verificar a sua correcção,
  - é mais fácil de implementar,
  - o número de intervenientes é menor,
  - só se paga o que se consome

Métodos de ataque

- Violar os mecanismos de autenticação
- Explorar inadequações na especificação ou implementação da *Trusted Computing Base* e adquirir direitos ou identidades indevidas
- Tentar modificar a *Trusted Computing Base*
- Obter segredos indevidos

- Como os ataques exploram *bugs* e fraquezas das implementações ou simples fraquezas de planeamento ou humanas, é geralmente impossível de enunciar todas as formas de ataque e quase sempre estão-se a descobrir novas.

Comunicação num sistema sem segurança



- Canal está acessível ao atacante
- Nomes usados na descrição dos protocolos de segurança:
  - Alice, Bob, Carol, Dave – participantes que querem comunicar
  - Mallory/Trudy – atacante que pode ler, interceptar, modificar suprimir ou re-introduzir mensagens nos canais ou tentar passar por um dos participantes (Eve é usado com frequência para um atacante que lê mensagens – eavesdropper)

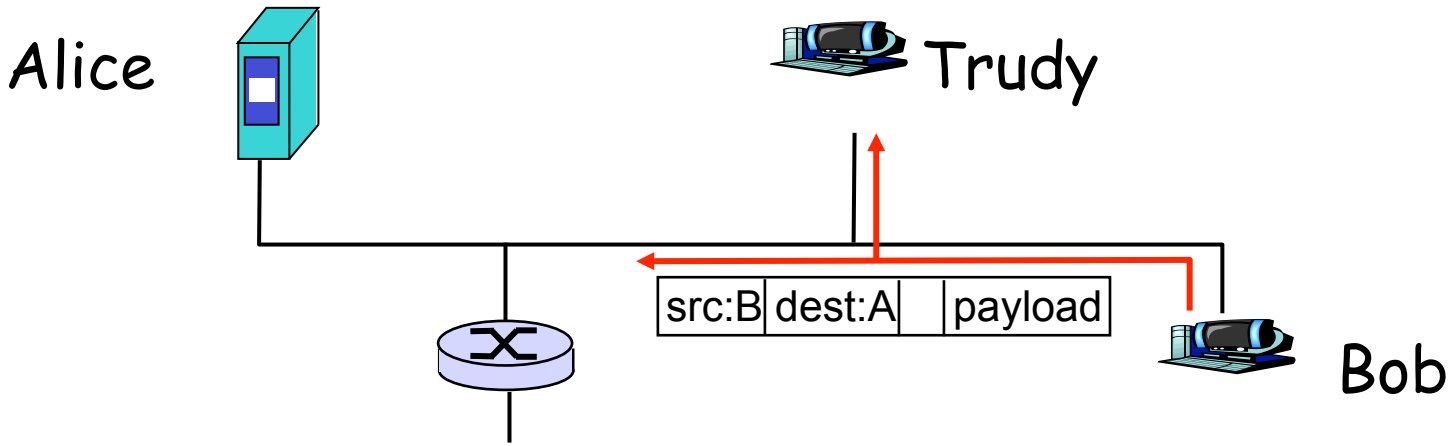
Ataques em sistemas distribuídos através da comunicação

- Indiscrição – obtenção de mensagens sem autorização (*Eavesdropping*)
- Mascarar-se ou pretender ser outro (*Masquerading*)
- Reemissão de antigas mensagens (*Message replaying*)
- Alteração indevida do conteúdo das mensagens (*Message tampering*)
  - Supressão de mensagens (*Message supression*)
- Ataques de impedimento de prestação de serviço (*Denial of service attacks*)

- Repudiar as mensagens enviadas pelo próprio
- Análise de tráfego (*Traffic analysis*)

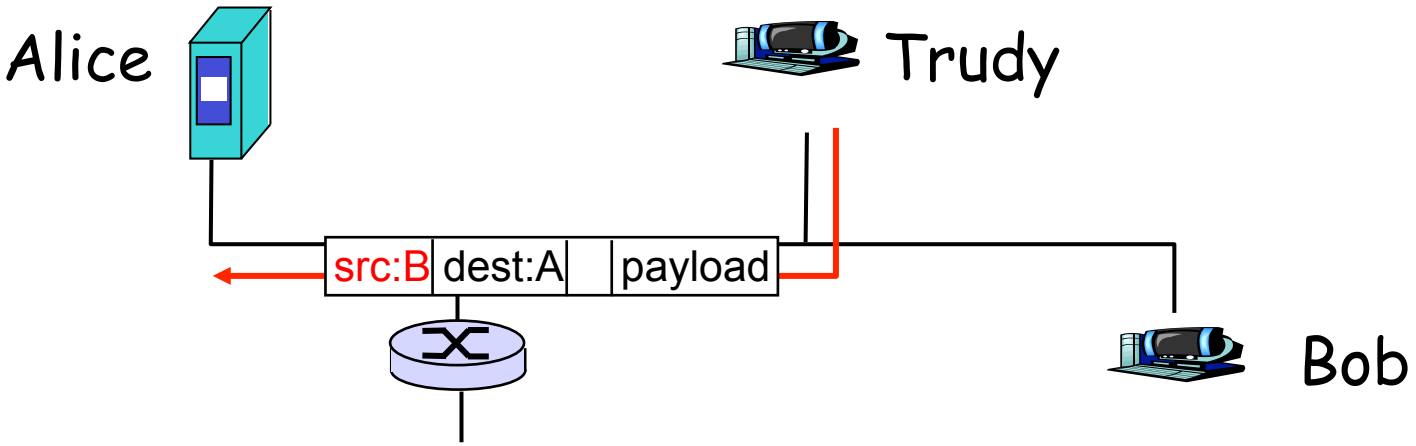
Eavesdropping (exemplo): cópia dos pacotes que transitam na rede

- *Packet sniffing*:
  - *broadcast media*
  - modo *promiscuous* da placa *ethernet* (NIC) permite copiar todos os pacotes
  - permite a leitura de todos os dados não cifrados (exemplo: *passwords*)
  - exemplo: *Trudy sniffs Bob's packets*



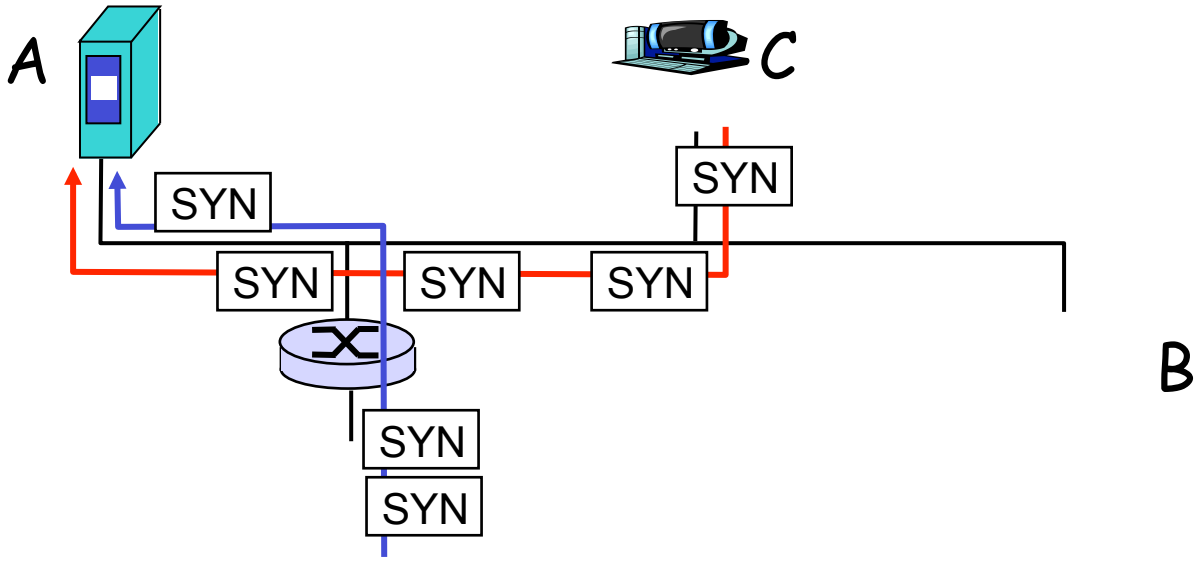
Masquerading (exemplo): mascarar o emissor

- *IP Spoofing*:
  - Permite a geração de pacotes directamente da aplicação mas em que o endereço origem é falso (está *spoofed*)
  - o receptor não pode detectar que o pacote está *spoofed*
  - e.g.: Trudy mascara-se como Bob



Impedimento de prestação de serviço (exemplo)

- Denial of service (DoS):
  - um conjunto de pacotes inundam o receptor impedindo-o de fazer qualquer trabalho útil (DoS- Denial Of Service attack)
  - Distributed DoS (DDoS): ataque distribuído e coordenado de impedimento da prestação de serviço
  - exemplo, C e um host remoto realizam um SYN-attack a A



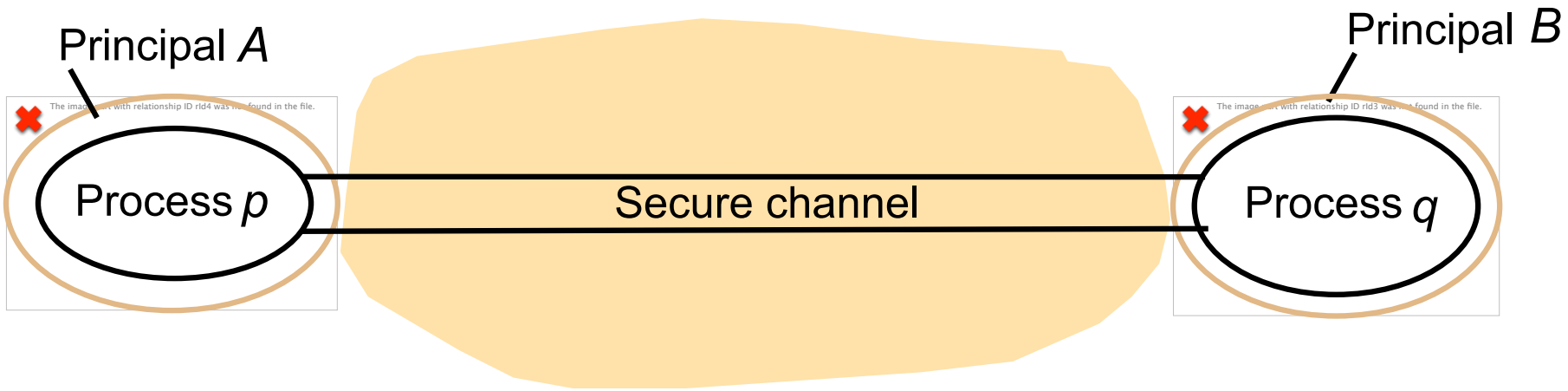
Mecanismos de segurança mais comuns

- Baseados na utilização de técnicas criptográficas
  - Construir canais seguros imunes à repetição (*replaying*) e à modificação do conteúdo
  - Autenticar clientes e servidores
  - Assinaturas digitais

- Introduzir padrões de comunicação falsos

- Nota: cifrar e não "encriptar"

Canais seguros



- Objectivo: trocar dados com confidencialidade, integridade e autenticidade
  - Num canal seguro os interlocutores (A e B) estão autenticados
  - O inimigo não pode copiar, alterar ou introduzir mensagens
  - O inimigo não pode fazer *replaying* de mensagens (*replaying* = reenvio)
  - O inimigo não pode reordenar as mensagens

Problemas do código móvel

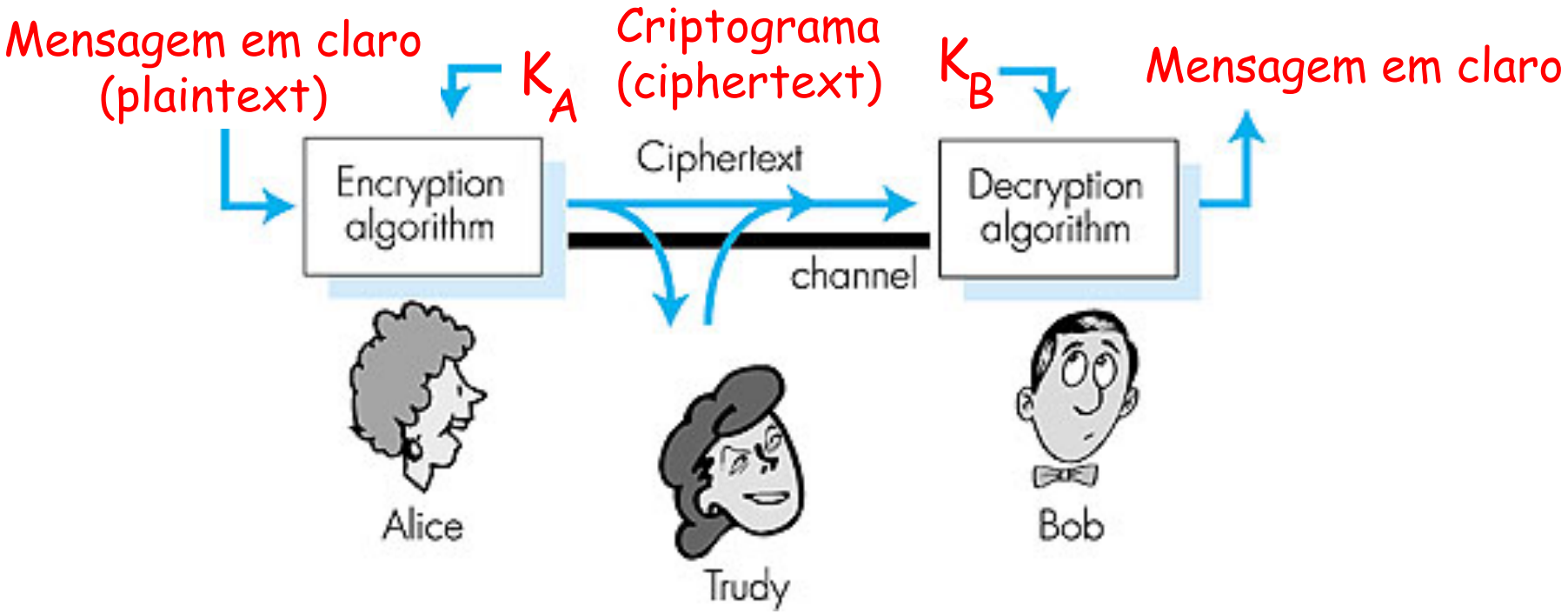
- O código obtido remotamente pode conter operações que comprometam a segurança de um sistema
- Como se pode minimizar o risco (exemplo Java VM):
  - Verificar que o código apenas contém operações legítimas
  - Manter o código obtido remotamente separado do código local
  - Associar permissões ao código obtido remotamente. As permissões dadas podem depender da origem do código, o que pode ser verificado através da assinatura do mesmo
    - Quando se executa uma operação que acede a um recurso do sistema (exemplo: *sockets*, ficheiros, etc.), são verificadas as permissões (de todas as classes do *stack*)
    - Possível porque todas estas operações são implementadas pelo biblioteca do sistema

Criptografia

- Criptografia é a disciplina que inclui os princípios, meios e métodos de transformação dos dados com a finalidade de:
  - esconder o seu conteúdo semântico
  - estabelecer a sua autenticidade
  - impedir que a sua alteração passe despercebida
  - evitar o seu repúdio
  - impedir a sua utilização não autorizada.

- Chave criptográfica é um parâmetro utilizado com um algoritmo criptográfico para transformar, validar, autenticar, cifrar ou decifrar dados.

A linguagem criptográfica



- Criptografia de chave simétrica : as chaves de cifra e de decifra são idênticas
- Criptografia de chaves assimétrica ou de chave pública : cifra-se com a chave pública, decifra-se com a chave privada do receptor, ou vice-versa

Notações mais frequentes

K_A	Alice's secret key
K_B	Bob's secret key
K_{AB}	Secret key shared between Alice and Bob
K_{privA}	Alice's private key (known only to Alice)
K_{pubA}	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

Criptografia clássica – ex.: código de César

- Normalmente baseiam-se em cifras de substituição
  - Uma cifra de substituição substitui um símbolo por outro
- Código de César é uma cifra de substituição para texto, considerando A=0,B=1,etc.
  - E(n,x) = (x+n) mod 26
  - D(n,x) = (x-n) mod 26
  - Ex.: para n = 3, têm-se as seguintes substituições

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC
E(3,"SISTEMAS DISTRIBUIDOS")="VLVWHPDV GLVWULEXLGRV"
```

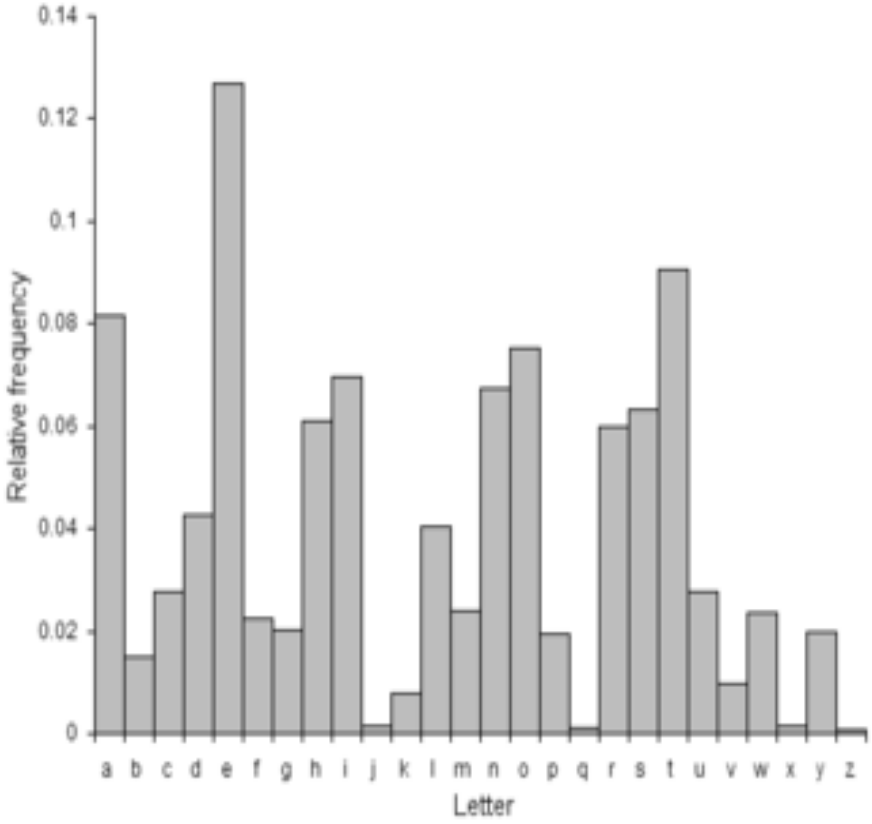
  - Usado por Júlio César para cifrar mensagens militares

Eficácia da criptografia

- O algoritmo criptográfico será tanto melhor quanto mais difícil for obter o texto original a partir do texto cifrado, sem se conhecer a chave. A eficácia de um ataque depende de dois factores:
  - Algoritmo de cifra
  - Cardinalidade do domínio da chave, isto é, número de bits da chave
- Métodos de ataque:
  - "Força bruta" - baseia-se na exploração sistemática de todas as chaves possíveis
  - Criptoanalíticos - baseia-se em explorar os métodos matemáticos utilizados em criptografia para descobrir como decifrar os dados (ou diminuir o número de possibilidades)

Criptoanálise – exemplo

- Análise de frequência: ideia aplicada ao código de César:
  - sabe-se que X é substituído sempre por Y
  - sabe-se a frequência do aparecimento dos diferentes símbolos no texto
  - O símbolo mais frequente no texto cifrado deve corresponder a um dos símbolos mais frequentes usualmente
- Se se souber algum texto da mensagem, pode-se usar essa informação para inferir parte da chave
  - Ataque à Enigma (WWII)



Eficácia da criptografia

- O algoritmo criptográfico será tanto melhor quanto mais difícil for obter o texto original a partir do texto cifrado, sem se conhecer a chave. A eficácia de um ataque depende de dois factores:
  - Algoritmo de cifra
  - Cardinalidade do domínio da chave, isto é, número de bits da chave
- Métodos de ataque:
  - "Força bruta" - baseia-se na exploração sistemática de todas as chaves possíveis
  - Criptoanalíticos - baseia-se em explorar os métodos matemáticos utilizados em criptografia para descobrir como decifrar os dados (ou diminuir o número de possibilidades)
- Nenhum algoritmo criptográfico é inteiramente seguro se o número de bits da chave tornar um ataque "força bruta" realista no quadro de dois factores:
  - O "valor" da informação
  - A capacidade computacional do atacante

Ataques força bruta

- Supondo que é possível arranjar mil milhões (10^3 x 10^6) de computadores, capazes de testarem mil milhões de chaves por segundo cada um, então é possível testar 10^18 chaves por segundo.
- Uma chave de 128 bits tem cerca de 3.4 x 10^38 combinações (10 levantado a 38). Com o poder computacional indicado, seriam necessários 10^13 anos para completar o ataque
  - Curiosidade: estima-se que a idade do universo é de cerca de 10^10 anos.
- Uma chave de 54 bits tem cerca de 6 x 10^16 combinações. Com o poder computacional indicado, seria necessário menos de um segundo para descobrir a chave.
- Conclusão: chaves geradas de forma "totalmente" aleatória e com um número de bits suficiente são relativamente seguras quando sujeitas a ataques do tipo "força bruta"

A teoria e a prática

- A análise anterior pressupõe que há uma distribuição perfeita da probabilidade de se ter seleccionado uma dada chave em todo o universo possível. Na prática tais geradores perfeitos de chaves não existem e o atacante pode, conhecendo alguma das fraquezas do gerador de chaves, ir analisar um espaço de pesquisa muito mais reduzido.
- Por outro lado, certas chaves são inadequadas e são facilmente quebradas conhecendo o algoritmo usado por isso não podem ser usadas.
- Ou seja, na prática, a geração de chaves adequadas é um problema muito delicado e os ataques de força bruta quase nunca são realizados de forma "cega" a todo o universo de chaves possíveis.

Confiança nos algoritmos criptográficos

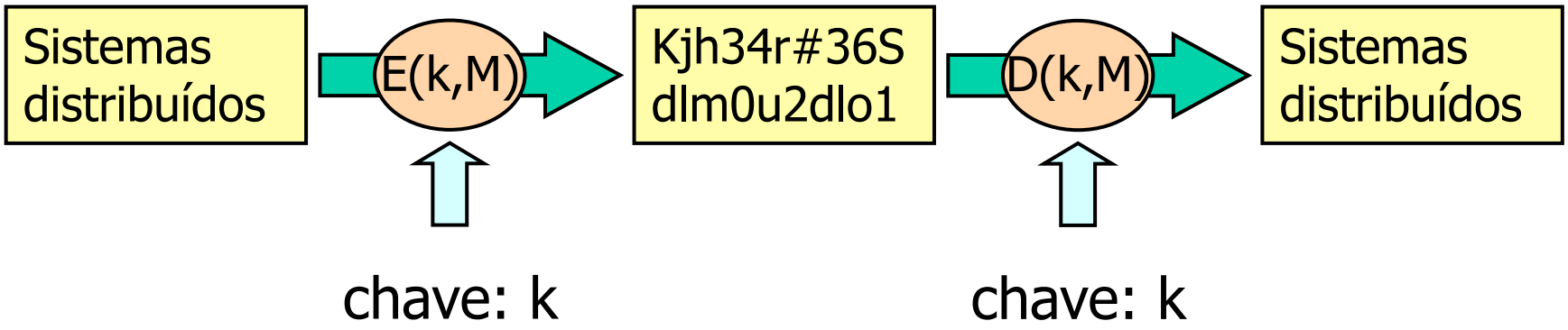
- Não é possível provar matematicamente que um algoritmo criptográfico não tem falhas capazes de o tornar frágil perante ataques criptoanalíticos. Só se consegue provar exactamente o contrário, através de exemplos.

- A segurança de um método é pois baseada em o mesmo ser público e ser sujeito à critica e análise por especialistas.
  - A segurança deve ser resultado do segredo das chaves e não do segredo do algoritmo criptográfico

- A ciência criptográfica é uma disciplina "especializada".

Criptografia simétrica

- Parceiros devem partilhar chave secreta
- Mensagem é cifrada e decifrada com a mesma chave
  - E(k,M) = X
  - D(k,X) = M
- Que garantias dá? (confidencialidade, autenticação?)
- Garante confidencialidade das mensagens
  - Dado X, deve ser computacionalmente impossível obter M sem saber K, mesmo conhecendo E e D



Algoritmos de criptografia simétrica mais comuns

- DES - Data Encryption Standard - chaves com 56 bits. História atribulada. Está a cair em desuso pois a chave é demasiado pequena para a potência de cálculo actual.

- Triple DES - DES reforçado - chave com 128 bits.
  - Aplica três vezes o algoritmo DES

- IDEA - *International Data Encryption Algorithm* - chave com 128 bits. Método desenvolvido na Europa. Após vários anos de utilização e divulgação pública não se conhecem ataques com êxito.
  - Baseado em álgebra de grupos: XOR, adições e multiplicações

- AES - *Advanced Encryption Standard* - Nova norma U.S.A. Definida por concurso público, ganho por uma equipa belga. Admite chaves de 128, 192 ou 256 bits.

DES - *Data Encryption Standard*

- Chave simétrica de 56 bits, blocos em claro e cifrados de 64 bits
- Qual o grau de seguranga do DES ?
  - DES Challenge: uma frase cifrada com uma chave de 56 bits ("Strong cryptography makes the world a safer place") foi decifrada (por força bruta) em 4 meses (cooperação internacional - 1997), e em 22 horas (1999)
    - http://en.wikipedia.org/wiki/DES_Challenges
  - não existem "backdoors" conhecidos
- Pode ser tornada mais segura
  - aplicando várias chaves sequencialmente em cada bloco (3-DES)
  - usando *cipher-block chaining* (ver mais à frente)

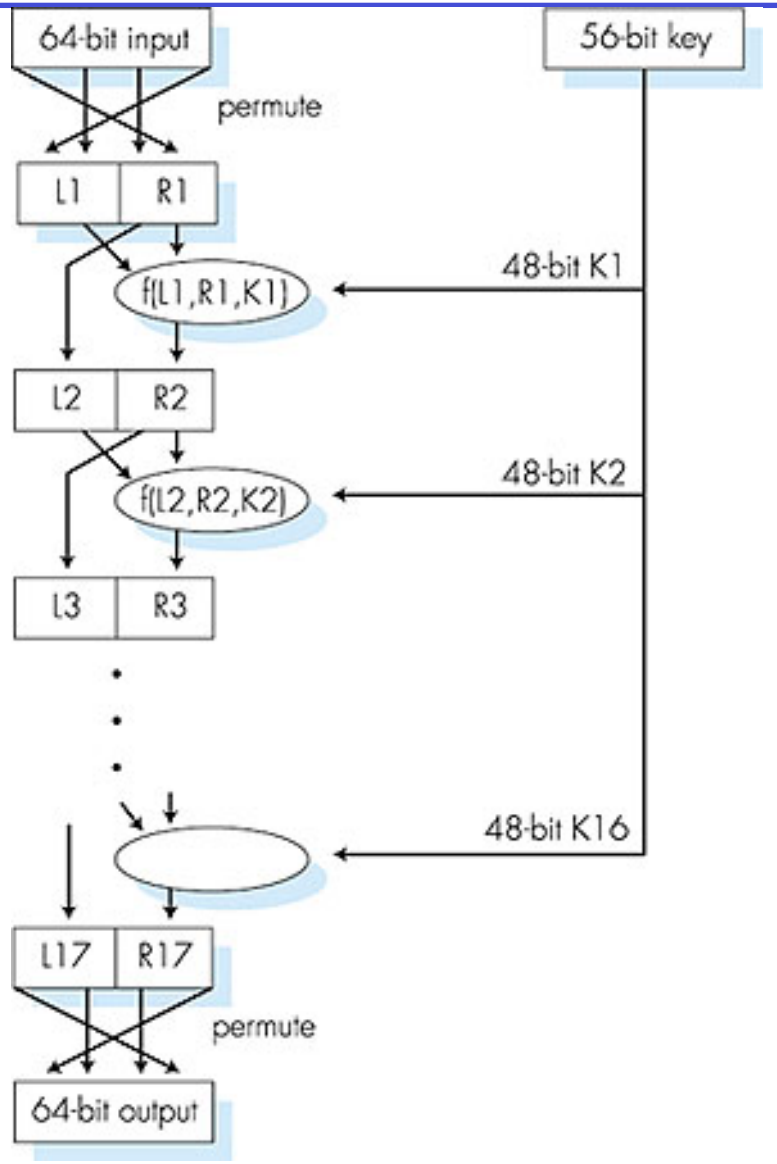
Funcionamento do DES

Operação do DES

Permutação inicial

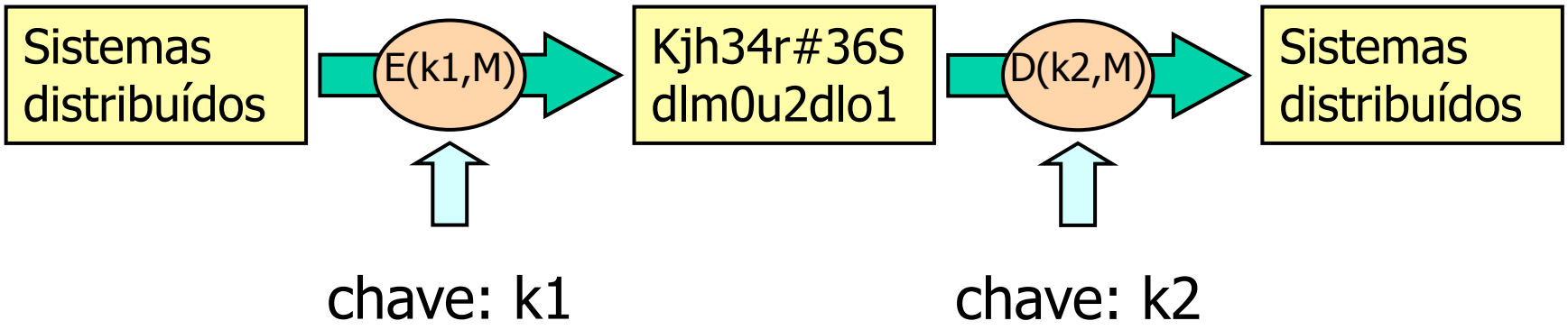
16 "rounds" idênticos de aplicação de uma função, cada um dos quais usa 48 bits diferentes da chave

Permutação final



Criptografia assimétrica

- Cada entidade tem duas chaves
  - Chave privada (Kpriv) é conhecida apenas pelo seu dono
  - Chave pública (Kpub) pode ser conhecida por todos
  - A partir de Kpub é impossível derivar Kpriv
- Mensagem é cifrada com uma chave e decifrada com a outra
- E(Kpub,M) = X; D(Kpriv,X) = M
  - Garante o quê? (confidencialidade, autenticação?)
- E(Kpriv,M) = X; D(Kpub,X) = M
  - Garante o quê? (confidencialidade, autenticação?)



Criptografia assimétrica

- Cada entidade tem duas chaves
  - Chave privada (Kpriv) é conhecida apenas pelo seu dono
  - Chave pública (Kpub) pode ser conhecida por todos
  - A partir de Kpub é impossível derivar Kpriv
- Mensagem é cifrada com uma chave e decifrada com a outra
- E(Kpub,M) = X; D(Kpriv,X) = M
  - Garante confidencialidade
    - Conhecendo Kpub e X deve ser computacionalmente impossível obter M sem saber Kpriv. Só receptor conhece Kpriv.
- E(Kpriv,M) = X; D(Kpub,X) = M
  - Garante autenticação do emissor
    - A partir de Kpub deve ser computacionalmente impossível obter Kpriv. Só quem possui Kpriv pode gerar X.

Algoritmos de criptografia assimétrica mais comuns

- RSA – algoritmo mais usado. Chave mínima recomendada: 768 bits.
  - Patente RSA expirou em 2000

- Algoritmos de curva elíptica – método para gerar pares de chaes pública/privada baseado nas propriedades das curvas elípticas. Usa chaves de dimensão menor.
  - Não está dependente da factorização de números

Curiosidade: Algoritmo RSA (Rivest, Shamir e Adelman)

RSA Challenge (640 bits quebrado)
http://www.rsa.com/rsalabs/node.asp?id=2093

- Para gerar as chaves
  - Escolhem-se dois números primos *grandes* , P e Q (P e Q > 10^{100})
    - N = P*Q, Z=(P-1) *(Q-1)
  - Escolhe-se *d*, que pode ser qualquer número menor que N que seja primo em relação a Z
  - Calcula-se *e*, que é um número tal que *e.d-1* é divisível por Z
  - A chave privada é (N,e); a chave pública é (N,d)
- A função de cifra é
  - E((e,N), M) = M^e mod N = c
- A função de decifra é
  - D((d,N), c) = c^d mod N = M

- Exemplo:

  P=5
  Q=11
  N=55
  Z=40

  d=3
  e=7

  E(2)= 2^7 mod 55
  = 18
  D(18)=18^3 mod 55
  = 2

NOTA: Segurança do método depende da dificuldade de factorizar N em P e Q

Funções de síntese

- Objectivo: uma função de síntese segura H (*Message Digest* ou *Secure Hash*) deve produzir um sequência pequena de bits (128,160,512,...) que permita identificar uma mensagem de qualquer dimensão
- Propriedades:
  1. Calcular H(M) é fácil (*computacionalmente*)
  2. Dado H(M) é *computacionalmente* impossível calcular M
  3. Dado M é *computacionalmente* impossível descobrir M2 tal que H(M) = H(M2)
- As funções de síntese com estas propriedades dizem-se "*Secure one-way hash functions*" ou "funções de dispersão unidireccionais seguras".
  - Porquê unidireccional e seguro?
- Para que é que serve?
  - Usado para garantir integridade dos dados

Funções seguras de síntese

- Função segura de síntese MD5
  - Calcula sínteses de 128 bits num processo em 4 fases
  - Uma das funções seguras de síntese computacionalmente mais eficientes
    - Conhecidos ataques que permitem gerar colisões.

- A função SHA-1 é também muito usada.
  - Norma USA. Conhecida publicamente
    - Conhecidos ataques que permitem diminuir o espaço de pesquisa para uma colisão para $O(2^{63})$ (em vez de (2^{80}))
    - Está-se a migrar para SHA-2
  - Produz uma síntese de 160 bits

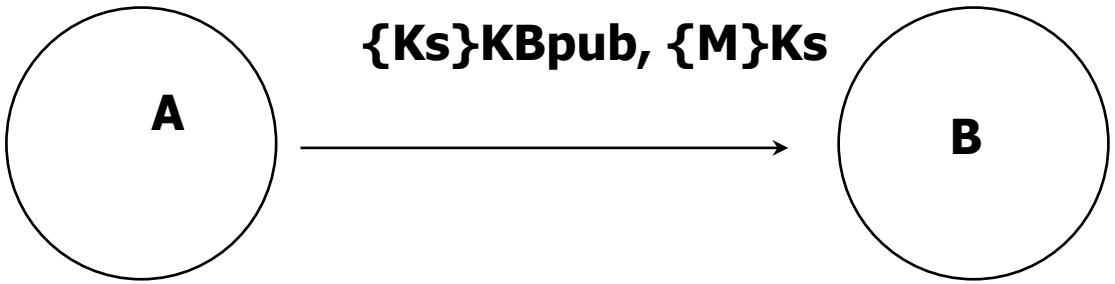
O desempenho dos diferentes métodos

	Key size/hash size (bits)	*Extrapolated speed (kbytes/sec.)*	*PRB optimized (kbytes/s)*
TEA	128	700	-
DES	56	350	7746
Triple-DES	112	120	2842
IDEA	128	700	4469
RSA	512	7	-
RSA	2048	1	-
MD5	128	1740	62425
SHA	160	750	25162

Ano: 1999

Utilização conjunta dos métodos

- Os algoritmos utilizados pelos métodos baseados em criptografia assimétrica são 100 a 1000 vezes mais pesados que os baseados em criptografia simétrica. Por esta razão, os dois métodos são utilizados em conjunto. Como?
- As chaves públicas são utilizadas para autenticação e para passar uma chave de sessão para utilização posterior de criptografia simétrica entre os dois principais.
- Exemplo:



Exposição das chaves privadas assimétricas

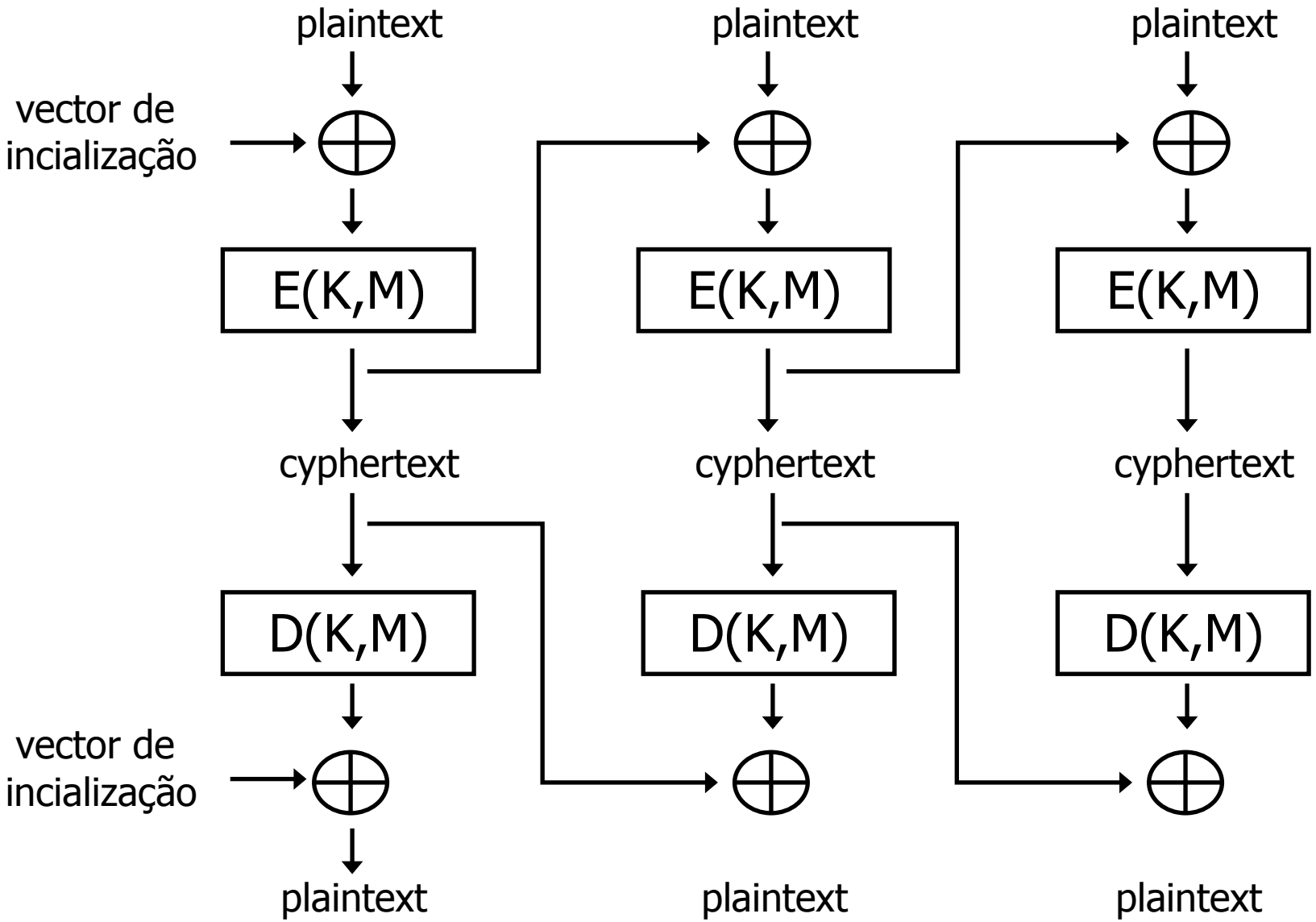
- Se as chaves públicas e privadas apenas forem usadas para a autenticação e troca de uma chave simétrica de sessão com o outro parceiro, então:
  - 1) as chaves públicas estão sempre expostas, mas
  - 2) as chaves privadas só são usadas para decifrar as mensagens iniciais devendo ser apagadas da memória imediatamente a seguir
  - 3) a chave de sessão tem a validade dessa sessão e nunca mais é reutilizada

- As chaves privadas assimétricas são geralmente de grande dimensão pelo que não é prático memorizá-las; devem ser guardadas em suportes estáveis onde estão cifradas por sua vez através de uma palavra chave (sendo a palavra chave a chave de um processo simétrico de cifra da chave secreta).
- Em alternativa a chave privada pode estar gravada num cartão inteligente (*smartcard*) que cifra / decifra sem expor a chave secreta.

Cifra por blocos encadeados

- Um algoritmo de cifra por blocos pode revelar-se frágil na medida em que cada bloco é cifrado de forma independente, o que pode revelar padrões repetidos que facilitem a relação do texto cifrado com o texto em claro e facilita o ataque por métodos cripto-analíticos.
- Uma técnica possível para melhorar o método é usar cifra por blocos encadeados (CBC - *cipher block chaining*)

Cifra por blocos encadeados



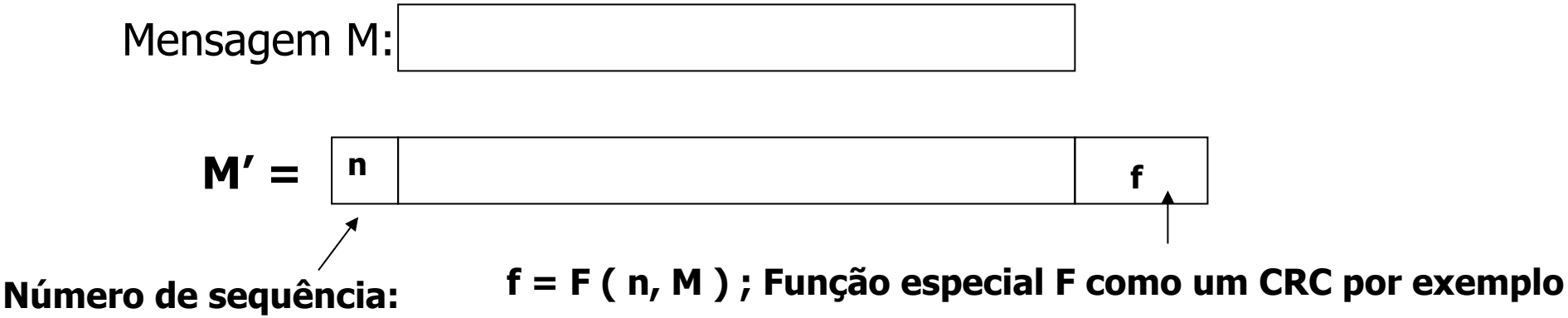
Cifra por blocos encadeados

- Um algoritmo de cifra por blocos pode revelar-se frágil na medida em que cada bloco é cifrado de forma independente, o que pode revelar padrões repetidos que facilitem a relação do texto cifrado com o texto em claro e facilita o ataque por métodos cripto-analíticos.
- Uma técnica possível para melhorar o método é usar cifra por blocos encadeados (CBC - *cipher block chaining*)
  - Durante o processo de cifra, antes de cifrar um blocos, faz-se XOR com o resultado da cifra anterior
  - Ao decifrar, após decifrar um bloco faz-se XOR com o bloco anterior cifrado
  - No início usa-se um vector de inicialização (por exemplo uma *timestamp* da dimensão de um bloco) para iniciar o processo. Desta forma, o mesmo texto, cifrado com a mesma chave, mas com vectores iniciais distintos, conduz a resultados distintos.
- Esta técnica só se pode aplicar em canais fiáveis pois a perca de uma mensagem impede o processo de decifra.

Como evitar que o receptor fique "baralhado"

- Se as mensagens a cifrar só contiverem bits significativos e válidos para as transacções em curso, a sua decifra com uma chave errada conduz, apesar disso, a um padrão de bits que pode ter significado (errado) para o receptor
- Problema?
- Este facto pode ser usado para levar o receptor a executar "disparates" o que pode ser uma fonte de insegurança
  - Pode ser usado para ataques de impedimento de prestação de serviço tentando saturar o receptor a processar dados aleatórios.
- Solução?
- Por este motivo, é necessário introduzir bits redundantes antes de cifrar as mensagens que impeçam o receptor de ficar "baralhado". As soluções mais simples são:
  - colocar um CRC
  - colocar um número de ordem
  - colocar um valor fixo pré-estabelecido, etc.

Exemplo



- M' é a mensagem que é cifrada e transmitida. O receptor recebe M' cifrada e decifra o primeiro bloco. Começa logo por controlar se o mesmo contem um número de sequência válido. Se não contiver, ignora o resto da mensagem; senão considera-a mas volta a controlar o valor de f no fim.

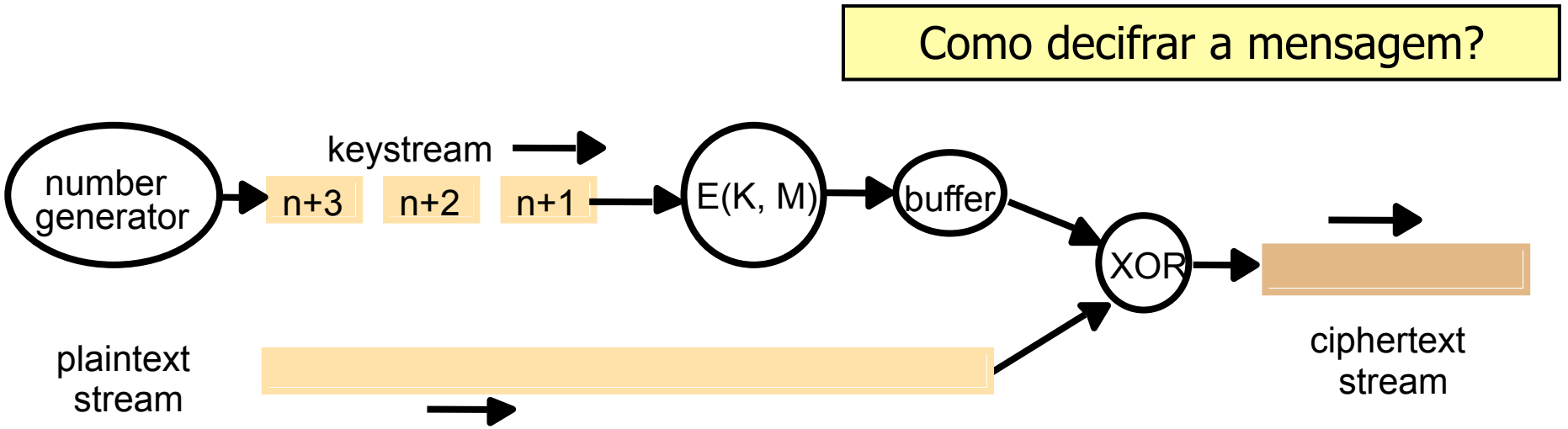
Controlo de integridade com funções de síntese seguras

- É possível usar as funções de síntese para controlar a integridade num canal de dados não seguro. O método consiste em usar MACs ("*Message Authentication Codes*") que são assinaturas, computacionalmente fáceis de calcular, baseadas em chaves secretas. O método funciona assim:
  1. A e B estabelecem uma chave secreta K só conhecida por ambos. K pode ser trocada aquando da autenticação, por exemplo.
  2. Para A enviar a mensagem M a B:
     - Calcula h = H(M+K)
     - Envia M,h
  3. Ao receber "M,h", B calcula h'=H(M+K) e verifica integridade da mensagem se h=h'
     - Porque é que garante autenticação e integridade?
     - Só A conhece K, logo só A consegue gerar H(M+K) – autentica emissor de M e garante que a mensagem não foi modificada

- NOTA: Não se pretende garantir confidencialidade

Cifra de streams

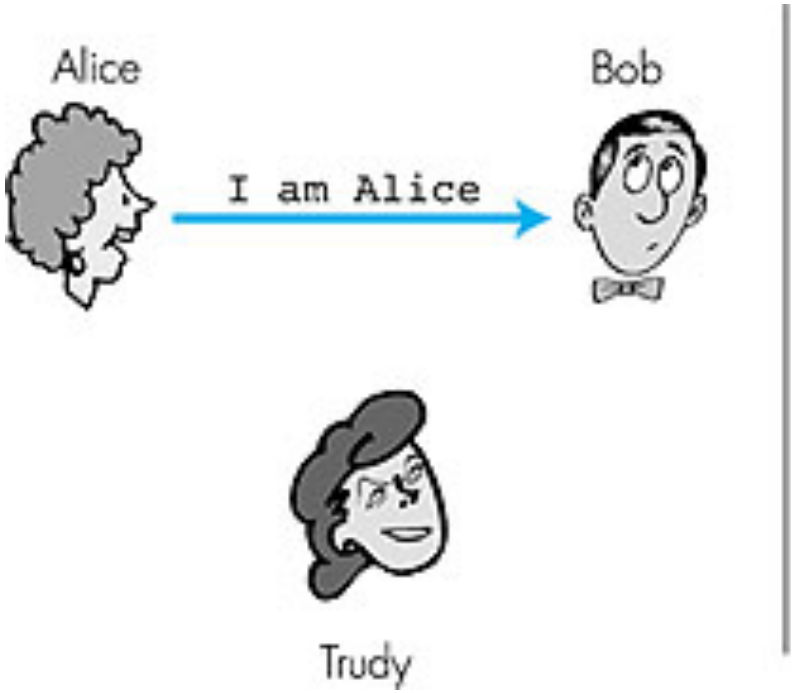
- Um algoritmo de cifra por blocos apenas pode cifrar bloco a bloco
- Quando se pretende cifrar dados de dimensão inferior a um bloco é necessário preparar um bloco que contenha os dados a enviar (padding) e cifrá-lo
- Alternativamente pode-se obfuscar a mensagem a transmitir usando um stream (keystream) criado a partir de um gerador de números aleatórios (fazendo XOR)



Autenticação

- Objectivo: Bob quer que a Alice lhe "demonstre" a sua identidade

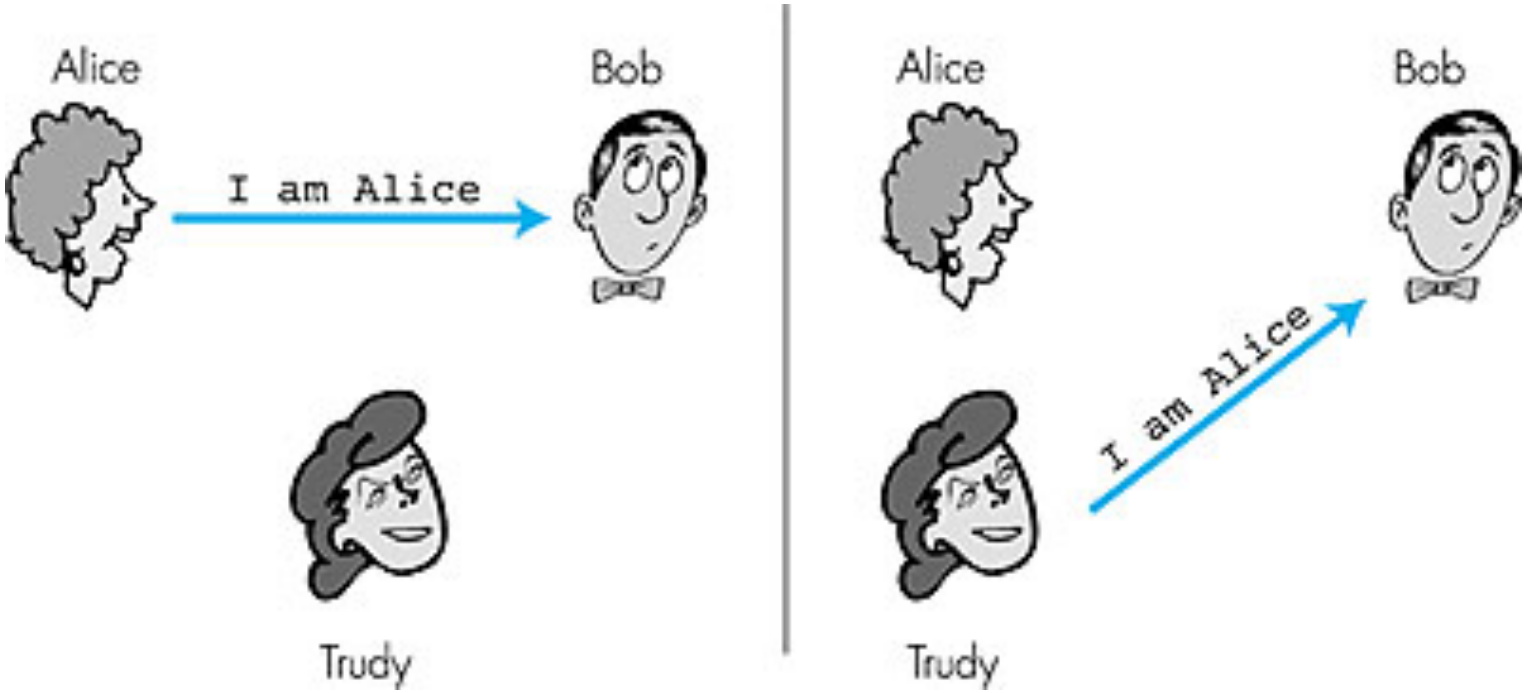
Protocolo 1: Alice diz simplesmente "I am Alice"



Autenticação

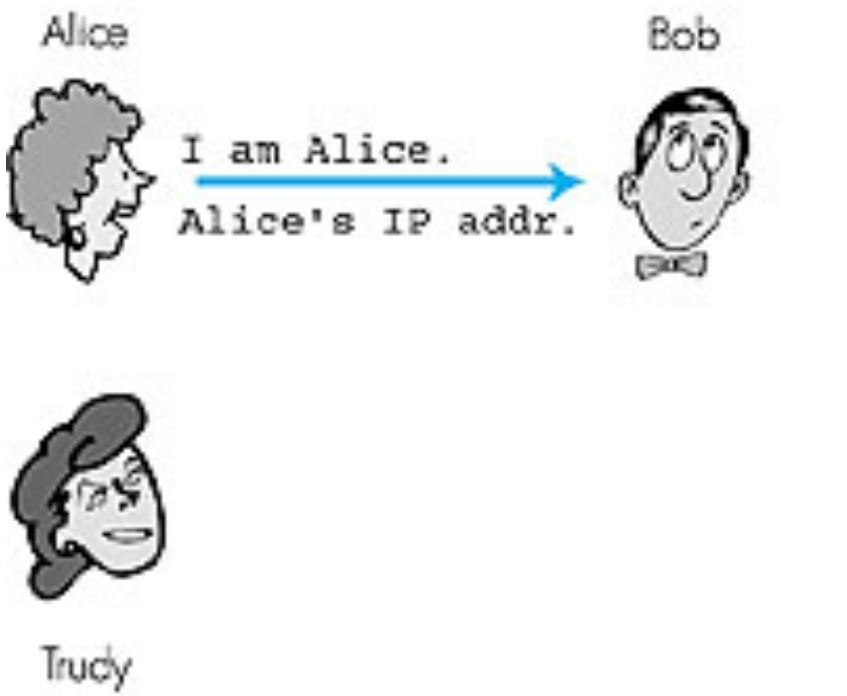
- Objectivo: Bob quer que a Alice lhe "demonstre" a sua identidade

Protocolo 1: Alice diz simplesmente "I am Alice"



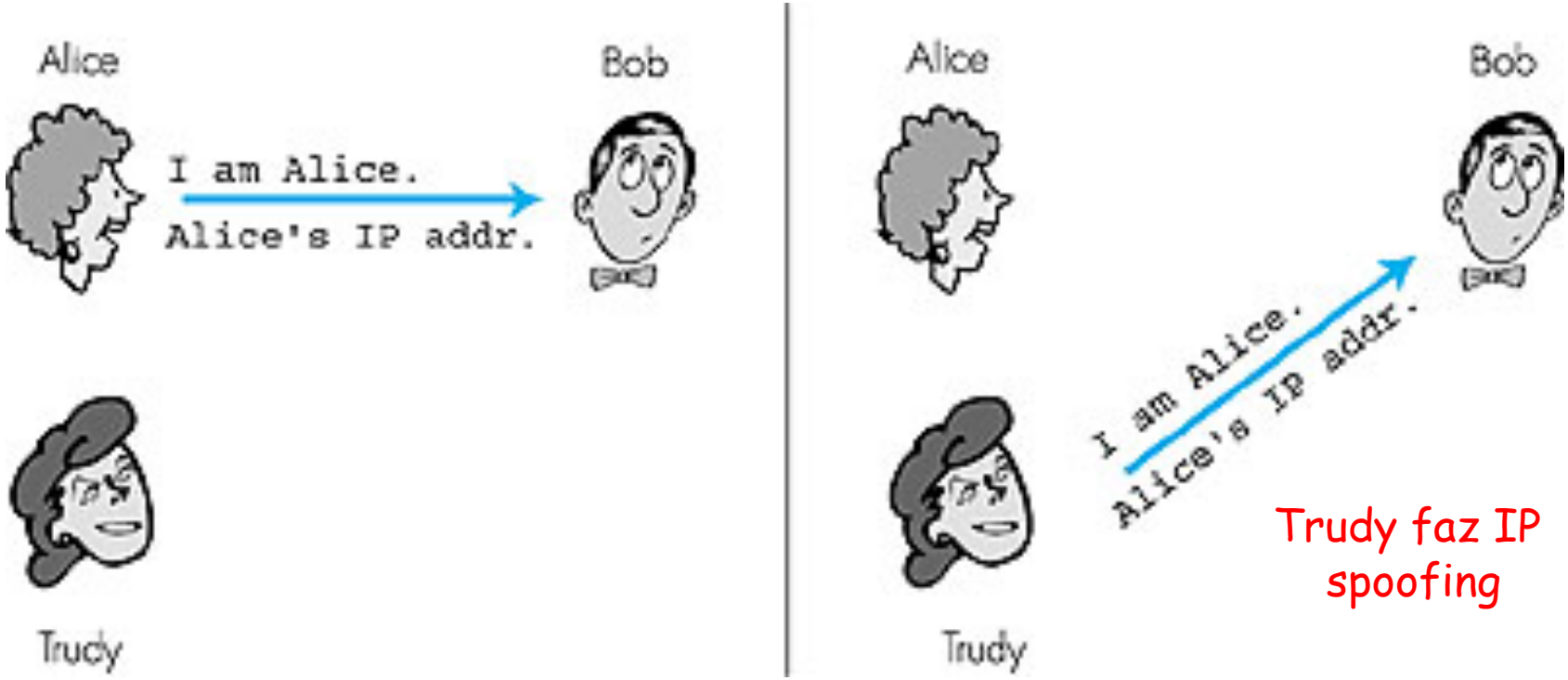
Segunda tentativa

Protocolo 2: Alice diz "I am Alice" e envia o seu endereço IP para o provar.



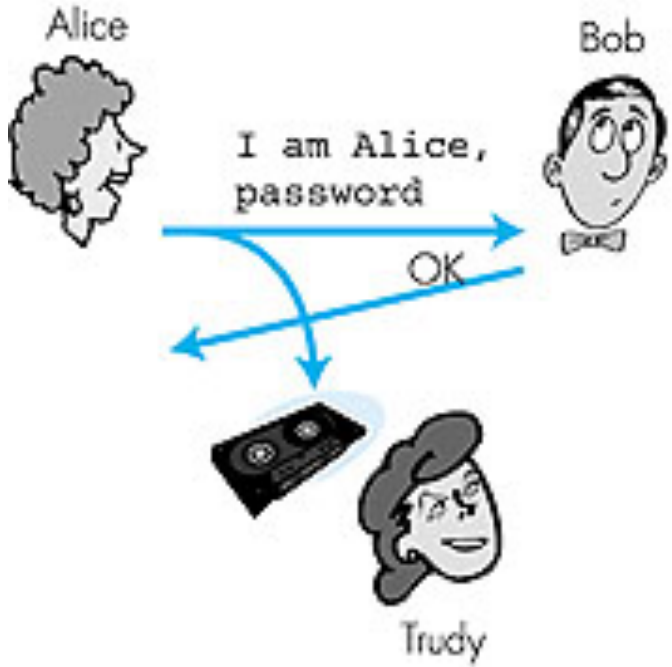
Segunda tentativa

Protocolo 2: Alice diz "I am Alice" e envia o seu endereço IP para o provar.



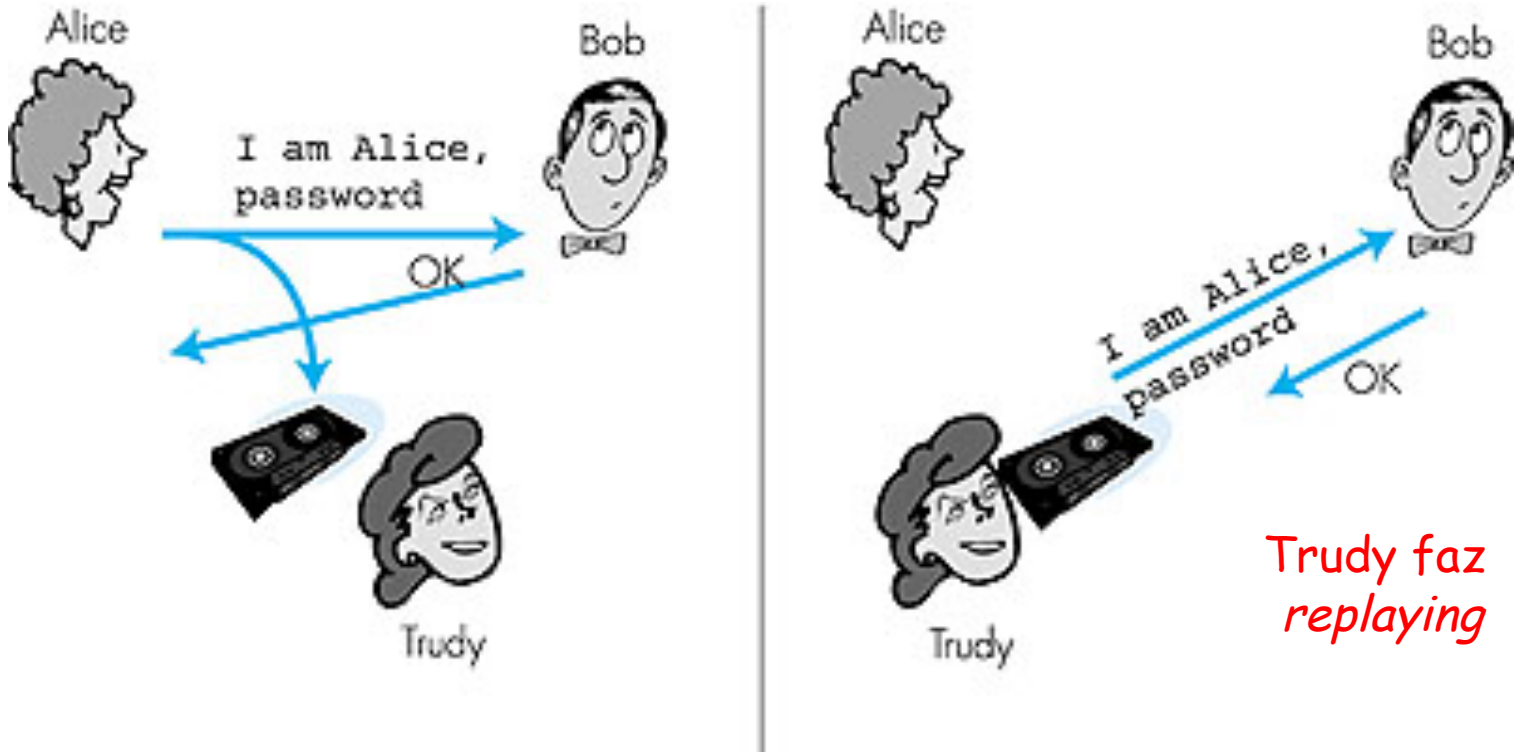
Terceira tentativa

Protocolo 3: Alice diz "I am Alice" e envia também a sua "password"



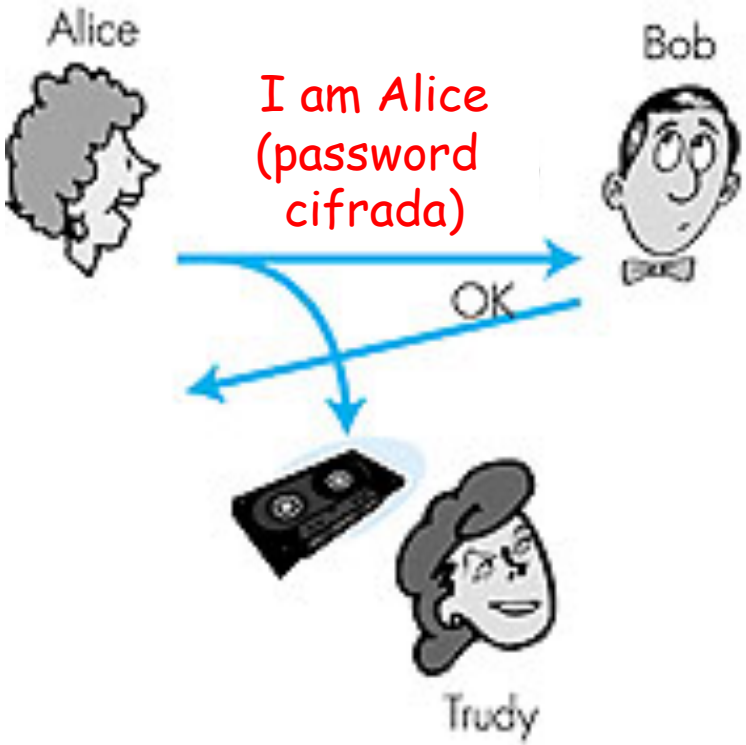
Terceira tentativa

Protocolo 3: Alice diz "I am Alice" e envia também a sua "password"



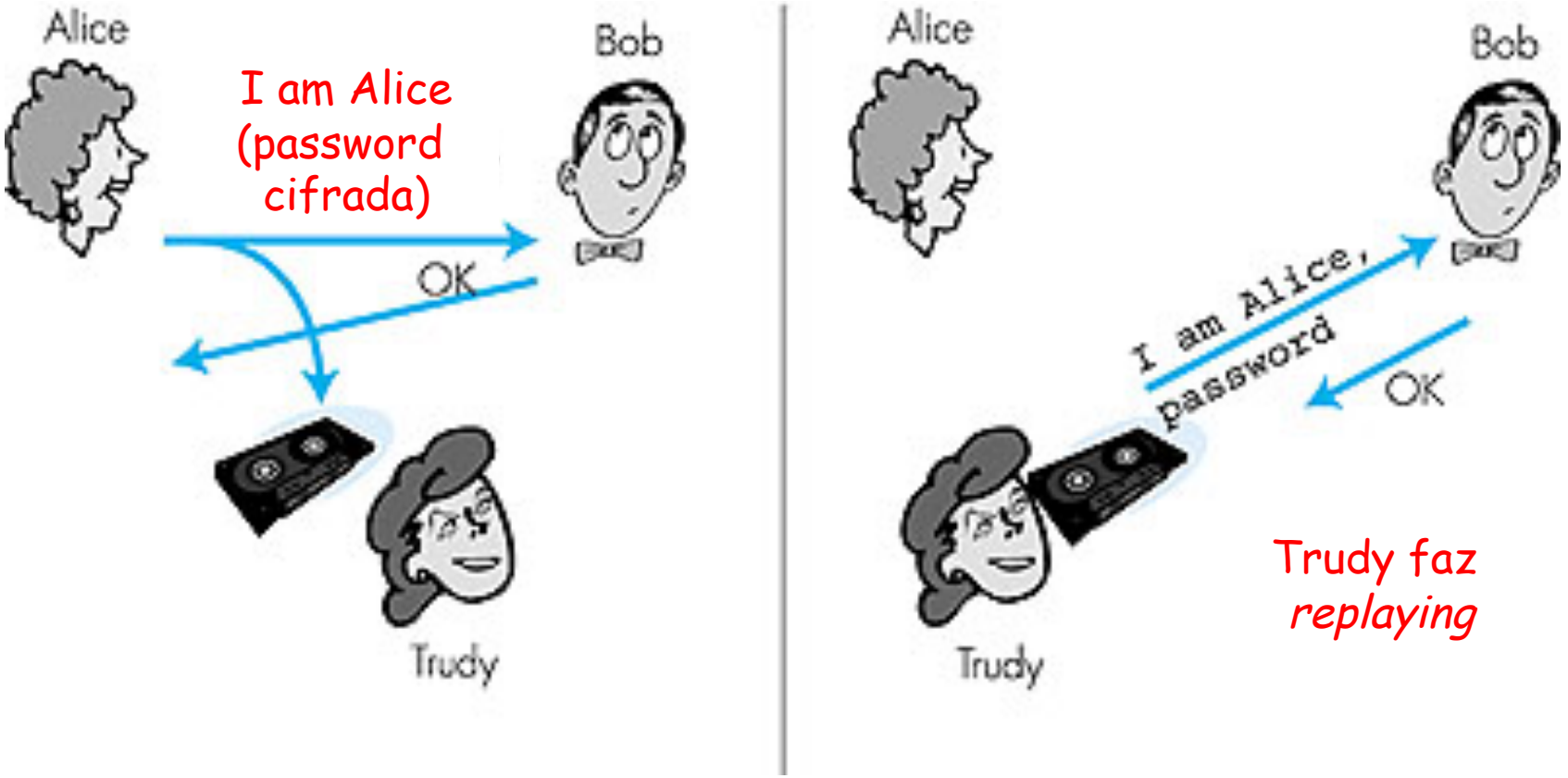
Variante da terceira tentativa

Protocolo 3': Alice diz "I am Alice" e envia também a sua "password" cifrada com uma chave comum a ambos



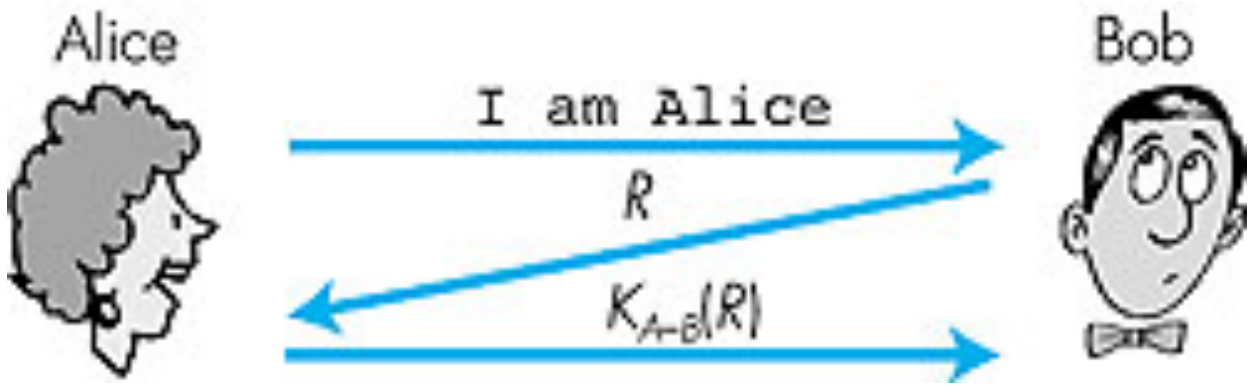
Variante da terceira tentativa

Protocolo 3': Alice diz "I am Alice" e envia também a sua "password" cifrada com uma chave comum a ambos



Método desafio / resposta

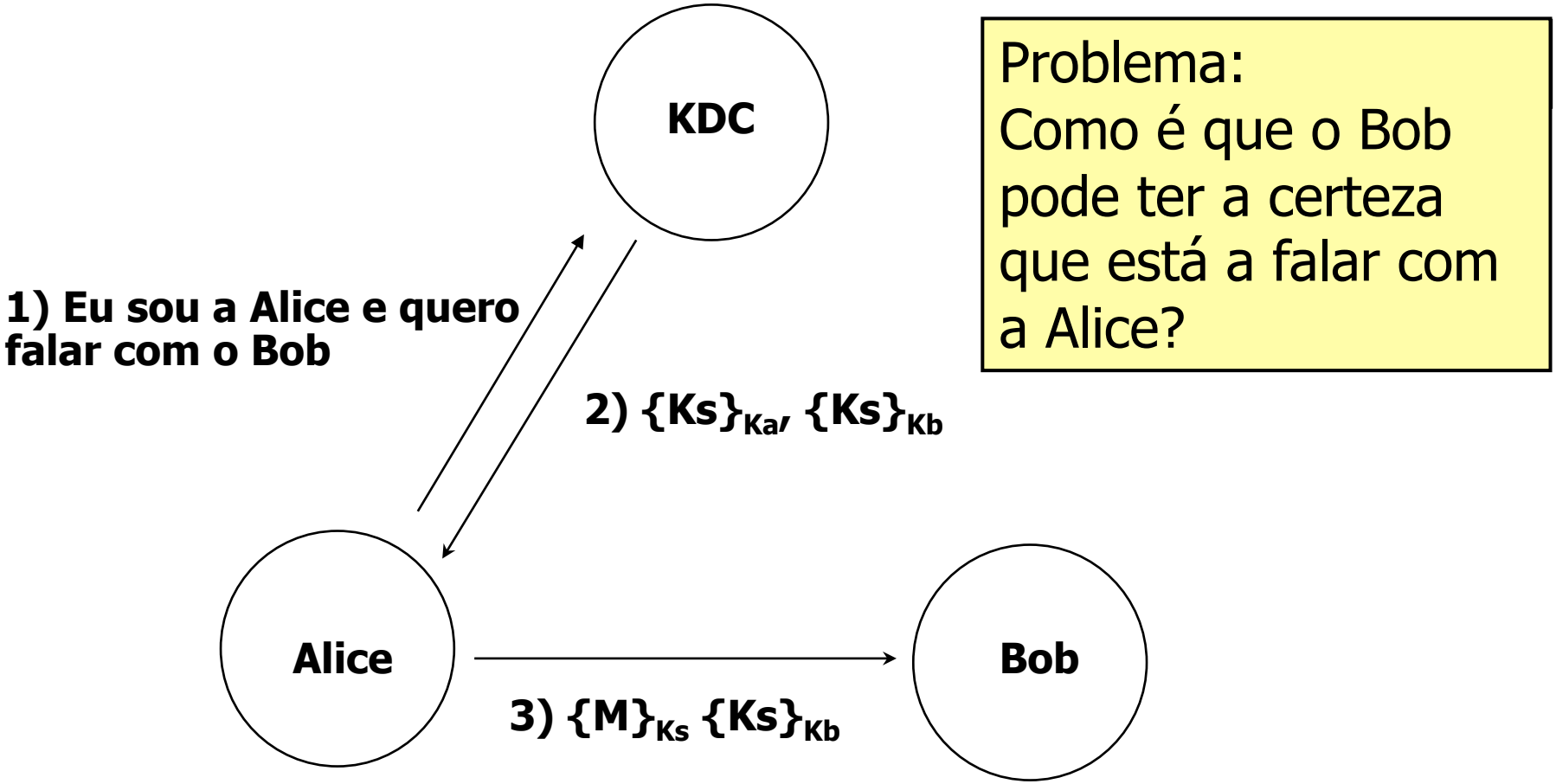
- Objectivo: evitar o ataque por *replaying*
- Nonce: número usado uma única vez
- Protocolo 4: para garantir a "frescura" da transacção, Bob envia à Alice o *nonce*, R. Alice deve devolver R cifrado com a chave secreta que ambos partilham.



Problemas e possíveis soluções

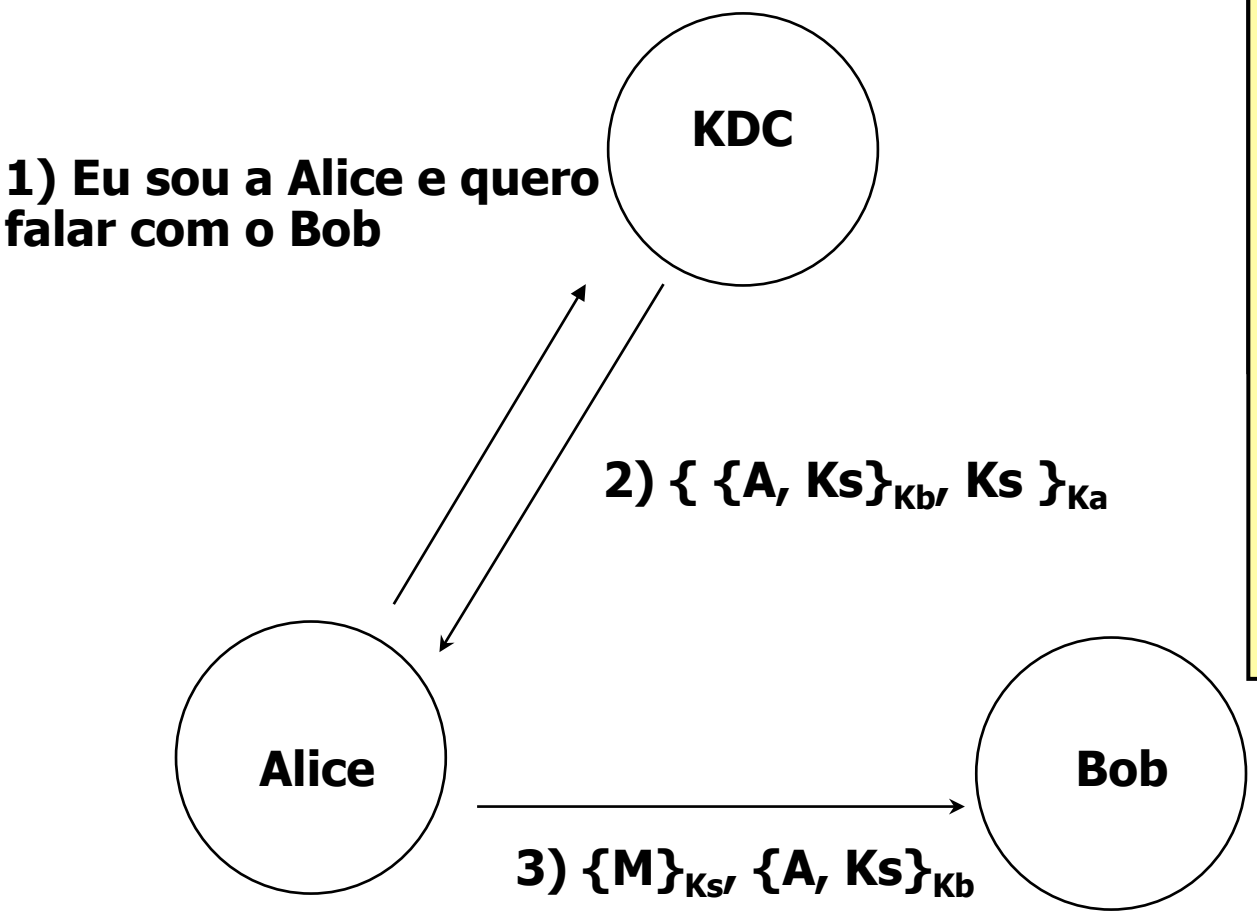
- Apesar de a chave secreta não necessitar de passar na rede durante a autenticação, Alice e Bob têm de arranjar alguma forma de se porem previamente de acordo sobre a chave secreta que vão usar.
  - Dado que tal chave não pode ser passada na rede, terão de se encontrar ou usar uma terceira pessoa para trocarem a chave.
  - Se isso se revelar um método complexo, vão ter tendência a manter a mesma chave durante muito tempo, o que é perigoso.
- **Solução**: usar uma chave diferente em cada sessão e usar um centro de distribuição de chaves no qual Alice e Bob confiam

Distribuição de chaves de sessão através de um KDC – primeira tentativa



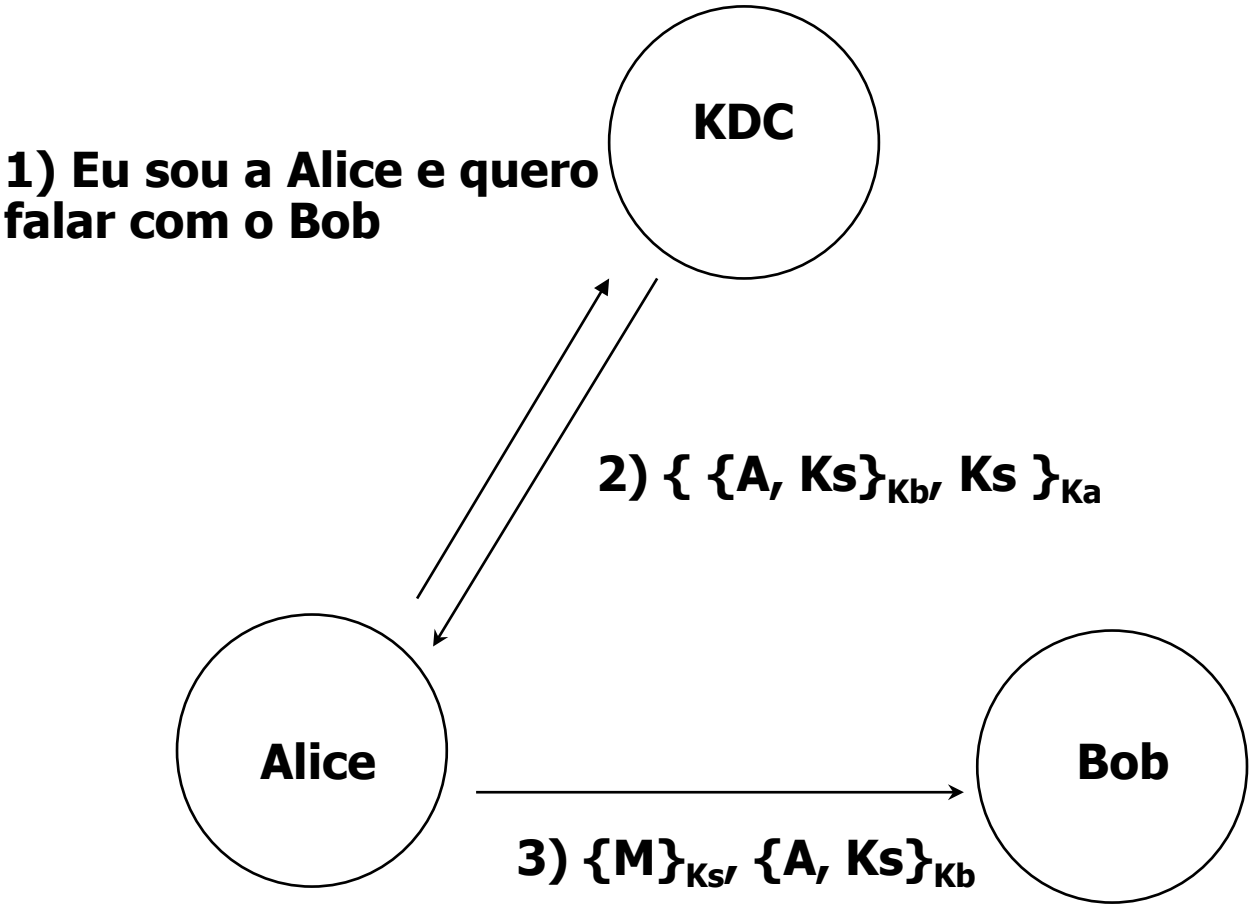
Problema:
Como é que o Bob pode ter a certeza que está a falar com a Alice?

Distribuição de chaves de sessão através de um KDC – segunda tentativa



Problema:
Como é que Bob pode ter a certeza que está a falar com Alice?
Por Alice ter sido capaz de obter $\{A,Ks\}K_b$. Só Alice conhece K_a

Distribuição de chaves de sessão através de um KDC – segunda tentativa



Problema:
Pode Trudy obter o conteúdo da mensagem?
Não, porque apenas Bob conhece K_b. Mas Trudy pode gravar a mensagem e mais tarde tentar incomodar Bob.

Protocolo de *Needham-Schroeder*

- Este protocolo permite a dois principais A e B (Alice e Bob) estabelecerem um canal seguro entre si e autenticarem-se mutuamente.

- Baseia-se na presença de um KDC (Key Distribution Center) que conhece as chaves secretas de A e B (Ka e Kb) e que é capaz de gerar chaves de sessão (Ks) que vão ser usadas por A e B para comunicarem de forma segura.

- O protocolo resiste aos ataques *eavesdropping*, *masquerading*, *message tampering* e *replaying*.

O protocolo NS com chaves simétricas

1) A -> KDC: **A, B, Na**

- *A* solicita uma chave para comunicar com *B*, sendo *Na* um número aleatório gerado por *A* para garantir a unicidade da transacção (*Na* deve ser único).

2) KDC -> A: **{ Na, B, Ks, { Ks, A }Kb }Ka**

- O *KDC* devolve Na, a chave e um *ticket* ({ Ks, A }Kb), tudo cifrado com a chave de A.

3) A -> B: **{Ks, A}K**b**, {Na'}Ks**

- *A* solicita comunicar com B, enviando-lhe o *ticket*

4) B -> A: **{Na'-1}Ks**

O protocolo NS com chaves simétricas

1) A -> KDC: **A, B, Na**

- *A* solicita uma chave para comunicar com *B*, sendo *Na* um número aleatório gerado por *A* para garantir a unicidade da transacção (*Na* deve ser único).

2) KDC -> A: **{ Na, B, Ks, { Ks, A }Kb }Ka**

- O *KDC* devolve Na, a chave e um *ticket* ({ Ks, A }Kb), tudo cifrado com a chave de A.
- **O que garante a A ter falado com o KDC ?**

3) A -> B: **{Ks, A}Kb, {Na'}Ks**

- *A* solicita comunicar com B, enviando-lhe o *ticket*

4) B -> A: **{Na'-1}Ks**

O protocolo NS com chaves simétricas

1) A -> KDC: A, B, Na

- *A* solicita uma chave para comunicar com *B*, sendo *Na* um número aleatório gerado por *A* para garantir a unicidade da transacção (*Na* deve ser único).

2) KDC -> A: { Na, B, Ks, { Ks, A }K_b }K_a

- O *KDC* devolve Na, a chave e um *ticket* ({ Ks, A }Kb), tudo cifrado com a chave de A.
- A recepção do *Na* único garante que A comunicou com o KDC – só o KDC conhece Ka, logo só KDC pode gerar a mensagem indicada.

3) A -> B: {Ks, A}Kb, {Na'}Ks

- *A* solicita comunicar com B, enviando-lhe o *ticket*
- **O que garante a B estar a falar com A ?**

4) B -> A: {Na'-1}Ks

O protocolo NS com chaves simétricas

1) A -> KDC: A, B, Na

- *A* solicita uma chave para comunicar com *B*, sendo *Na* um número aleatório gerado por *A* para garantir a unicidade da transacção (*Na* deve ser único).

2) KDC -> A: { Na, B, Ks, { Ks, A }Kb }Ka

- O *KDC* devolve Na, a chave e um *ticket* ({ Ks, A }Kb), tudo cifrado com a chave de A.
- A recepção do *Na* único garante que A comunicou com o KDC – só o KDC conhece Ka, logo só KDC pode gerar a mensagem indicada.

3) A -> B: {Ks, A}Kb, {Na'}Ks

- *A* solicita comunicar com B, enviando-lhe o *ticket*
- B sabe que está a falar com A, porque apenas A pode obter Ks associada a um ticket indicando A (porque Ks passa cifrado com Ka em 2)). Além disso, é impossível forjar um ticket, porque só B e KDC conhecem Kb

4) B -> A: {Na'-1}Ks

- **O que garante a A estar a falar com B ?**

O protocolo NS com chaves simétricas

1) A -> KDC: A, B, Na

- *A* solicita uma chave para comunicar com *B*, sendo *Na* um número aleatório gerado por *A* para garantir a unicidade da transacção (*Na* deve ser único).

2) KDC -> A: { Na, B, Ks, { Ks, A }Kb }Ka

- O *KDC* devolve Na, a chave e um *ticket* ({ Ks, A }Kb), tudo cifrado com a chave de A.
- A recepção do *Na* único garante que A comunicou com o KDC – só o KDC conhece Ka, logo só KDC pode gerar a mensagem indicada.

3) A -> B: {Ks, A}Kb, {Na'}Ks

- *A* solicita comunicar com B, enviando-lhe o *ticket*
- B sabe que está a falar com A, porque apenas A pode obter Ks associada a um ticket indicando A (porque Ks passa cifrado com Ka em 2)). Além disso, é impossível forjar um ticket, porque só B e KDC conhecem Kb

4) B -> A: {Na'-1}Ks

- B prova ser B por ser capaz de decifrar {Na'}Ks, o que pressupõe ter sido capaz de decifrar {Ks, A}Kb
- **Porque não pode B enviar {Na'}Ks ?**

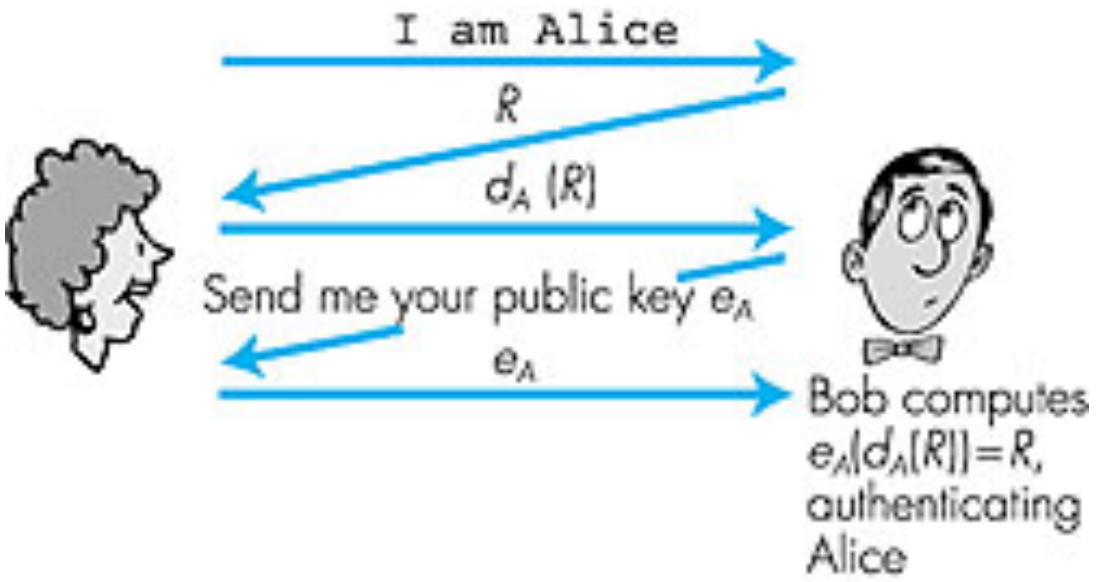
Um possível ataque e respectiva solução

- A mensagem 3) pode ser *replayed* mais tarde. Isso, em princípio, não tem problema mas se por acaso Ks for um dia comprometida, então Trudy pode conseguir autenticar-se perante Bob como se fosse Alice. A solução consiste em acrescentar uma *timestamp* ao *ticket* que deve ser testada por Bob após ter recebido a mensagem 3). Tal exige que os relógios de A, B e do KDC estejam sincronizados a menos de um valor considerado suficiente para tornar impossível a quebra da chave Ks.
- O protocolo fica:
- 1) A -> KDC: A, B, Na
- 2) KDC -> A: { Na, B, Ks, { Ks, A, t }Kb }Ka
- 3) A -> B: {Ks, A, t}K_b, {Na'}Ks
- 4) B -> A: {Na'-1}Ks

Autenticação via chave pública

Problema: como é que Bob e Alice se autenticam utilizando chaves assimétricas?

Resposta: usa um *nonce*, e criptografia assimétrica



Protocolo de Needham-Schroeder com chaves públicas

- Pressupondo que A e B conhecem as chaves públicas um do outro de forma certificada, podem estabelecer um canal seguro e autenticarem-se mutuamente através de:

1. **A -> B:** **{ A, Na }KBpub**
2. **B -> A:** **{ Na, Nb, Ks } KApub**
3. **A -> B:** **{ Nb } Ks**

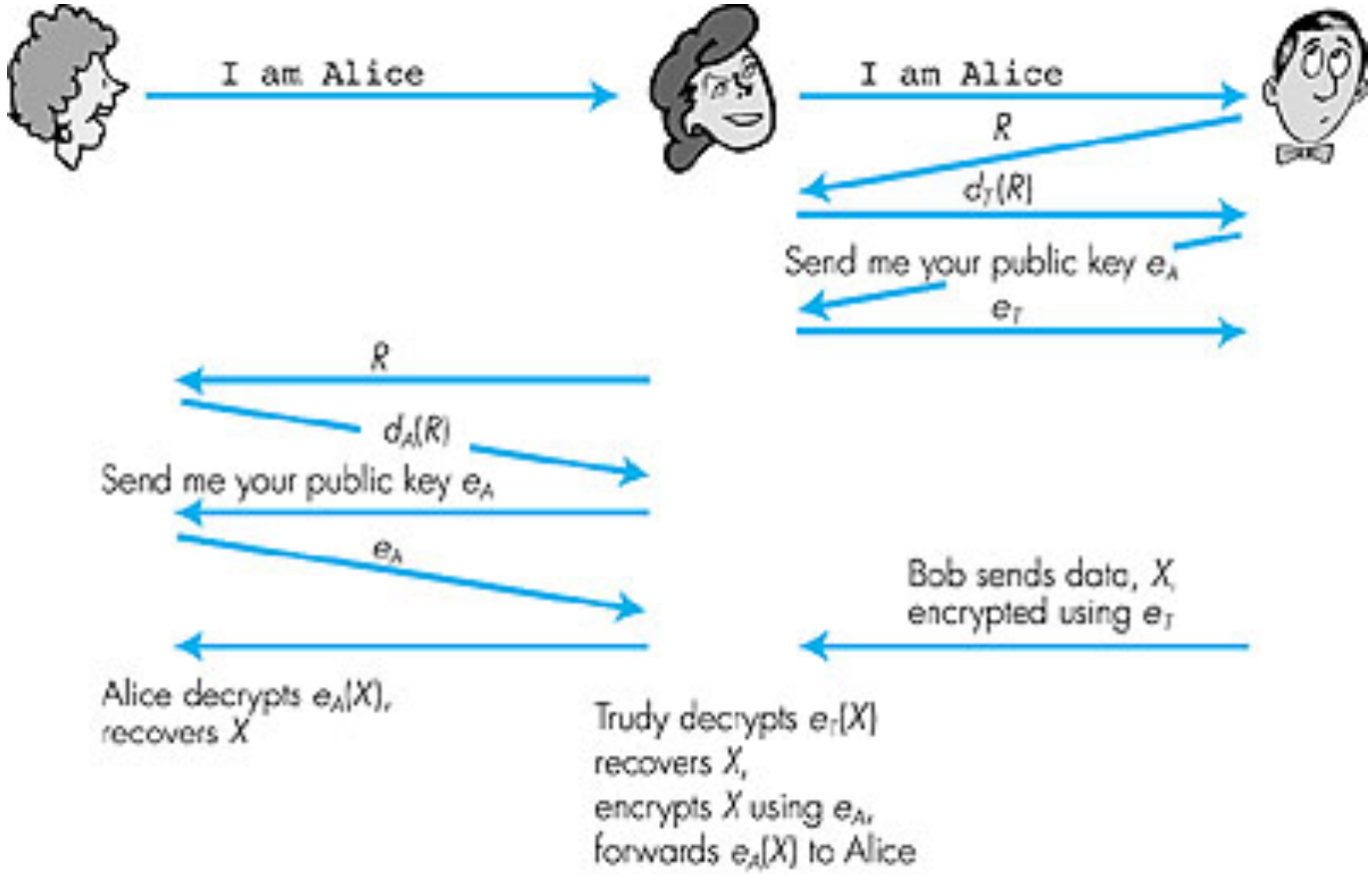
- O que garante a A estar a falar com B?
  - Receber Na em 2 – apenas B consegue obter Na, porque apenas B tem KBpriv
- O que garante a B estar a falar com A?
  - Receber Nb em 3 – apenas A consegue obter Nb, porque apenas A tem KApriv
- O que garante que Ks é seguro?
  - Ks apenas passa na rede em 2 – apenas A consegue obter Ks, porque apenas A tem KApriv (B conhece Ks porque gerou Ks)
- Criptografia assimétrica é lenta. Como soluciona o problema?
  - Negoceia chave simétrica para usar durante a comunicação

Protocolo de Needham-Schroeder com chaves públicas

- Se A apenas quiser criar um canal cifrado e ter a certeza de que está a falar com B basta:
  - **A -> B:** **{ A, Na, Ks }KBpub**
  - **B -> A:** **{ Na } Ks**

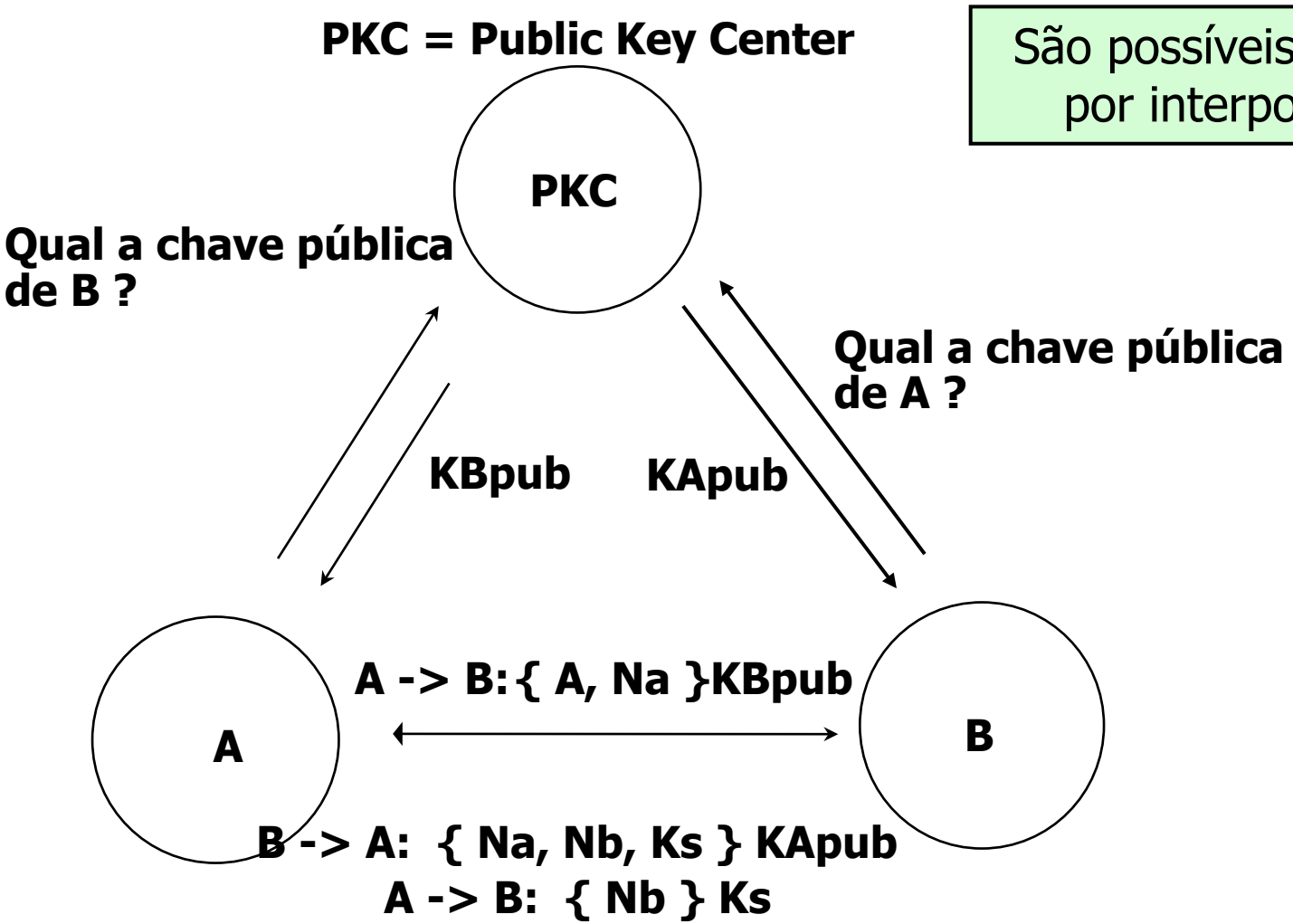
  - **O que garante a A estar a falar com B?**

Ataques por interposição



São necessárias chaves públicas *"certificadas"*

Centro de distribuição de chaves públicas



Vantagens face à criptografia simétrica

- O *Public Key Center* (PKC) apenas conhece aquilo que é público, isto é, as chaves públicas dos principais. Tal é um progresso notável pois o papel da *trusted computing base* foi drasticamente reduzido.

- Para que tudo funcione bem é apenas necessário assegurar que o PKC tem as chaves públicas verdadeiras e que os principais têm a certeza que estão a falar com o verdadeiro PKC e não com um impostor.

- Nos protocolos anteriores, um intruso que se consiga fazer passar pelo PKC, consegue levar A e B a usarem chaves conhecidas
  - São necessárias chaves públicas *"certificadas"*

Distribuição das chaves públicas

- É necessário ter absoluta segurança de que se está a dialogar com uma fonte fidedigna que nos está a entregar a verdadeira chave pública que pretendemos
  - Se se conhece a chave pública dessa fonte fidedigna, uma forma de obter esta confiança é essa fonte cifrar a sua resposta com a sua chave secreta

- Métodos de distribuição de chaves:
  - ***Certificate Granting Authority*** – entidades cujas chaves públicas são bem conhecidas e que "assinam" as chaves / certificados que entregam. Este método está hoje em dia normalizado de forma oficial.
  - ***Web of trust*** - método informal por transitividade da relação de confiança, também suportado na "assinatura" das informações trocadas entre principais. Este método tem sido vulgarizado pelo programa PGP para troca de correio electrónico.

Assinaturas digitais

Objectivo: desenvolver um mecanismo digital que substitua as assinaturas efectuadas num documento em papel

- Quais as propriedades?

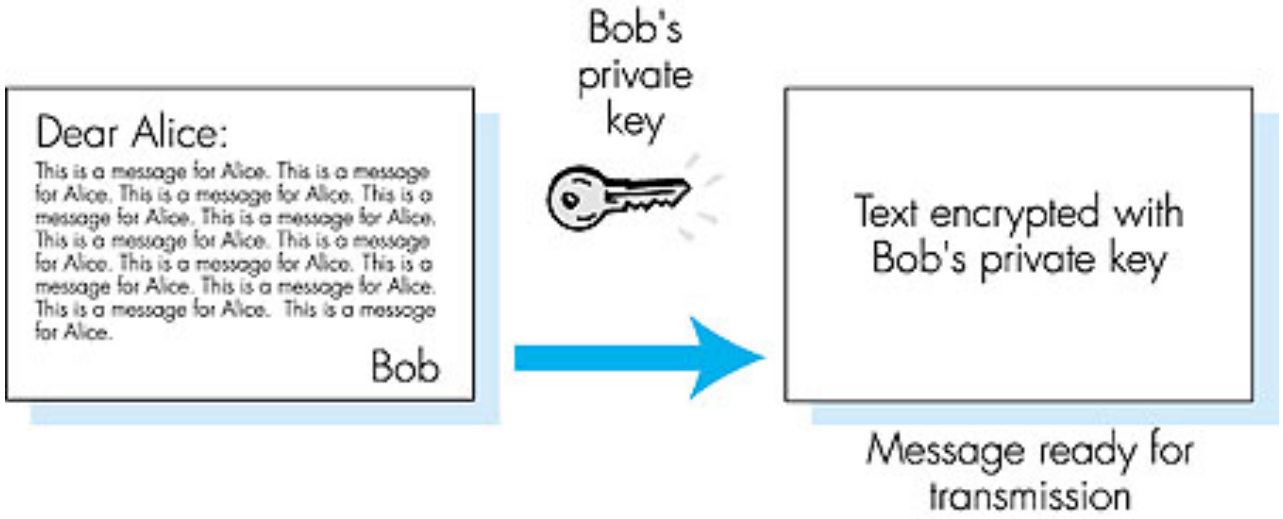
Um sistema de assinaturas digitais deve ter as seguintes propriedades:

- Autenticação: o receptor deve poder verificar que a assinatura é autentica
- Integridade: a assinatura deve garantir que a mensagem assinada não foi alterada, nem durante o trajecto, nem pelo receptor, mesmo que tenha passado em claro
- Não repudiamento: o emissor não poderá negar que de facto enviou a mensagem assinada

NOTA: O objectivo das assinaturas não é *esconder* o conteúdo

Assinatura digital: primeira tentativa

- Assinatura digital da mensagem M efectuada por Bob: {M}KBpriv



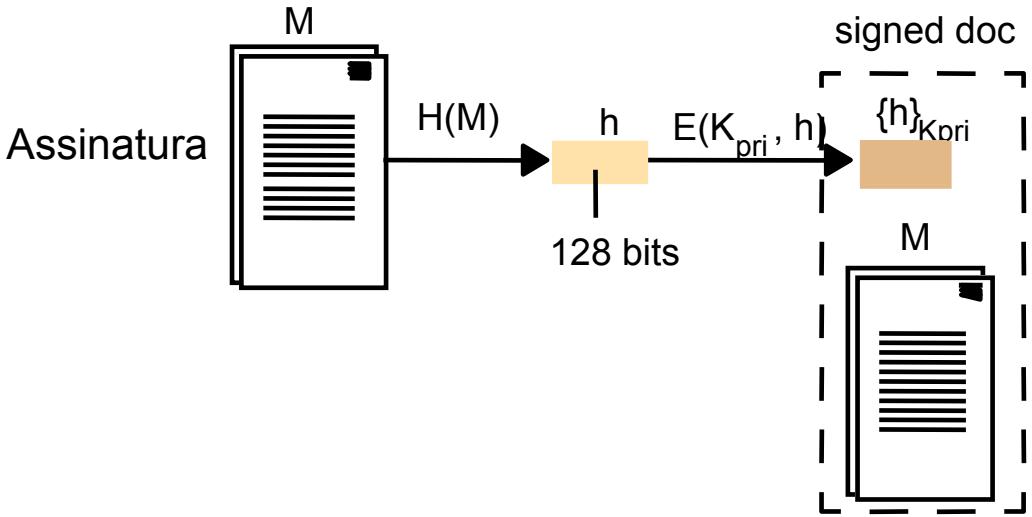
- Propriedades:
  - Autenticação verificada decifrando M = {{M}KBpriv}KBpub
  - Integridade e não repudiamento garantidos porque ninguém consegue alterar/criar {M}KBpriv sem conhecer KBpriv
  - Dimensão da assinatura idêntica à do documento
  - Necessidade de cifrar todo o documento com chaves assimétricas

Solução?

Assinatura digital: solução com chaves públicas

- Assinatura digital da mensagem M efectuada por Bob: {H(M)}KBpriv
  - Mensagem a enviar: M,{H(M)}KBpriv
  - H(M): função de síntese segura

Utilização de assinaturas digitais com chaves públicas

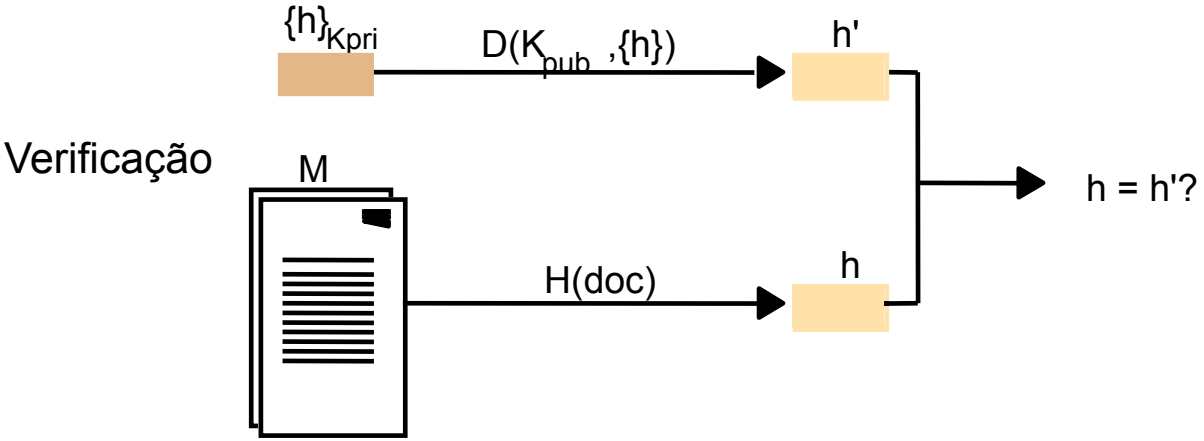


Bob envia uma mensagem com uma assinatura digital:

M,{H(M)}KBpriv

Alice verifica a assinatura e a integridade da mensagem:

H(M) = {H(M)}KBpriv}
KBpub ?



Assinatura digital: solução com chaves públicas

- Assinatura digital da mensagem M efectuada por Bob: {H(M)}KBpriv
  - Mensagem a enviar: M,{H(M)}KBpriv
  - H(M): função de síntese segura

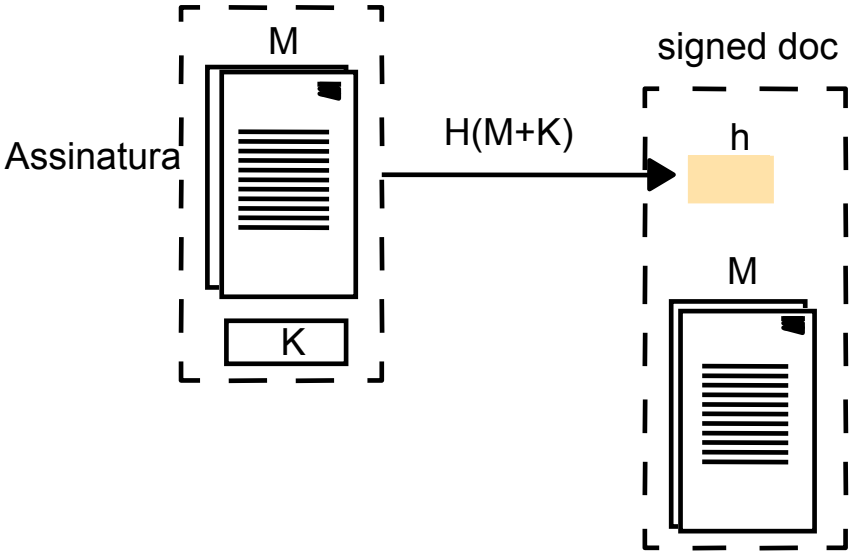
- Propriedades:
  - Verificação ao receber M',{H(M)}KBpriv : H(M') = {{H(M)}KBpriv}KBpub
  - Autenticação, integridade e não repudiamento garantidos porque apenas B consegue criar {H(M)}KBpriv e ninguém consegue alterar M para M' de forma que H(M')=H(M)

  - Dimensão da assinatura pequena e constante
  - Apenas assinatura é cifrada – mensagem pode passar em claro

Assinatura digital – solução com chaves simétricas (MACs)

- Assinatura digital da mensagem M efectuada por Bob: H(M+K)
  - Mensagem a enviar: M,H(M+K)
  - H(M): função de síntese segura

Utilização de assinaturas digitais com chaves simétricas (MACs)

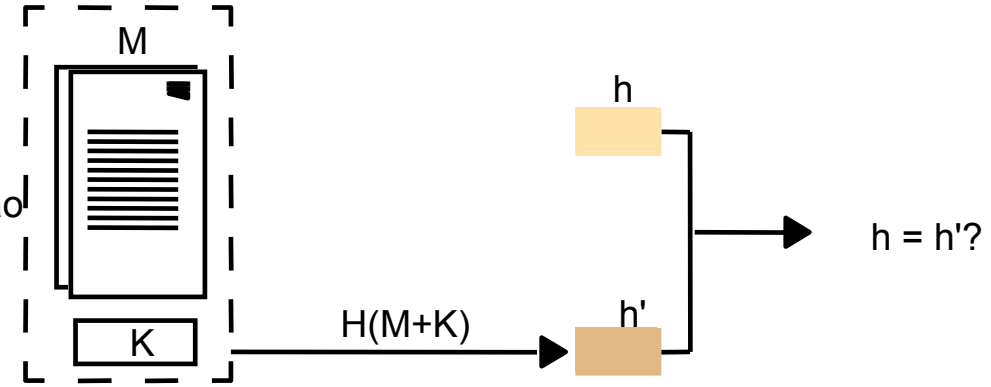


Bob envia uma mensagem com uma assinatura digital:

M, H(M+K)

Alice verifica a assinatura e a integridade da mensagem:

H(M+K) recebido = H(M+K) computado ?



Assinatura digital – solução com chaves simétricas (MACs)

- Assinatura digital da mensagem M efectuada por Bob: h=H(M+K)
  - Mensagem a enviar: M,h
  - H(M): função de síntese segura

- Propriedades:
  - Verificação ao receber M', h' : H(M'+K) = h'
  - Autenticação, integridade e não repudiamento garantidos porque apenas B consegue criar H(M+K) e ninguém consegue alterar M para M' de forma que H(M'+K)=H(M+K)

Certificados de chaves públicas

- A segurança baseada em chaves públicas depende da validade das chaves públicas.
- Objectivo: um certificado deve permitir estabelecer a autenticidade de uma chave pública (por declaração de uma entidade fidedigna)
- Certificado contém assinatura relativa, pelo menos, à informação do nome e chave pública da entidade. Exemplo:

1. *Certificate type*	**Public key**
2. *Name*	**Bob's Bank**
3. *Public key*	K_{Bpub}
4. *Certifying authority*	**Fred – The Bankers Federation**
5. *Signature*	$\{Digest(field\ 2 + field\ 3)\}_{K_{Fpriv}}$

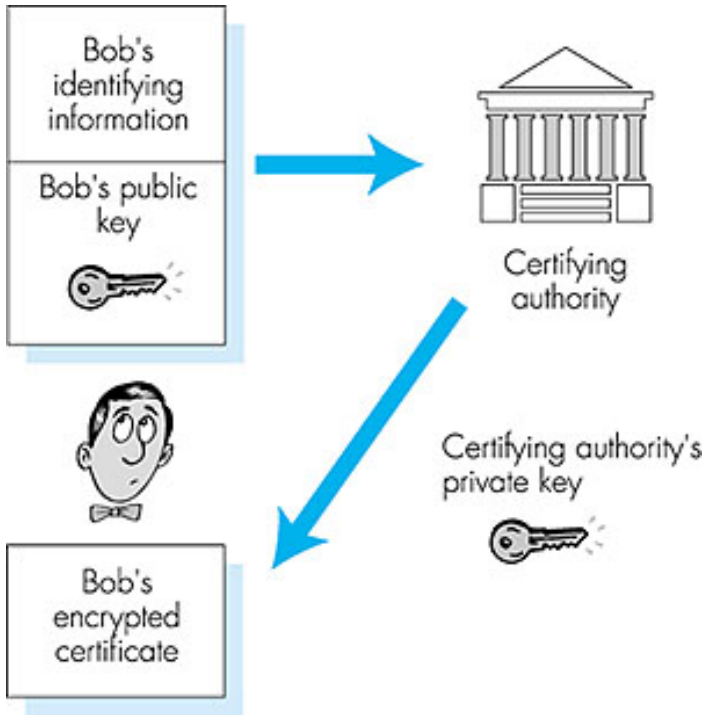
Autoridades de certificação

- Uma *certification authority* (CA) associa chaves públicas a principais
  - Em geral, os chaves públicas (certificados) das CAs são bem-conhecidos

- Quando a Alice quer verificar qual a chave do Bob:

1) obtém um certificado dessa chave (Bob pode enviá-lo !).

2) verifica autenticidade do certificado verificando a assinatura efectuada pela CA (usando chave pública no certificado da CA)



Certificados X.509

Entidade	Nome identificativo, chave pública
Emissor	Nome identificativo, assinatura
Período de validade	Data de início, data de fim
Informação administrativa	Versão, número de série
Informação adicional	

Repudiamento e validade de uma chave

- Quando uma chave privada é comprometida é necessário revogar a chave pública que lhe estava associada. Quando se utilizam certificados, é necessário que as entidades de certificação memorizem os certificados válidos e revogados.
  - Quando se utiliza um certificado devia-se verificar se o mesmo estava válido ou tinha sido revogado.
- Quando uma chave privada é comprometida é impossível garantir a autenticidade de qualquer mensagem
- Assim, a utilização de certificados por si só não é uma panaceia universal.

Algoritmo de Diffie-Hellman

- Alice e Bob pretendem trocar entre si um senha ou chave de sessão (K) sem que esta passe em claro na rede (nota: não se pretendem autenticar).

- Sejam dois números aleatórios *m* e *n* que são parâmetros públicos do algoritmo:
  1) A gera um número secreto Xa e calcula: Ya = m ^ Xa mod n
  2) B gera um número secreto Xb e calcula: Yb = m ^ Xb mod n
  3) A e B trocam entre si Ya e Yb (valores públicos passados em claro)
  4) A calcula Ka = Yb ^Xa mod n
  5) B calcula Kb = Ya ^Xb mod n

- Comprova-se que K = Ka = Kb e que é computacionalmente impossível calcular Xa ou Xb a partir de Ya ou Yb, isto é, está-se a usar uma função sem inverso (*one-way function*).
  - E.g. m = 2; n = 64;
    Xa = 2; Ya= 4; Xb =3; Yb = 8
    Ka = 8^2 = 2^6 = 64
    Kb = 4^3 = 2^6 = 64

Segurança futura perfeita

- Um protocolo de distribuição de chaves de sessão diz-se que garante **segurança futura perfeita** se não envolve segredos de longa duração que, uma vez comprometidos no futuro, permitiriam conhecer uma sessão do passado.
- Nos algoritmos NS (e similares), se uma chave secreta/privada for comprometida é possível obter as chaves secretas negociadas com base nessa chave para uso nas comunicações
  - Trudy pode ficar a conhecer o conteúdo das mensagens trocadas para todas as sessões que tenha gravado
  - Os segredos são de longa duração
- No algoritmo Diffie-Hellman, se a Alice e o Bob usarem valores aleatórios de Xa e Xb, que não voltem a ser usados, a chave é única e foi obtida sem recurso a segredos de longa duração visto que o algoritmo, m e n são públicos.
  - Xa, Xb, Ka, Kb são segredos apenas enquanto dura a sessão: depois podem (devem) ser descartados

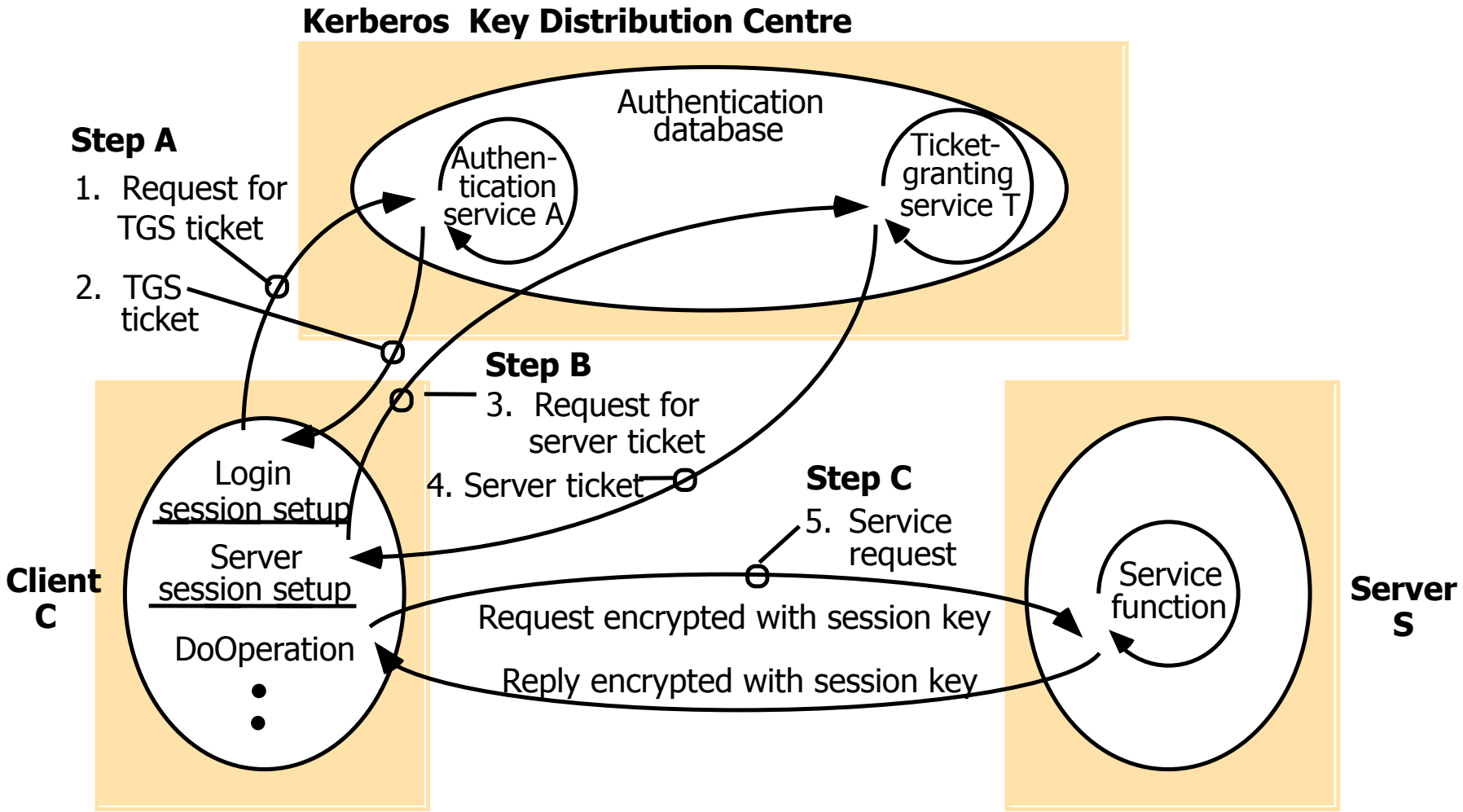
Casos de estudo

- Kerberos
- SSL
- PGP

O sistema Kerberos

- Sistema desenvolvido no MIT para fornecer serviços de autenticação e segurança numa organização
  - Usa uma variante do protocolo Needham-Schroeder com criptografia simétrica
- Servidor Kerberos
  - Serviço de autenticação
    - Permite autenticar um cliente no sistema
    - Serviço de autenticação conhece chave secreta (simétrica) de todos os utilizadores e do TGS
  - Serviço de *tickets* para servidores (TGS – *ticket granting service*)
    - Permite obter um *ticket* para aceder a um serviço
    - Serviço de *tickets* conhece chaves secretas (simétricas) de todos os servidores

O sistema Kerberos



O protocolo Kerberos versão 5.0

- Autenticação no sistema
  - **A -> AS: A, TGS, n1**
    - Cliente (A) pede para fazer *login* no sistema
  - **AS -> A: { Ks, n1}K_a, {A,TGS, t1, t2, Ks}K_{tgs}**
    - Serviço de autenticação devolve *ticket,* válido de t1 até t2, e chave que permite falar com TGS
  - **O que garante a A estar a falar com AS?**
  - **O que garante ao AS estar a falar com A?**

O protocolo Kerberos versão 5.0

- Autenticação no sistema
  - **A -> AS: A, TGS, n1**
    - Cliente (A) pede para fazer *login* no sistema
  - **AS -> A: { Ks, n1}K_a, {A,TGS, t1, t2, Ks}K_{tgs}**
    - Serviço de autenticação devolve *ticket,* válido de t1 até t2, e chave que permite falar com TGS
  - **O que garante a A estar a falar com AS?**
    - A recepção de {n1...}K_a garante a A que está a falar com AS (pois só AS conhece também Ka) e que não existe *replaying* (n1 é um *nonce*)
  - **O que garante ao AS estar a falar com A?**
    - Nada, mas apenas A pode usar informação recebida para as fases seguintes do protocolo por ser o único a conhecer Ka

O protocolo Kerberos versão 5.0 (cont.)

- Obtenção de *ticket* para aceder a serviço
  - **A -> TGS: {A, TGS, t1, t2, Ks}Ktgs, {A, t3}Ks, B, n2**
    - Cliente pede a TGS ticket para aceder ao servidor B
  - **TGS -> A: {B, n2, Kab}Ks, {A, B, t1', t2', Kab}Kb**
    - TGS devolve chave (Kab) e um *ticket* para A aceder ao serviço B

  - **O que garante a A estar a falar com TGS?**
  - **O que garante a TGS estar a falar com A?**

O protocolo Kerberos versão 5.0 (cont.)

- Obtenção de *ticket* para aceder a serviço
  - **A -> TGS: {A, TGS, t1, t2, Ks}Ktgs, {A, t3}Ks, B, n2**
    - Cliente pede a TGS ticket para aceder ao servidor B
  - **TGS -> A: {B, n2, Kab}Ks, {A, B, t1', t2', Kab}Kb**
    - TGS devolve chave (Kab) e um *ticket* para A aceder ao serviço B

  - **O que garante a A estar a falar com TGS?**
    - A recepção de {n2...}Ks garante a A que está a falar com TGS (pois só TGS consegue obter Ks) e que não existe *replaying* (n2 é um *nonce*)
  - **O que garante a TGS estar a falar com A?**
    - "Autenticador" {A, t3}Ks autentica A perante TGS por demonstrar que A conhece Ks. Time-stamp (t3) impede *replaying* desta mensagem.
    - NOTA: Ka apenas é usada na autenticação. A partir desse momento, o ticket enviado pelo AS permite ao cliente comunicar de forma segura e autenticar-se no TGS

O protocolo Kerberos versão 5.0 (cont.)

- Acesso ao serviço
  - **A -> B:** **{A, t3'}Kab, {A, B, t1', t2', Kab}K_b, n3**
    - Cliente pede para se autenticar com o servidor
  - **B -> A:** **{n3}Kab**
    - Cliente verifica autenticidade do servidor. A partir deste momento, cliente e servidor podem usar Kab para comunicarem de forma segura.

- **O que garante a A estar a falar com B? Como se evita o replaying?**
- **O que garante a B estar a falar com A? Como se evita o replaying?**

O protocolo Kerberos versão 5.0 (cont.)

- Acesso ao serviço
  - **A -> B: {A, t3'}Kab, {A, B, t1', t2', Kab}K_b, n3**
    - Cliente pede para se autenticar com o servidor
  - **B -> A: {n3}Kab**
    - Cliente verifica autenticidade do servidor. A partir deste momento, cliente e servidor podem usar Kab para comunicarem de forma segura.

  - **O que garante a A estar a falar com B? Como se evita o replaying?**
    - Apenas B pode produzir {n3}Kab, porque apenas B consegue obter Kab a partir do ticket. Replaying é evitado por n3 ser um nonce.
  - **O que garante a B estar a falar com A? Como se evita o replaying?**
    - Apenas A conhece Kab, logo só A pode produzir {A,t3'}Kab. t3' é um timestamp usado para evitar replaying.

Notas sobre o protocolo Kerberos v. 5.0

- Os valores de t1, t1', t2, t2', t3, t3' são estampilhas temporais geradas a partir do relógio (que se pressupõem estar fracamente sincronizados)
  - Permite aplicar tempo limite a *tickets*, revogando autorizações dadas
  - Permite protecção contra *replaying* de mensagens antigas (ou utilização indevida de *tickets* encontrados em memória)

- Clientes devem obter novos tickets quando termina a validade dos tickets obtidos.
- Clientes devem obter um ticket para cada servidor que queiram contactar.

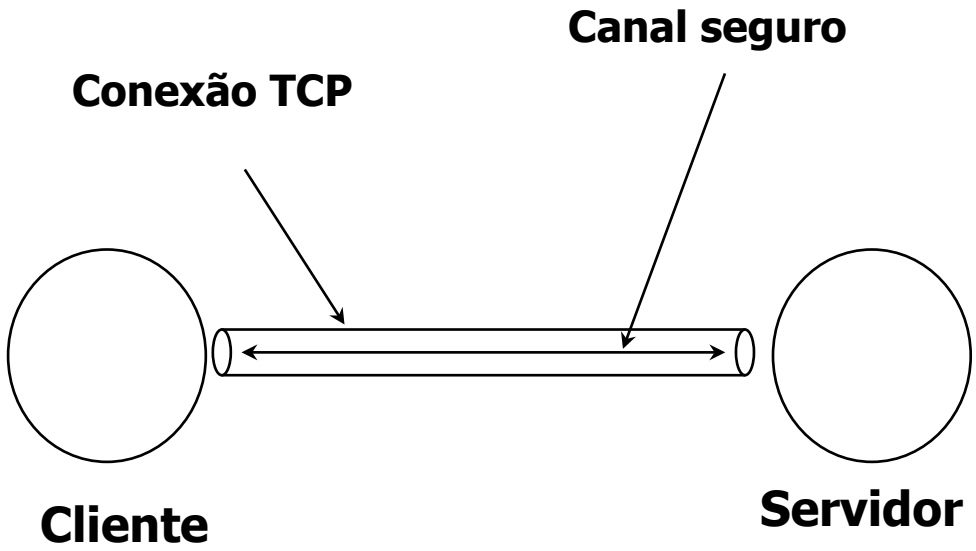
Vantagens da variante Kerberos sobre o protocolo Needham-Schroeder simples

- O AS conhece as senhas / chaves / passwords de todos os clientes mas estas só são usadas para autenticar inicialmente os clientes
- A senha / chave / password do cliente (que usa workstations avulso) só está na memória das mesmas durante alguns milisegundos
- Os tickets têm uma duração limitada (tanto com o TGS como com os servidores)
  - Validade dos tickets é geralmente de algumas horas
  - Permite interromper o serviço a clientes a quem tenha sido cancelado o registo
- O TGS apenas tem de conhecer os servidores e emitir chaves de sessão (repare-se que os clientes são muito mais numerosos que os servidores)

- Quando se usa um sistema tipo Kerberos, quais as diferentes formas de controlo de acessos executadas?

SSL - Secure Socket Layer

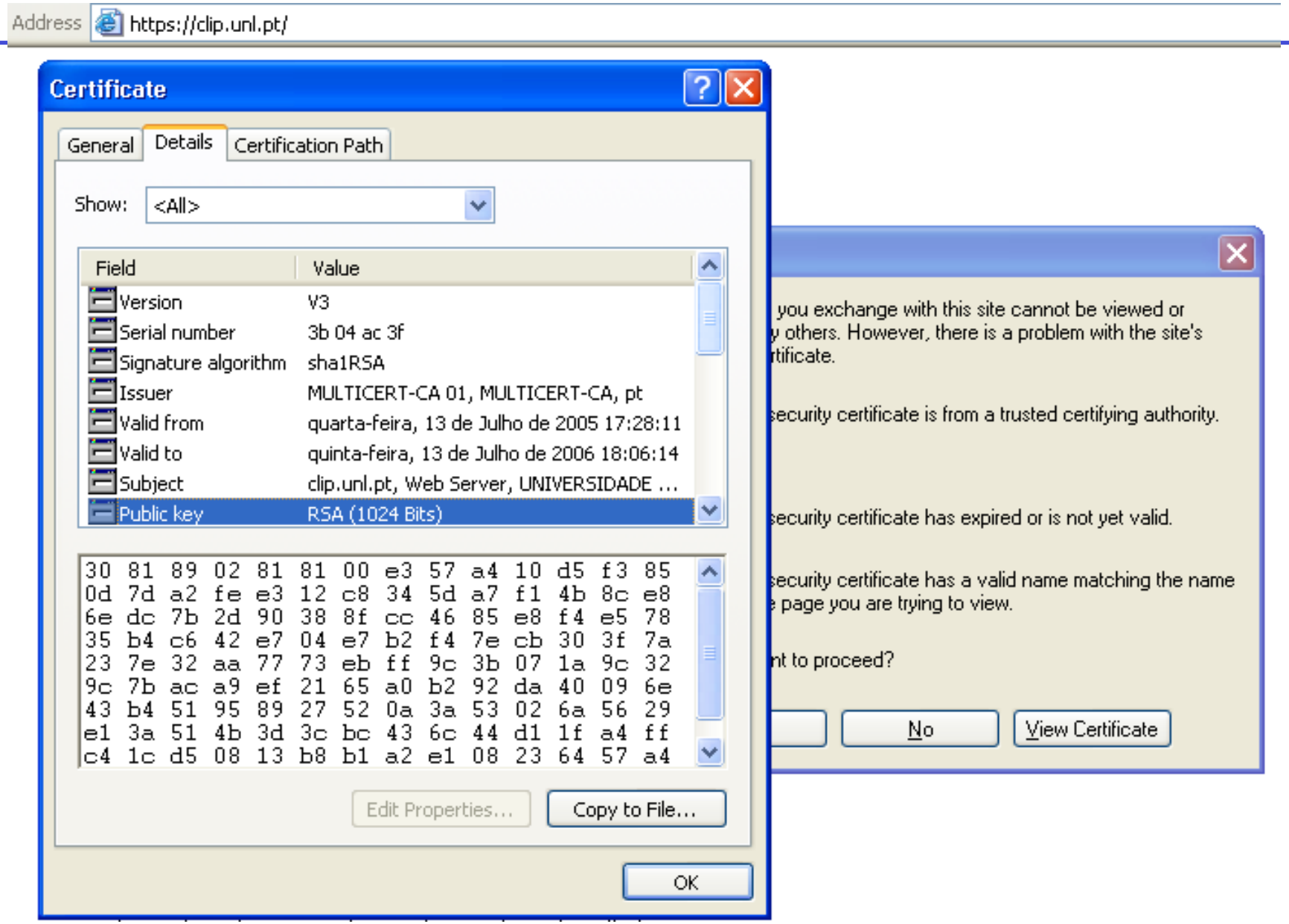
- Trata-se de um protocolo proposto e desenvolvido pela Netscape, do tipo sessão, isto é sobre o nível transporte, que permite estabelecer canais seguros e autenticar clientes e servidores. Hoje em dia trata-se de uma norma IETF.



O protocolo SSL

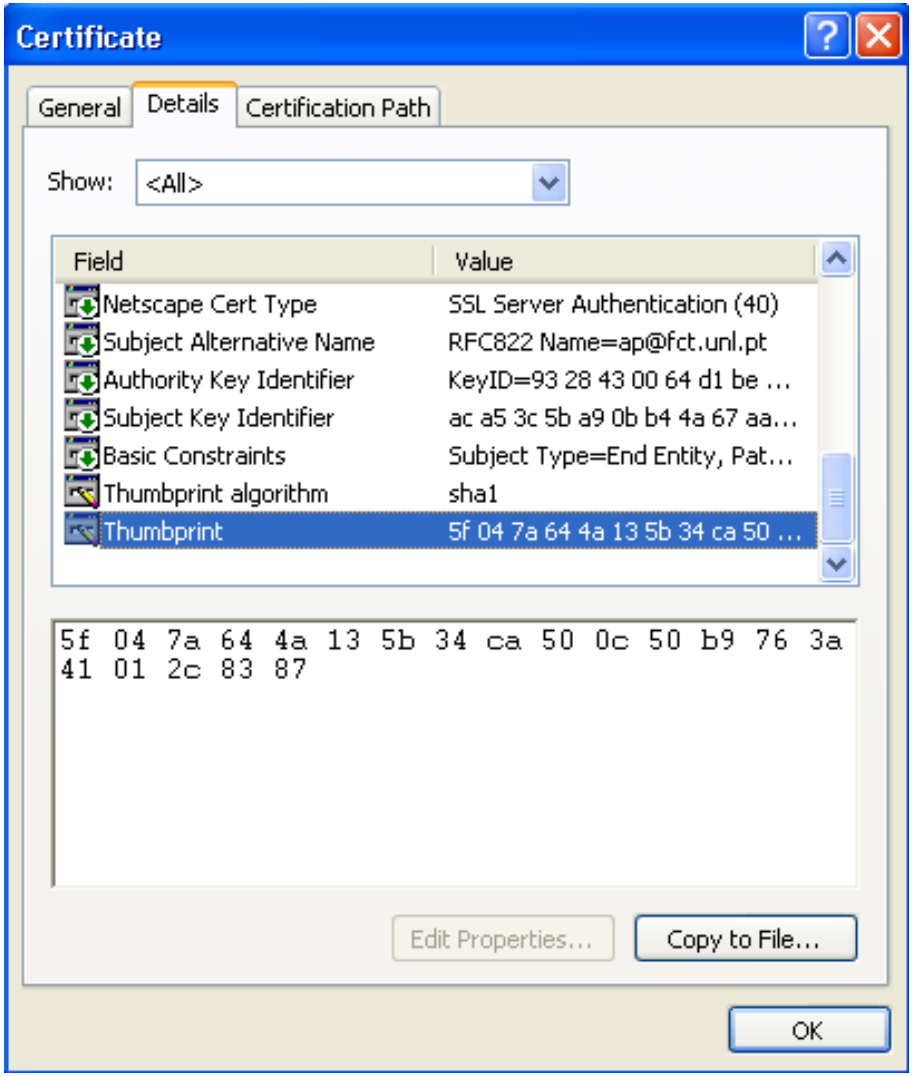
- O protocolo SSL funciona por cima do nível de transporte e permite fornecer segurança a qualquer aplicação baseada em TCP
- É usado entre browsers e servidores WWW (https).
- Funcionalidades de segurança:
  - autenticação do servidor
  - cifra dos dados
  - autenticação do cliente (opcional)
- Autenticação do servidor:
  - O cliente (browser) inclui as chaves públicas de várias CAs
  - O cliente solicita ao servidor um certificado do servidor emitido por uma CA em que ele confie.
  - O cliente verifica o certificado do servidor com a chave pública da CA
- Autenticação do cliente:
  - Processa-se de forma semelhante
- Veja no seu browser na secção de segurança.

Certificado CLIP



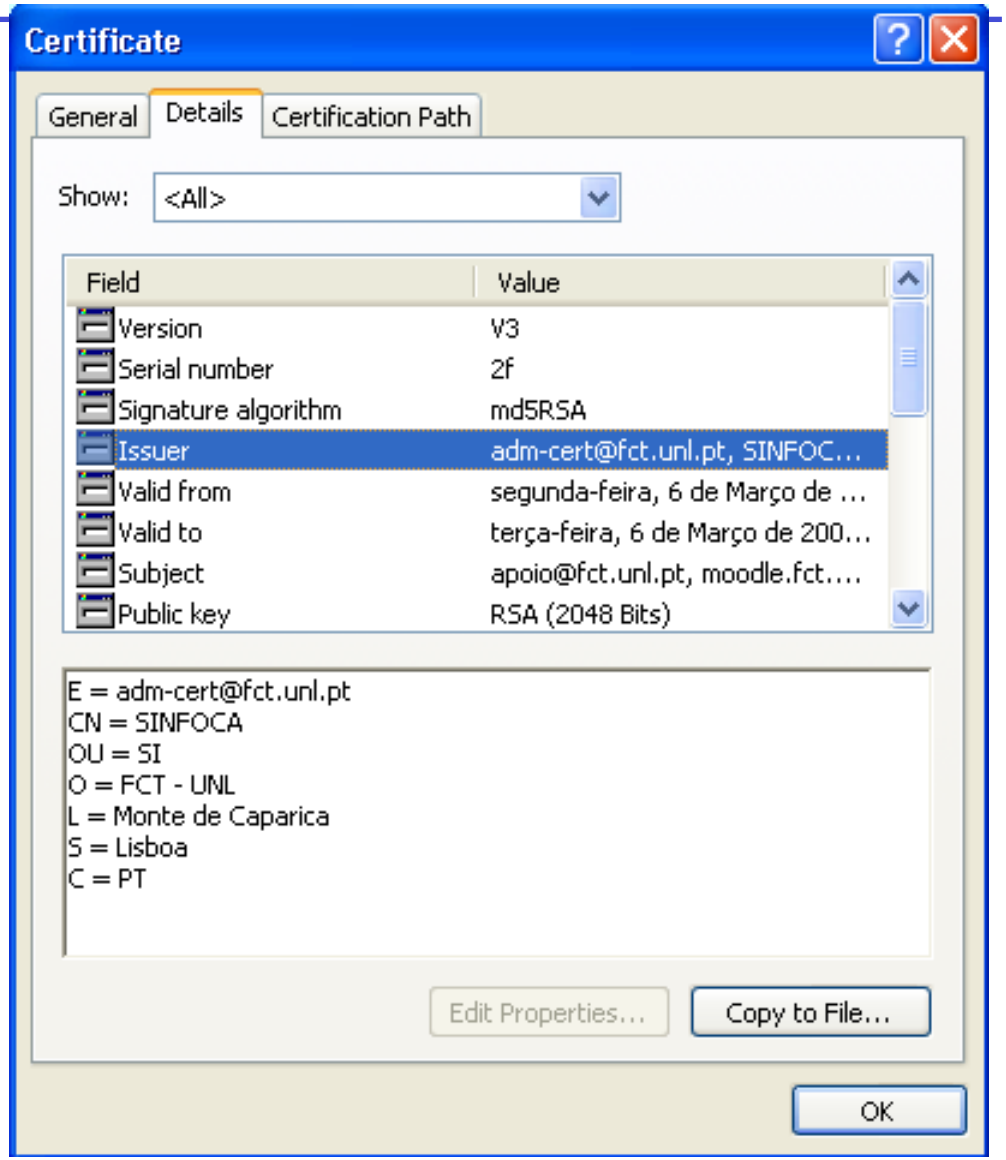
Certificado CLIP

- Assinado pela Multicert
  - Empresa de certificação nacional: SIBS,CTT,INCM,PTPrime

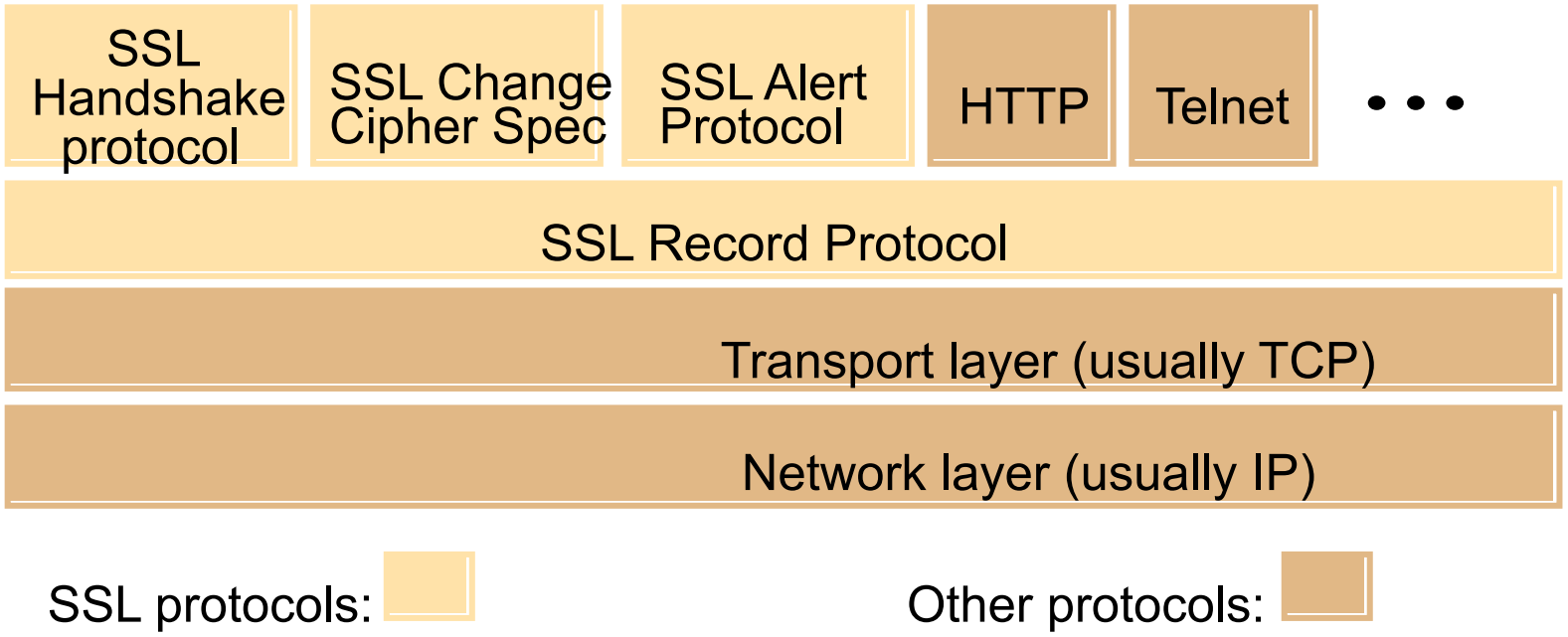


Certificado Moodle@FCT

- Certificado emitido localmente
- Que garantias fornece?

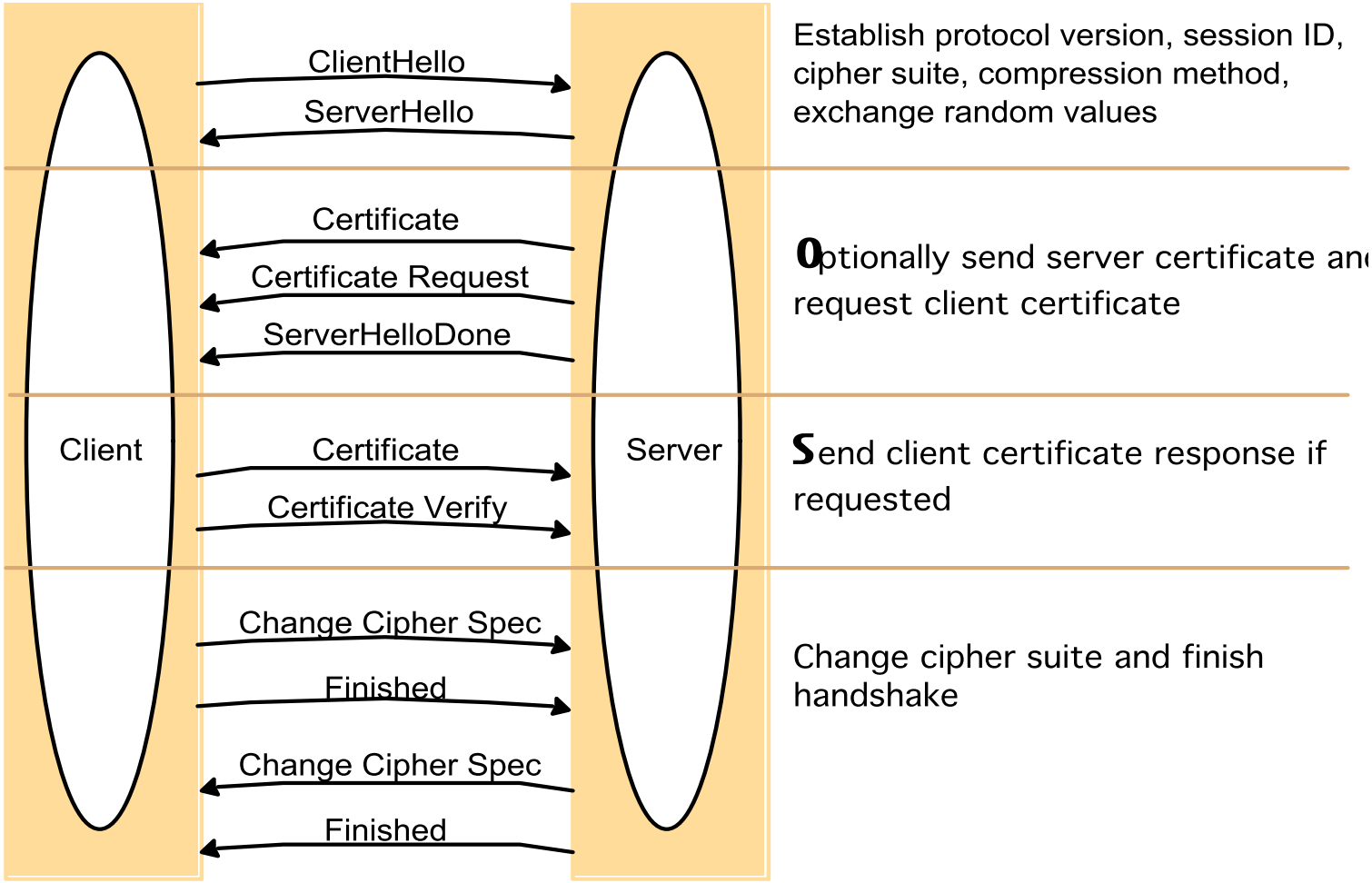


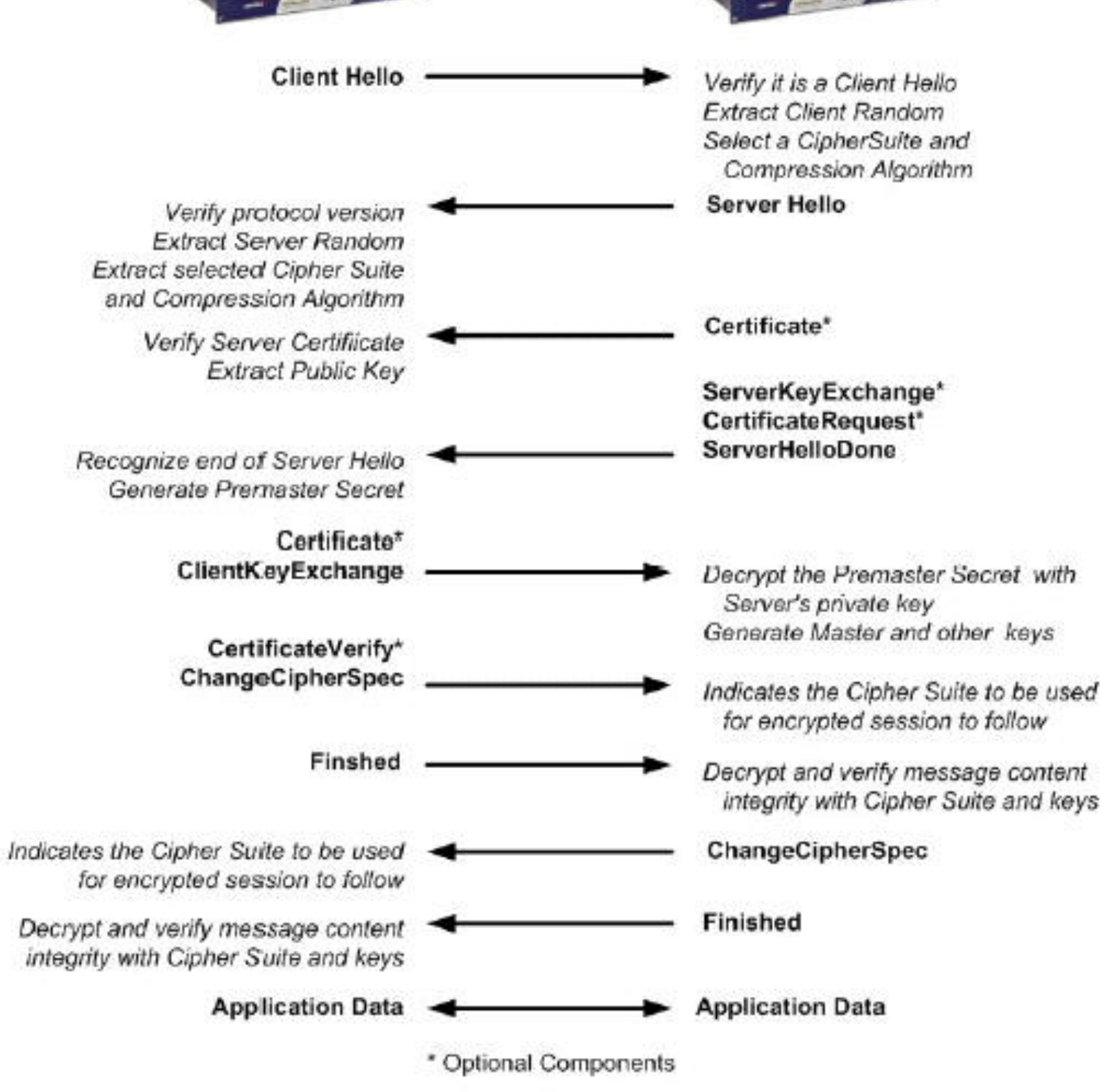
Protocolos SSL



- *SSL Record Protocol*: implementa canal seguro, cifrando e autenticando as mensagens
- *SSL Handshake protocol + SSL CCS + SSL AP*: estabelecem e mantém um canal seguro entre um cliente e um servidor

SSL handshake protocol



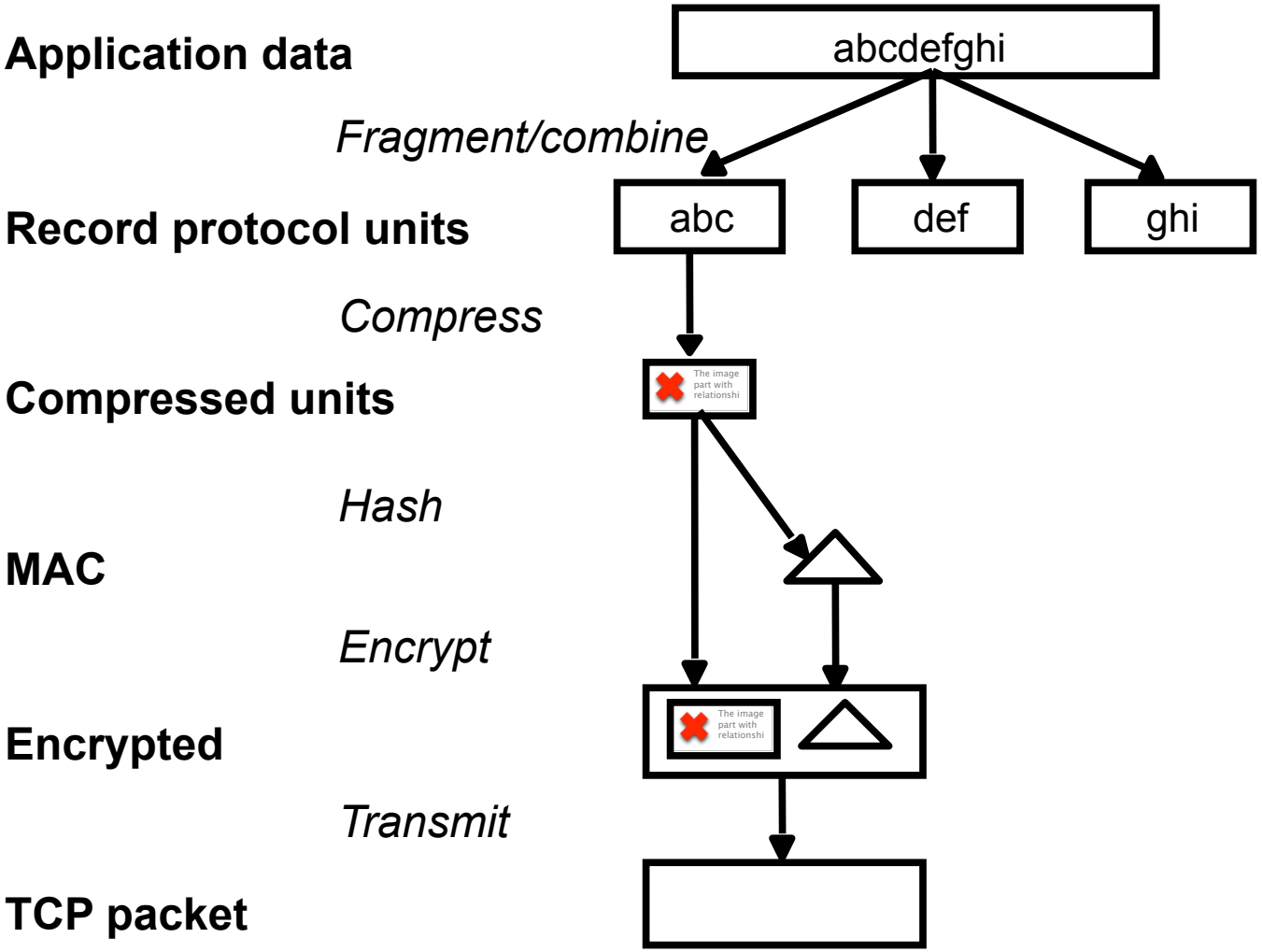

Client Hello
Verify it is a Client Hello
Extract Client Random
Select a CipherSuite and
Compression Algorithm
Verify protocol version
Extract Server Random
Extract selected Cipher Suite
and Compression Algorithm
Server Hello
Verify Server Certifiicate
Extract Public Key
Certificate*
ServerKeyExchange*
CertificateRequest*
ServerHelloDone
Recognize end of Server Hello
Generate Premaster Secret
Certificate*
ClientKeyExchange
Decrypt the Premaster Secret with
Server's private key
Generate Master and other keys
CertificateVerify*
ChangeCipherSpec
Indicates the Cipher Suite to be used
for encrypted session to follow
Finshed
Decrypt and verify message content
integrity with Cipher Suite and keys
Indicates the Cipher Suite to be used
for encrypted session to follow
ChangeCipherSpec
Decrypt and verify message content
integrity with Cipher Suite and keys
Finished
Application Data
Application Data
* Optional Components

SSL handshake configuration options

Component	*Description*	*Example*
Key exchange method	the method to be used for exchange of a session key	RSA with public-key certificates
Cipher for data transfer	the block or stream cipher to be used for data	IDEA
Message digest function	for creating message authentication codes (MACs)	SHA

- SSL permite usar diferentes *Cipher Suites*. Durante o *handshake* o servidor indica quais estão disponíveis e o cliente selecciona um.
- Autenticação do servidor: servidor envia certificado e cliente envia *premaster secret* cifrado com a chave pública do servidor
  - *Premaster secret* é usado para gerar as chaves de cifra (uma para cada sentido) e a chave a usar no MAC
- Autenticação do cliente: cliente envia certificado e cliente envia assinatura de parte das mensagens trocadas

SSL record protocol



Pretty good privacy (PGP)

- Esquema de cifra e assinatura de e-mail.
  - Trata-se de uma norma *de facto*.
- Utiliza criptografia simétrica, criptografia assimétrica ou de chave pública, funções de hash seguras e assinaturas digitais.
- Fornece autenticação, secretismo, integridade e não repudiamento.
- O inventor, Phil Zimmerman, foi alvo de um processo legal nos EUA durante 3 anos.

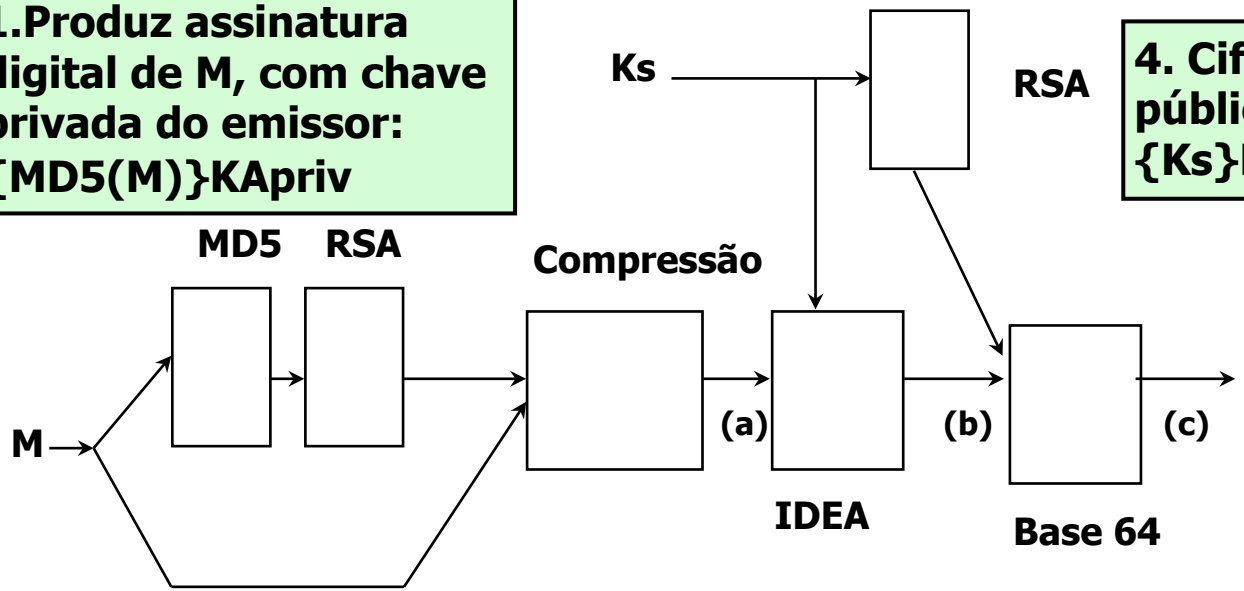
Exemplo de uma mensagem PGP:

```
---BEGIN PGP SIGNED MESSAGE--

Hash: SHA1

Bob:My husband is out of town
   tonight.Passionately yours, Alice

---BEGIN PGP SIGNATURE---
Version: PGP 5.0
Charset: noconv
yhHJRHhGJGhgg/12EpJ
   +lo8gE4vB3mqJhFEvZP9t6n7G6m5Gw2
---END PGP SIGNATURE---
```

Funcionamento do PGP

Envio da mensagem M de A para B

1.Produz assinatura digital de M, com chave privada do emissor: {MD5(M)}KApriv



4. Cifra Ks com chave pública do receptor: {Ks}KBpub

5. Resultado de 3+4 são concatenados e codificados em ASCII 7 bits usando o método base 64. O resultado é enviado através de correio electrónico.

2. Comprime M+{MD5(M)}KApriv

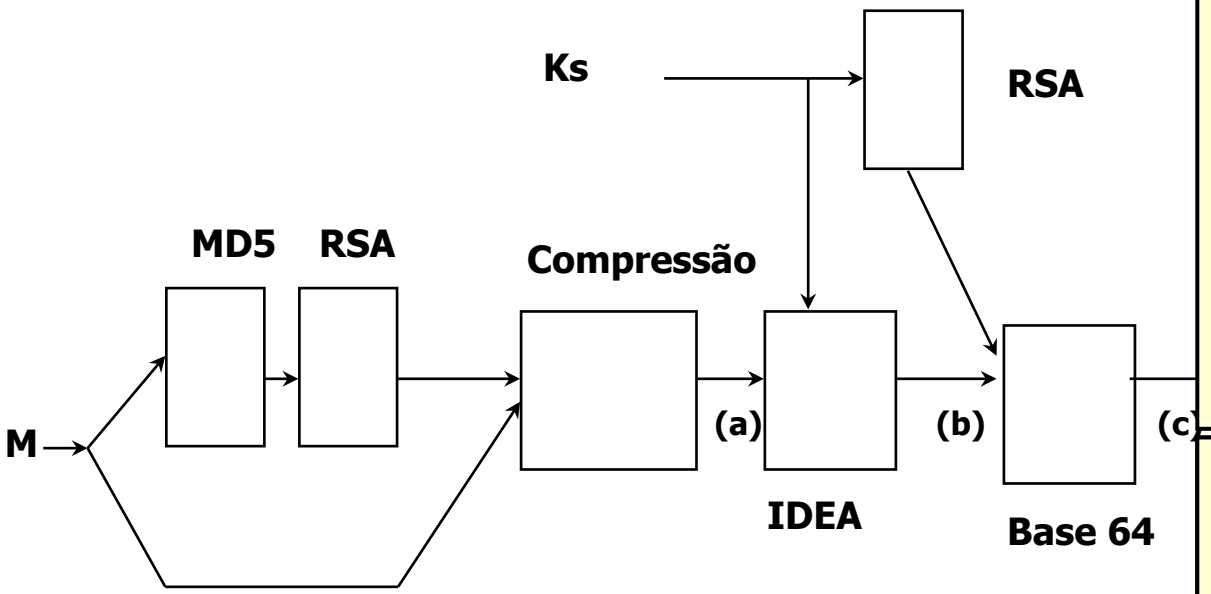
3. Gera Ks e cifra resultado de 2 com Ks (algoritmo IDEA)

(a) M, { MD5(M) } KApriv

(b) { M, { MD5(M) } KApriv } Ks

(c) { M, { MD5(M) } KApriv } Ks , { Ks } KBpub

Funcionamento do PGP



(a) M, { MD5(M) } KApriv

(b) { M, { MD5(M) } KApriv } Ks

(c) { M, { MD5(M) } KApriv } Ks , { Ks } KBpub

O que garante que só o receptor (B) pode ler a mensagem?

A mensagem é cifrada com Ks, logo só quem conheça Ks pode decifrar a mensagem.

Ks é cifrada com KBpub, logo só B pode obter Ks, porque só B conhece KBpriv

O que garante ao receptor (B) que foi o emissor (A) que enviou a mensagem?

Apenas o emissor pode produzir a assinatura digital da mensagem, porque apenas A conhece KApriv

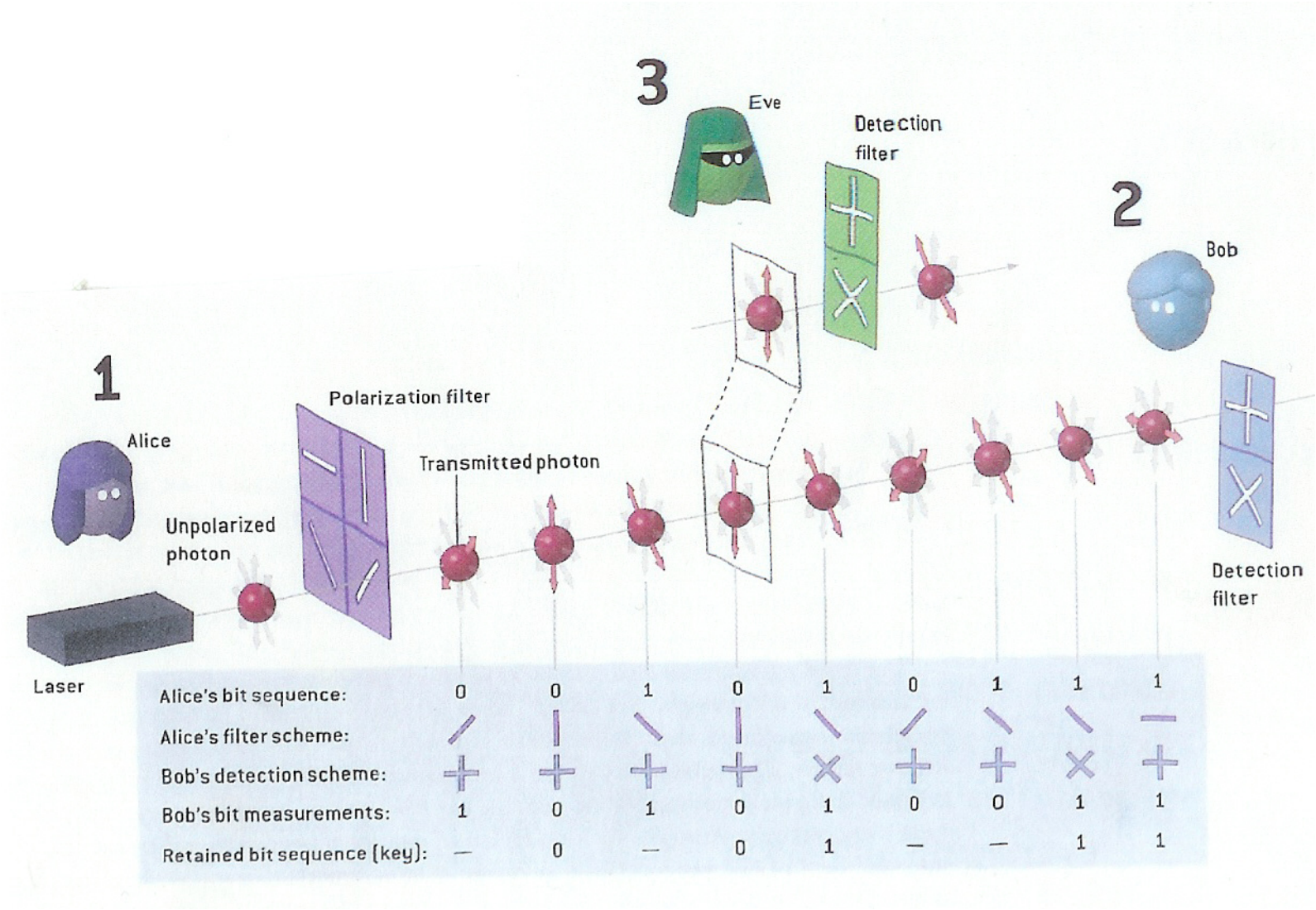
Funcionamento do PGP

1. Produz assinatura da mensagem M. Seja M a mensagem a enviar, H a função de *hash* seguro, e KApriv a chave privada RSA de A, a assinatura gerada é {H (M)} KApriv.
2. A concatenação da mensagem M com a assinatura é comprimida
3. Uma chave secreta Ks é então gerada e o resultado de 2) é cifrado através do algoritmo IDEA com a chave Ks.
4. Ks é cifrada com a chave pública do receptor, KBpub.
5. O resultado de 4) e 5) são concatenados e codificados em ASCII 7 bits pelo método base 64.
6. O resultado de 5) é enviado através de correio electrónico.

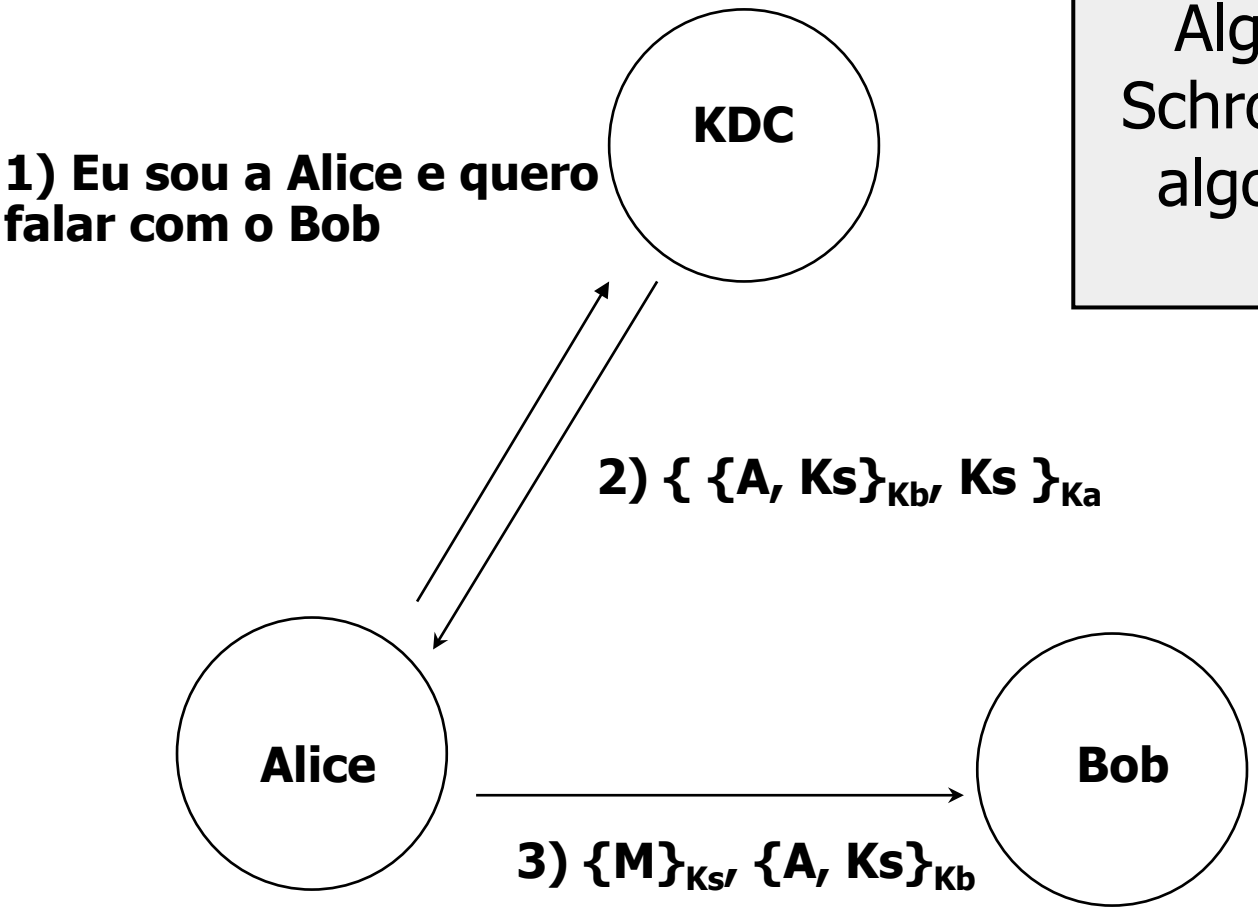
Distribuição de chaves em PGP

- O programa PGP vulgarizou um método de distribuição de chaves dito *Web of Trust* que consiste em cada pessoa adquirir as chaves públicos dos interlocutores através de outras pessoas em quem tem confiança, ou através de servidores públicos.
  - PGP fornece um método de assinatura de chaves para que cada principal possa assinar as chaves que envia a quem confia nele (que já conhece a sua chave ou chaves públicas).
  - Para fazer *bootstrap* da *Web of trust* utilizam-se métodos administrativos que ultrapassam o âmbito da problemática criptográfica, nomeadamente relacionados com a forma como as chaves são depositadas no servidor de chaves públicas.
- Cada utilizador pode ter várias chaves privadas (private key ring) e dispõe de um directório de chaves públicas de terceiros (public key ring) classificadas por grau de confiança da fonte que forneceu a chave.
  - As chaves podem ter 512 (utilização casual), 1024 (utilização comercial) ou 2048 bits (utilização segura).

Criptografia e computação quântica: quantum key distribution



Ideia base

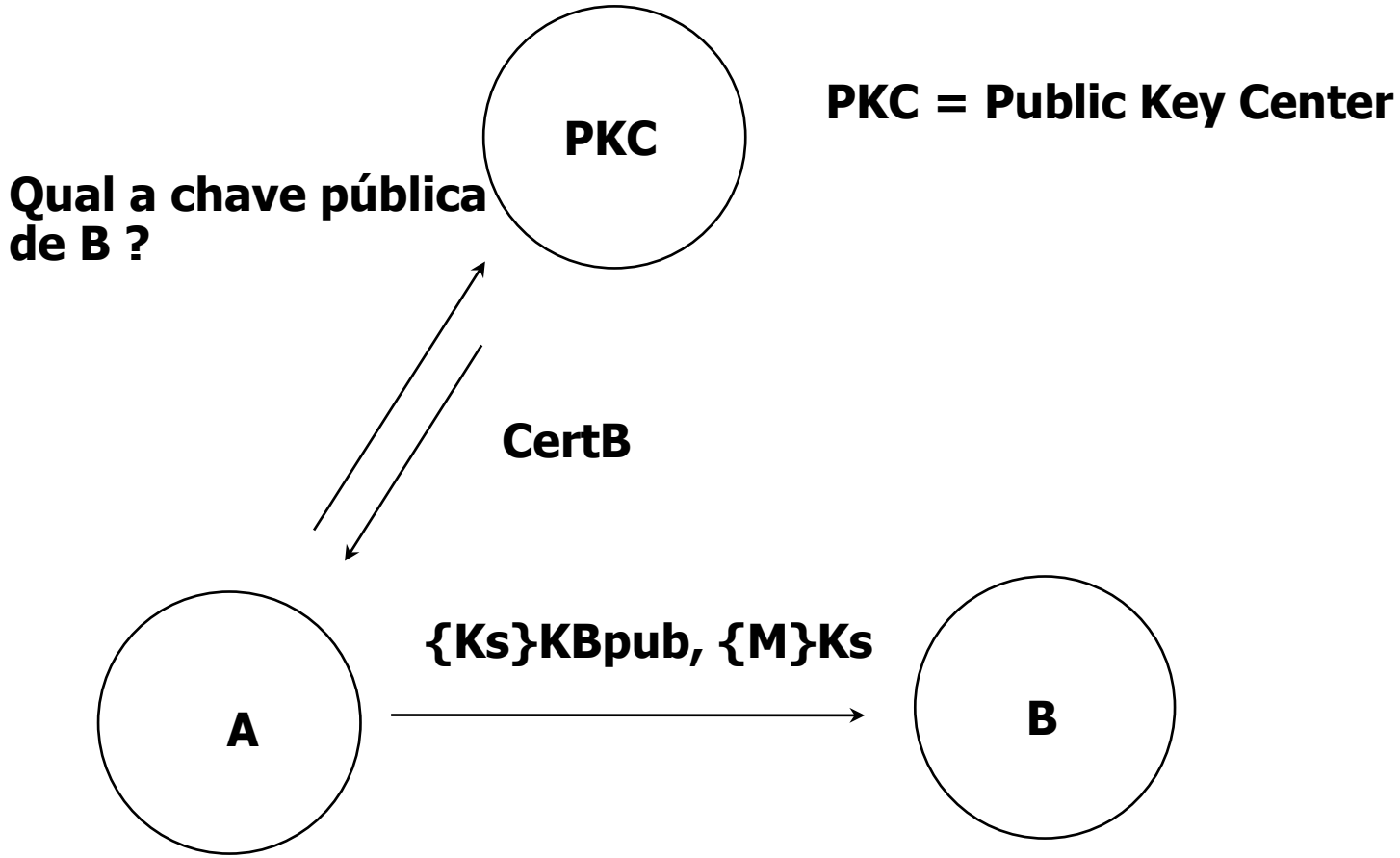


Algoritmo Needham-Schroeder melhora este algoritmo, impedindo *replaying*

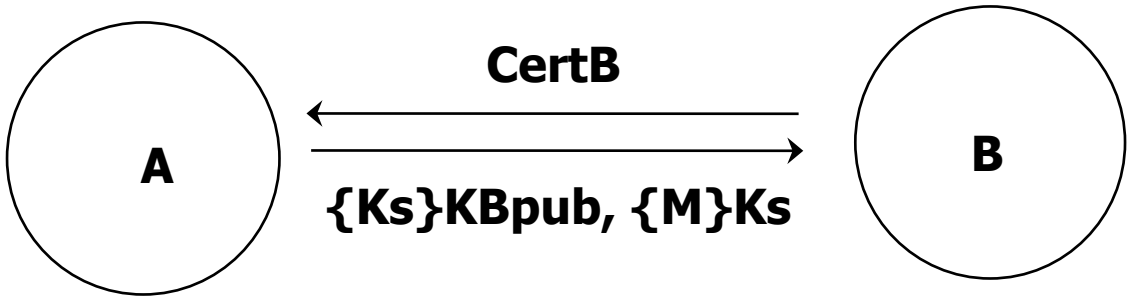
Resumo

- Estabelecimento de canal cifrado
  - Criptografia simétrica
    - Parceiros devem partilhar segredo
    - Parceiros devem partilhar segredo com terceira entidade (KDC) (mas não necessariamente entre si)
  - Criptografia assimétrica
    - Parceiros necessitam de conhecer chave pública do parceiro
    - Parceiros podem conhecer chave pública de terceira entidade (PKC) que assina chave pública das entidades
      - Usada geralmente para negociar chave de sessão simétrica, com a qual os dados são transmitidos

Ideia base


PKC
PKC = Public Key Center
Qual a chave pública de B ?
CertB
{Ks}KBpub, {M}Ks
A
B
Verifica validade de CertB, com base nos certificados das CA que conhece

Ideia base



Verifica validade de CertB, com base nos certificados das CA que conhece

Resumo

- Estabelecimento de canal cifrado
  - Criptografia simétrica
    - Parceiros devem partilhar segredo
    - Parceiros devem partilhar segredo com terceira entidade (KDC) (mas não necessariamente entre si)
  - Criptografia assimétrica
    - Parceiros necessitam de conhecer chave pública do parceiro
    - Parceiros podem conhecer chave pública de terceira entidade (PKC) que assina chave pública das entidades
      - Usada geralmente para negociar chave de sessão simétrica, com a qual os dados são transmitidos
  - Diffie-Hellman
    - Permite estabelecer canal cifrado entre dois parceiros, mas não os autentica

Resumo

- Autenticação dos parceiros (ideia base)
  - Criptografia simétrica
    - Capacidade de decifrar mensagem cifrada com chave secreta
  - Criptografia assimétrica
    - Capacidade de cifrar mensagem com chave privada
    - Capacidade de decifrar mensagem cifrada com chave púbica
  - Diffie-Hellman
    - Autenticação deve ser feita de outra forma

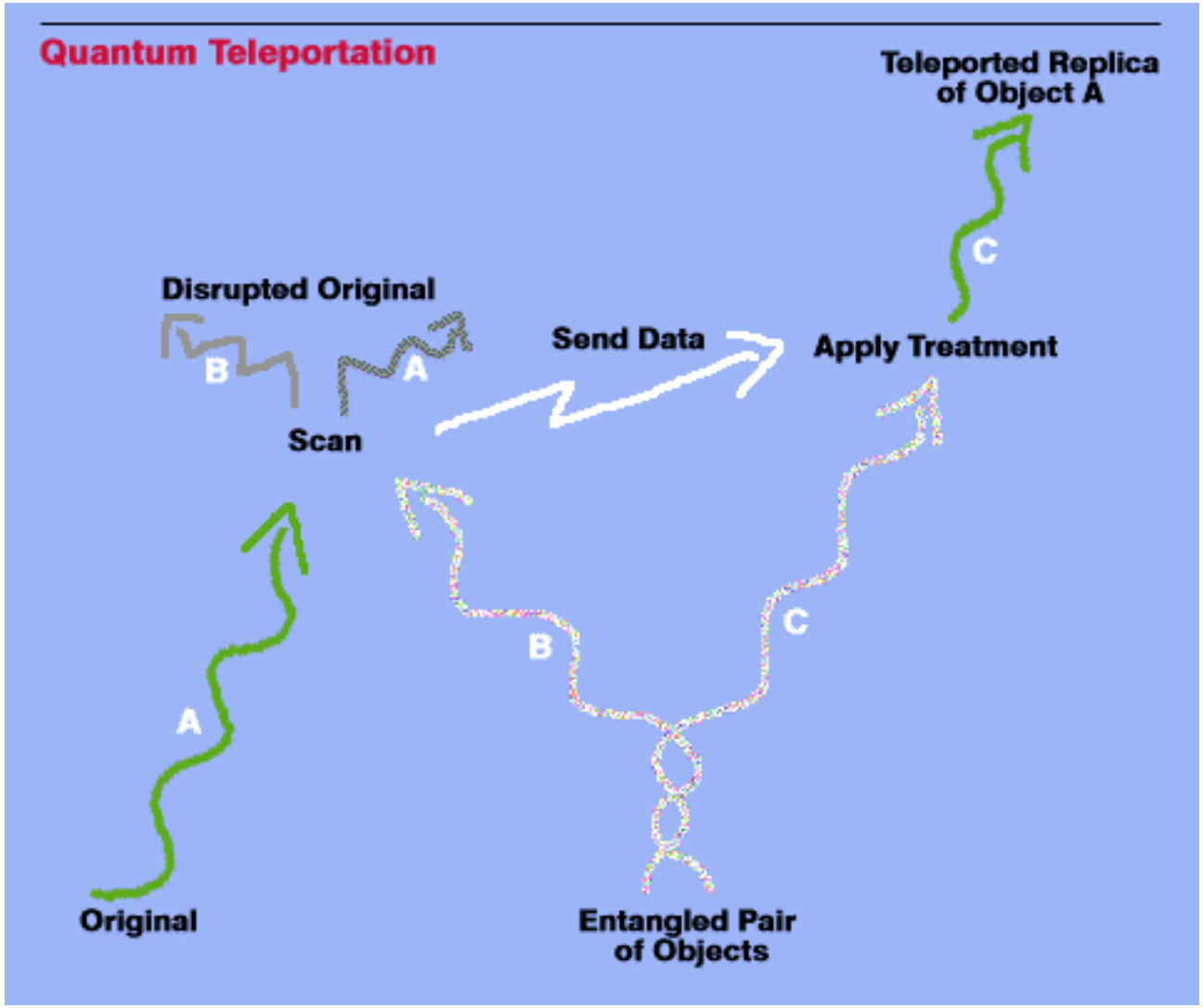
Resumo

- Garantia de integridade e autenticidade das mensagens
  - Assinaturas digitais baseadas em chaves assimétricas
  - MACs

Resumo

- Estabelecimento de canal cifrado
  - Criptografia simétrica
    - Parceiros devem partilhar segredo
    - Parceiros devem partilhar segredo com terceira entidade (mas não necessariamente entre si)

Criptografia e computação quântica: quantum teleportation



Para saber mais

- Bibliografia base
  - G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems - Concepts and Design, Addison-Wesley, 4th Edition, 2005
    - Capítulo 7

- Bibliografia adicional:
  - James F. Kurose and Keith W. Ross, "Computer Networking - A Top-Down Approach Featuring the Internet," Addison Wesley Longman, Inc., Second Edition, 2003 – capítulo 7.
  - Tanenbaum and Maarten van Steen "Distributed Systems – Principles and Paradigms," Prentice-Hall, 2002 – capítulo 8.
  - W. Stallings, "Cryptography and Network Security – Principles and Practice," Second Edition 1999 – Este livro é integralmente dedicado a este tópico.

Problema 1:

- Três entidades: Clientes (C), S1, S2
  - S1 tem KpubS1/KprivS1. Clientes conhecem KpubS1
  - S1 conhece nome,pwd de cada cliente
  - S1 e S2 partilham Ks
- C quer executar operação em S2
  - C envia op para S2
  - S2 envia res para C
- Propriedades: secretismo, integridade, autenticação

Problema 2:

- Três entidades: Clientes (C), S1, S2
  - C partilha Ks com S1
  - C partilha Ks2 com S2
- C quer executar operação com duas partes, uma em S1 e outra em S2
- Protocolo com 3 mensagens C->S1; S1->S2 ; S2->C
  - C envia M1 para S1 e M2 para S2;
  - S1 envia R1 para S2
  - S2 envia R2 para C
- Propriedades: secretismo, integridade, autenticação

Problema 3

- Três entidades: Clientes (C), AP, S
  - S conhece user/pwd de C
  - S tem KpubS/KprivS. C conhece KpubS
  - AP e S partilham Ks
- C e AP querem negociar chave de sessão
- Protocolo com 4 mensagens C->AP; AP->S ; S->AP; AP->C
- Propriedades: secretismo, integridade, autenticação

Sistemas Distribuídos

Capítulo 7

Introdução aos sistemas de designação, de descoberta e de localização de serviços

Nota prévia

- A apresentação utiliza algumas das figuras livro de base do curso

  G. Coulouris, J. Dollimore and T. Kindberg,
  Distributed Systems - Concepts and Design,
  Addison-Wesley, 4th Edition, 2005

Organização do capítulo

- Introdução à problemática da designação de objectos distribuídos
- Nomes e estruturação do espaço de nomes
- Servidores de designação
- Serviços de descoberta

Da necessidade dos nomes

- Num sistema distribuído os nomes são imprescindíveis para designar computadores, serviços, utilizadores, objectos remotos, ficheiros e recursos em geral, … Os diferentes componentes do sistema, assim como os utilizadores, só podem partilhar recursos se os poderem designar.
- Os nomes permitem designar e identificar entidades ou objectos.
- Por vezes, os nomes incluem informação relativa a propriedades/atributos dos objecto: Exemplos?
  - "xpto@foo.com", http://asc.di.fct.unl.pt/sd1

- Um serviço de nomes permite obter dados (atributos) sobre um entidade dado o seu nome.
- Um serviço de directório ou descoberta permite obter dados sobre as entidades que satisfazem um dada descrição.

O que são nomes ?

- Nomes são sequências de símbolos (geralmente codificados como sequências de bytes, bits) que designam entidades.
- Um nome tem de ser interpretado para se chegar (aos dados da) entidade. Este processo de interpretação diz-se resolver ou interpretar o nome (to resolve).
- A associação entre um nome e uma entidade designa-se por ligação (binding). Em geral, os nomes estão ligados a atributos da entidade e não directamente à entidade (ex: ????).
- A interpretação de um nome deve ser feita num contexto pois o mesmo nome em contextos diferentes pode designar objectos diferentes (exemplo?).
- Os nomes tomam várias formas conforme o nível do sistema em que são interpretados.

Nomes e identificadores

Designar - acção de apontar, indicar, mostrar, escolher

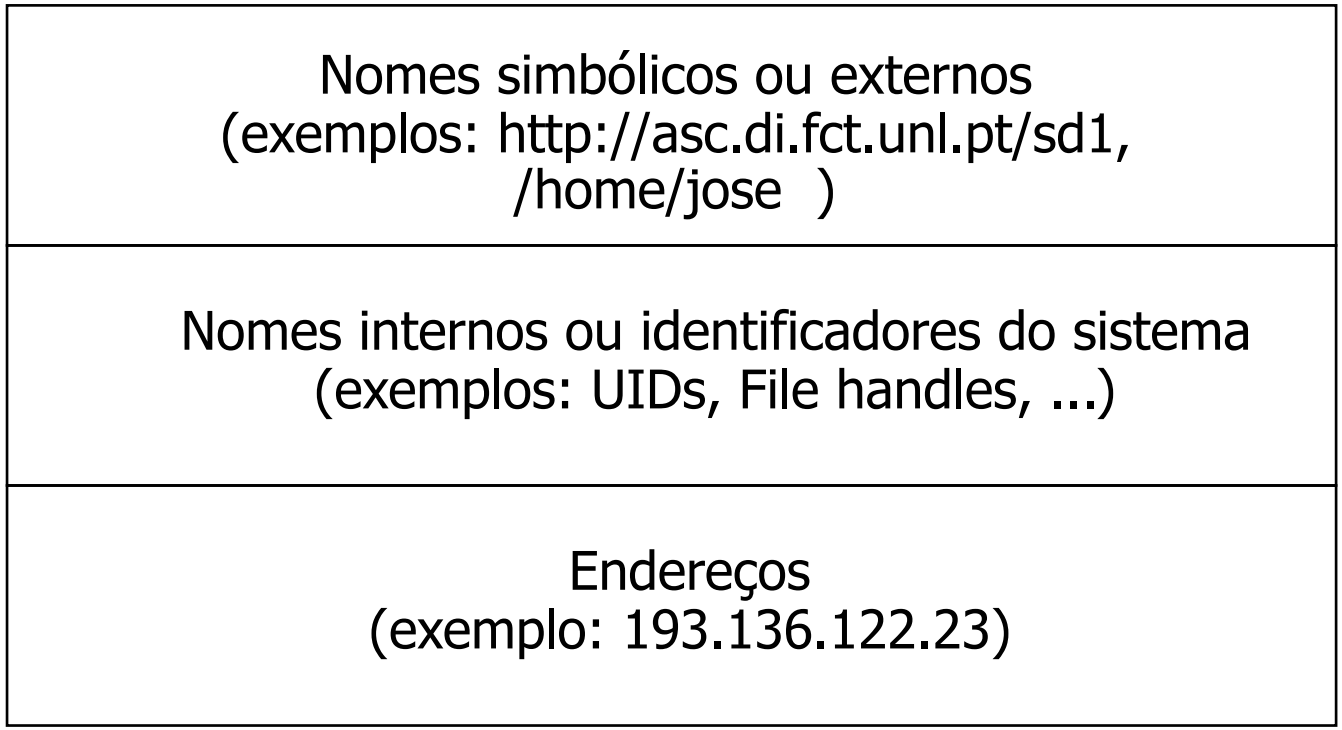
Identificar - tornar idêntico, o que faz com que uma coisa seja idêntica a outra, o que permite saber se duas coisas são distintas ou não

- Um nome permite designar uma entidade
  - Num dado contexto, uma entidade pode ser designada por mais do que um nome.
- Um identificador permite identificar uma entidade
  - Num dado contexto, uma entidade tem um e um só identificador.
    - Dois identificadores distintos identificam duas entidades diferentes. Se duas entidades tiverem o mesmo identificador, então são a mesma entidade.
  - Um identificador único permite identificar uma entidade de forma permanente
    - Se UID1=UID2, então Ent1=Ent2 para todo o sempre
    - Se UID1≠UID2, então Ent1 ≠ Ent2 para todo o sempre

Nomes, identificadores e endereços na prática

- Utiliza-se o termo nome simbólico, textual, orientado aos utilizadores, ou externo para designar ou identificar entidades através de nomes mnemónicos e legíveis para os utilizadores. Exemplos?
  - www.di.fct.unl.pt, /home/nmp/myfile
- Utiliza-se o termo identificador único sistema (UID) para designar sequências de símbolos, geralmente sem significado mnemónico, que permitem identificar entidades de forma (quase) permanente, ao nível interno do sistema distribuído. Exemplos?
  - AF.65.8F.89.1B.23.FF.45.A5.89.8B (uid de um objecto remoto)
- Utiliza-se o termo endereço para designar formas especiais de designação transitória, volátil ou temporária das entidades, geralmente associadas à localização das mesmas. Exemplo?
  - 10.0.0.12
  - Um endereço é um nome que dá acesso imediato a uma entidade.

Hierarquias de designação



Para se manipularem os objectos assim designados, os nomes são traduzidos de um espaço de designação noutro. Exemplos?

- abertura de um ficheiro em tempo de execução, determinação do endereço IP associado a um nome DNS, acesso a um objecto remoto, etc.

URLs, URNs, URIs

Vantagens dos URNs?

Vantagens dos URLs?

- Um URL (Uniform Resouce Locator) é um nome que indica a localização de um objecto

  scheme:scheme-specific-location

  Ex: http://asc.di.fct.unl.pt/sd1

  - Um URL é basicamente um endereço

- Um URN (Uniform Resource Name) é um nome que permite identificar um objecto independentemente da sua localização

  urn:nameSpace:nameSpace-specificName

  Ex: urn:ISBN:0-201-64233-8

  - Para aceder a um objecto será geralmente necessário existir um serviço de nomes que converta um URN num URL

- URLs e URNs são URIs (Uniform Resource Identifiers)

Geração de UIDs

- De forma geral os UIDs asseguram coerência referencial e propriedades dos identificadores devido a não serem reutilizados. Esta propriedade torna a sua geração mais difícil.
- A forma mais fácil de se gerarem UIDs consiste em utilizar geradores de números aleatórios ou estampilhas horárias.
  - Tais identificadores são potencialmente puros, no sentido em que não codificam propriedades das entidades que identificam, pelo que a sua utilização se pode revelar muito difícil. Porquê?
- Os UIDs não são geralmente visíveis aos utilizadores ou são complementados com "sugestões" ("hints"). Alternativamente usam-se nomes não puros, prefixando-os com endereços de sites, por exemplo.

Nomes globais *versus* nomes contextuais

- Todos os nomes são contextuais pelo que duas entidades computacionais só podem partilhar um nome se partilharem directa ou indirectamente um contexto comum de interpretação de nomes. Caso contrário não se "entenderiam".
- Um sistema distribuído necessita de um contexto comum a todas as entidades computacionais, para que essas entidades possam interagir. Os nomes relativos a esse contexto global dizem-se nomes globais, por oposição a nomes relativos a outros contextos mais limitados que se dizem nomes contextuais.
- Os nomes globais são independentes do contexto e podem passar-se livremente entre as diferentes entidades computacionais.

Exemplos

- Nomes globais:
  - nfs://phoenix.students.di.fct.unl.pt/home/joe
  - http://www.google.com
  - smb://fatdata.berkley.edu/students/johnDeere
  - rmi://bigserver.di.fct.unl.pt/computeService
  - 193.136.122.1
- Nomes contextuais:
  - 10.200.0.2
  - /home/joe
  - C:\database\students

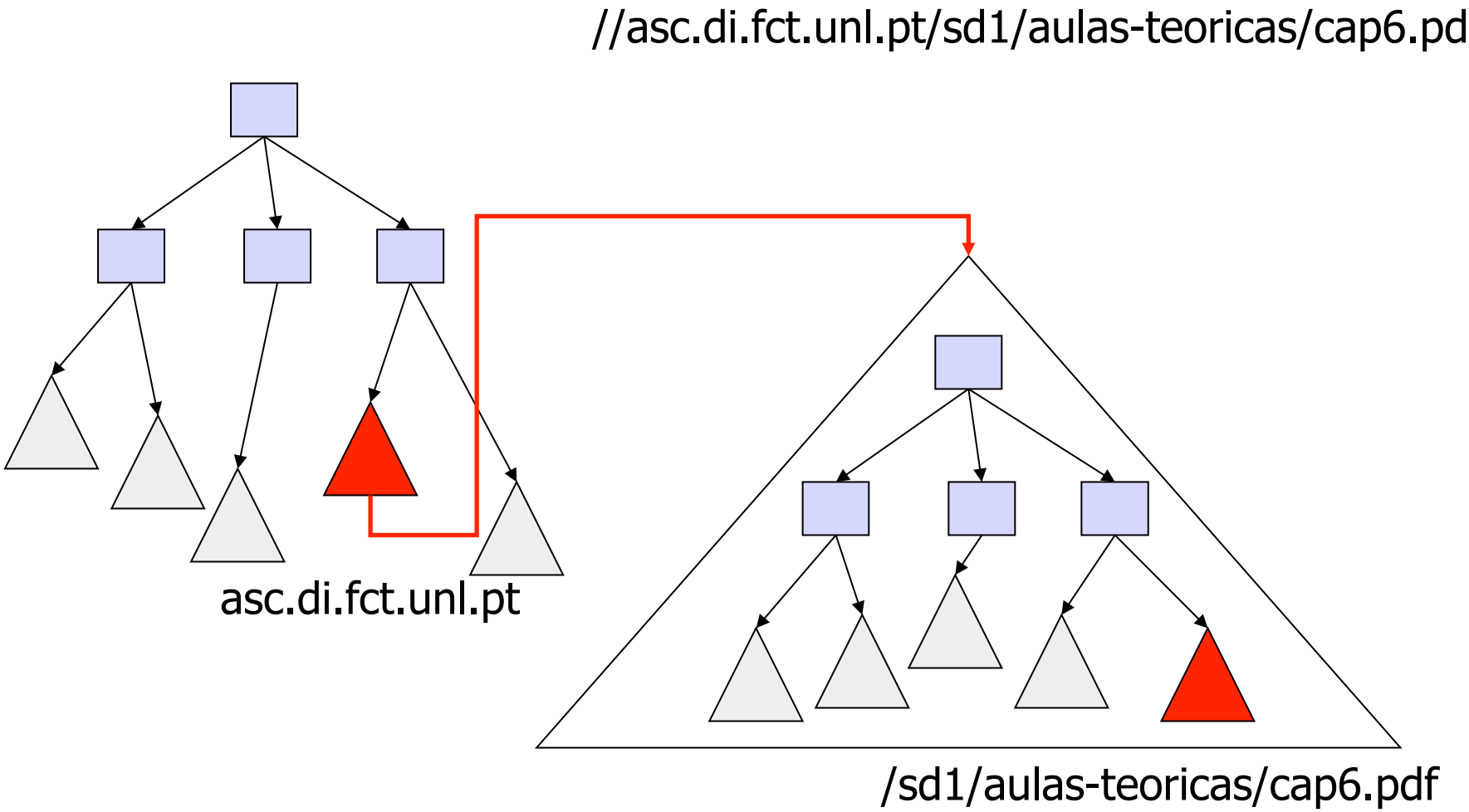
Organização do espaço de nomes

- Um domínio de nomeação é um espaço de nomes para o qual existe uma única autoridade administrativa para atribuir nomes.
- Em geral, num sistema coexistem vários contextos e domínios de designação e a tradução dos nomes envolve interpretações parciais em domínios do mesmo nível ou de níveis diferentes.
- Assim, estabelecem-se "ligações" entre os diferentes domínios e procedem-se a várias traduções de nomes até se chegar ao objecto. Exemplo: http:// asc.di.fct.unl.pt/sd1/aulas-teoricas/cap6.pdf

Quais os vários domínios ?

- Geralmente, os contextos iniciais são globais e são materializados por um serviço de nomes. Os outros contextos podem ser materializados por outros servidores de nomes ou directamente pelos servidores que gerem os objectos.

Generalização a vários espaços



Serviço de designação ou de nomes

- Um serviço de designação ou serviço de nomes é um serviço que permite obter um conjunto de atributos de uma entidade dado o seu nome. A resolução do nome é realizada a partir de um contexto de interpretação do nome.
- Cada um desses nomes pode designar utilizadores, servidores, serviços, objectos remotos, ficheiros, etc.
- Um serviço de nomes geralmente implementa uma base de dados (distribuída) que associa nomes aos atributos das entidades que estes designam.
- Existem duas operações importantes:
  - **Lookup**: dado um nome devolve os atributos associados ao mesmo
  - **Bind**: que associa um nome a um conjunto de atributos

No concreto

- O serviço de nomes pode ser um serviço isolado e específico ou estar integrado noutro serviço. Exemplos?
  - O DNS é um serviço de nomes puro pois não desempenha nenhum outro papel.
  - Os servidores de ficheiros integram igualmente um serviço de nomes dos ficheiros.

- Num serviço de nomes puro, as entidades existem independentemente do serviço de nomes. O serviço de nomes permite obter atributos que permitem depois aceder ao objecto.
- Nos serviços de nomes integrados, a designação e a gestão das entidades estão intimamente integradas. Geralmente, o serviço de nomes é usado internamente para aceder às entidades geridas pelo serviço.

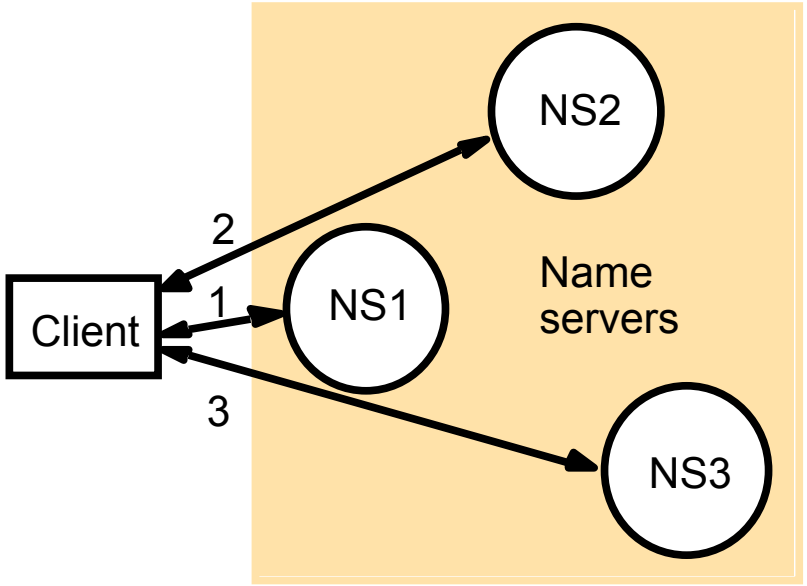
Na prática

- Existe um ou mais serviços de nomes puros, ao mais alto nível: as entidades nomeadas têm uma taxa de criação, supressão e de mudança de atributos relativamente lenta. Exemplo?
  - Serviços de nomes para domínios, utilizadores, serviços, máquinas, etc.

- No que toca às entidades cujas taxas de criação, supressão ou de mudança de estado são mais elevadas, opta-se geralmente por integrar o serviço de designação dessas entidades com o próprio serviço de gestão das mesmas. Exemplo?
  - Serviço de gestão de ficheiros integra a gestão dos nomes dos mesmos.

- Porquê esta diferença?
  - Com baixas taxas de alteração é mais fácil criar um serviço independente que seja fiável e global.

Resolução iterativa do nome pelo cliente

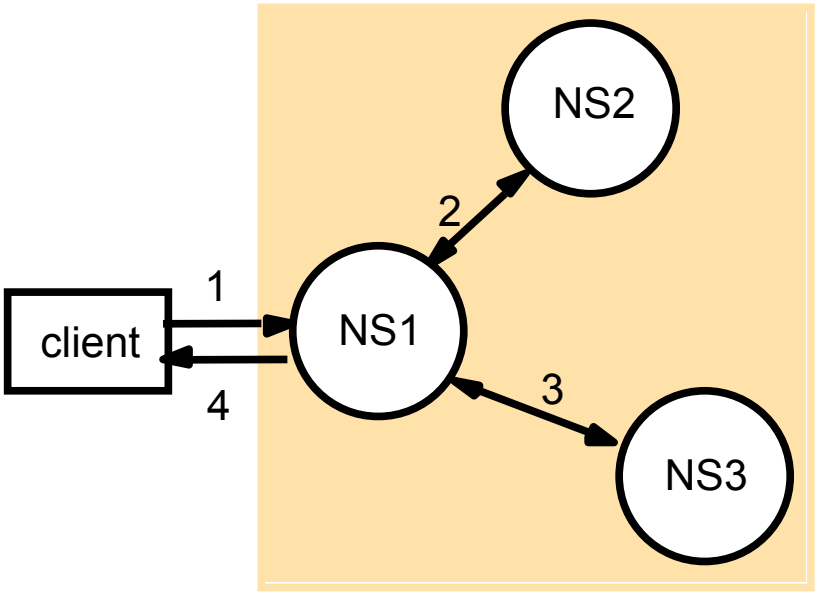


Características?

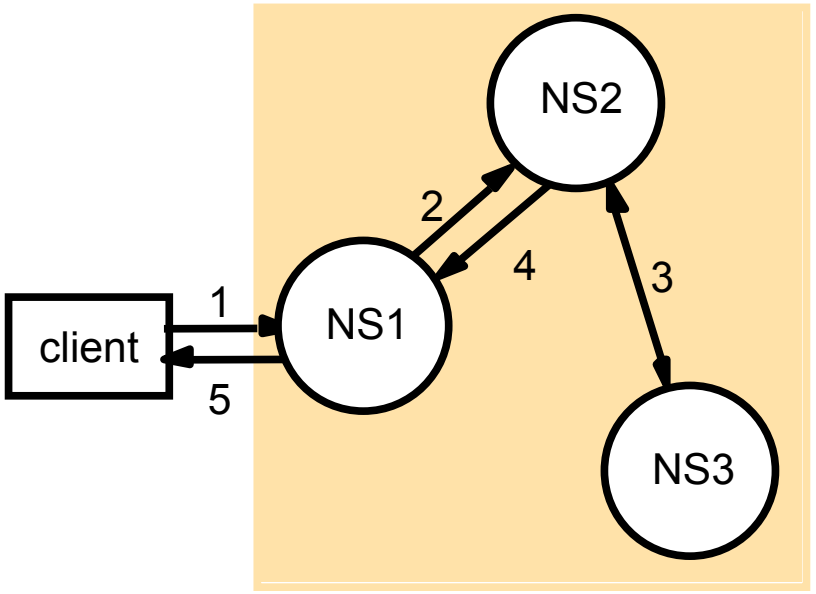
Servidor processa cada pedido rapidamente.
Resolução começa em servidores diferente.

- Cliente apresenta nome ao servidor local de nomes
  - Se servidor conhece o nome, devolve atributos
  - Se servidor não conhece o nome, indica outro servidor onde resolver o nome

Resolução recursiva do nome pelos servidores



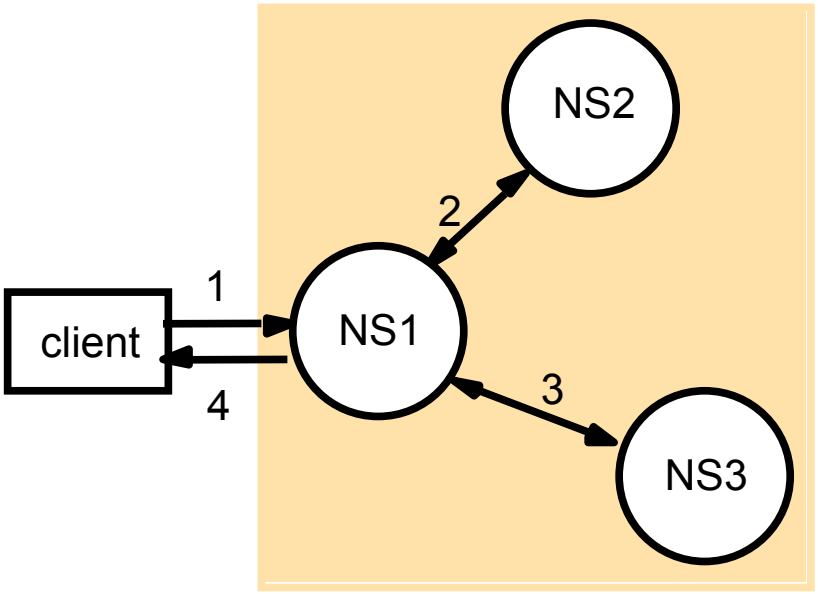
Non-recursive
server-controlled



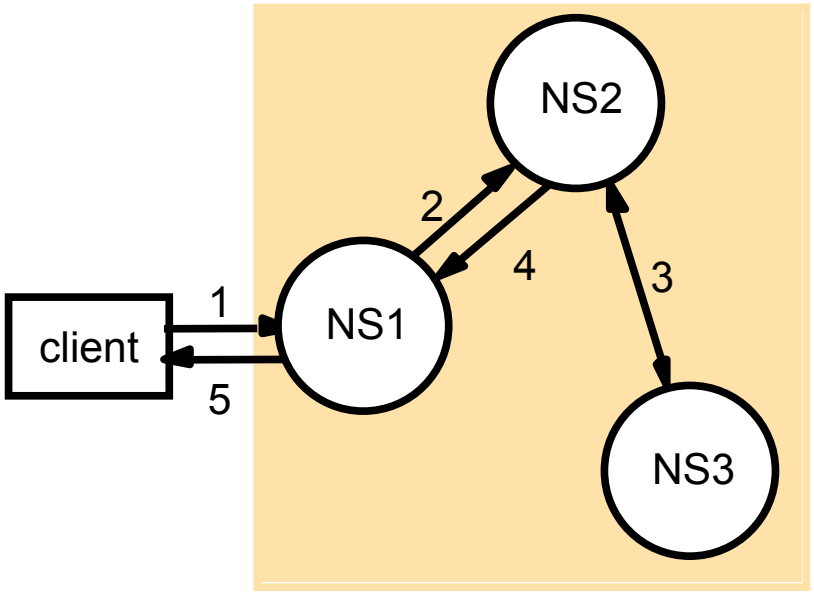
Recursive
server-controlled

- Cliente apresenta nome ao servidor local de nomes
  - Se servidor conhece o nome, devolve atributos
  - Se servidor não conhece o nome, o próprio servidor resolve o nome iterativamente ou recursivamente

Resolução recursiva do nome pelos servidores



Non-recursive server-controlled



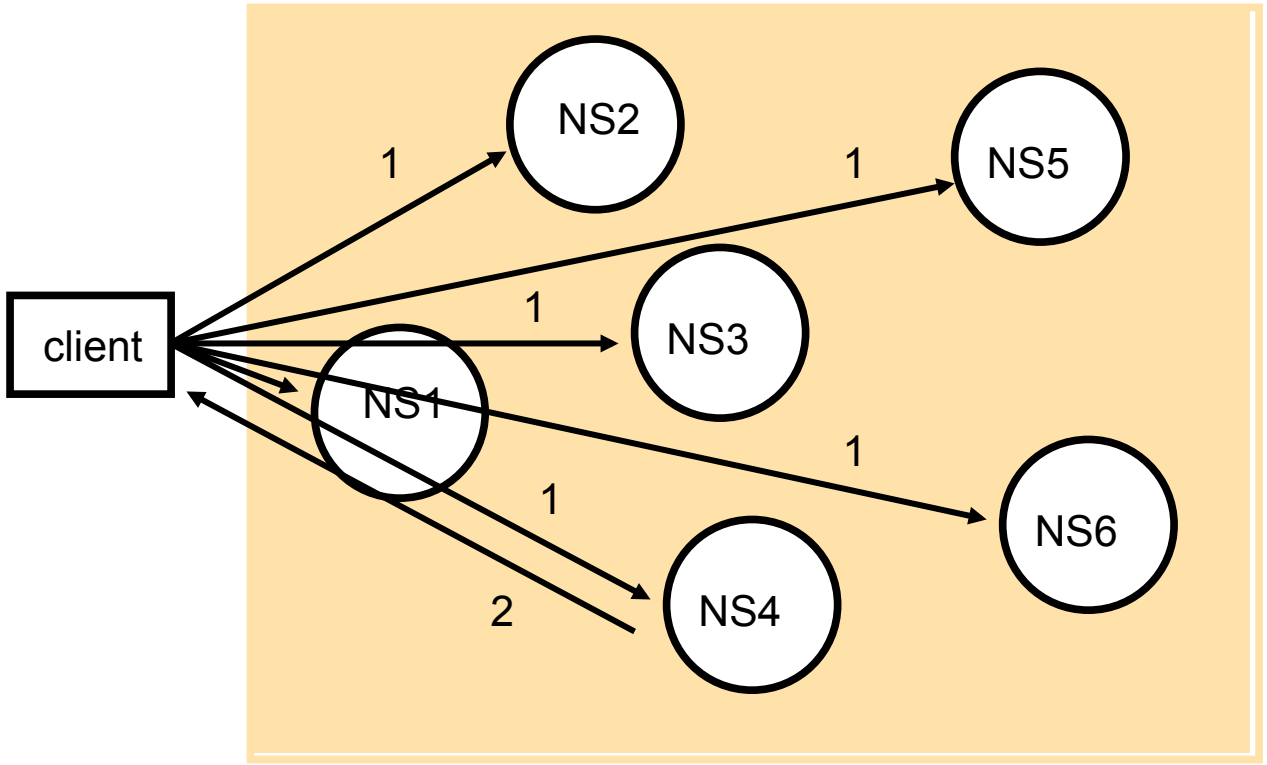
Recursive

- Cliente apresenta nome ao servido
  - Se servidor conhece o nome, dev
  - Se servidor não conhece o nome, recursivamente

Características?

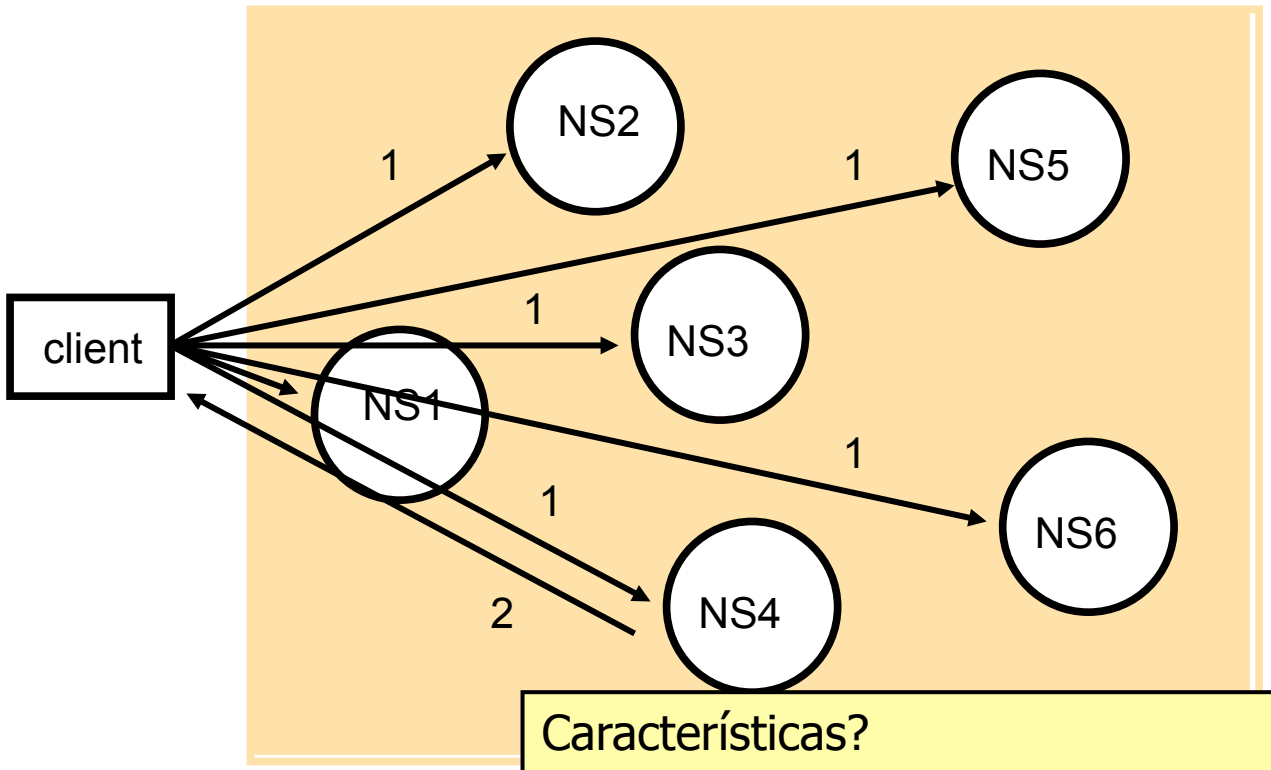
Cada pedido fica pendente até estar resolvido (ocupando recursos).
Pode permitir contornar problemas de permissões.
Resolução começa em servidores diferente.

Resolução por multicasting



- Cliente envia mensagem *multicast* aos servidores de nomes
  - O servidor de nomes que conhece o nome, devolve os atributos

Resolução por multicasting



- Cliente envia mensagem *multicast*
  - O servidor de nomes que conhece

Características?

Apenas possível em ambientes que suportam *multicast*.
Todos os servidores de nomes vêem todos os pedidos.

Mascaramento das falhas e *caching*

- Serviço de nomes é muitas vezes fundamental no acesso ao sistema.
- Para mascarar falhas e fornecer elevada disponibilidade pode recorrer-se à replicação e *caching* de contextos.
  - Ao colocar cópias da informação (réplicas) em servidores distintos é possível tolerar falhas dos servidores e da rede.
  - Como é que os clientes sabem quais são os servidores?
  - Problema idêntico a saber um servidor - por exemplo, os atributos de um contexto contém a lista das réplicas.
- O *caching* e replicação também permitem melhorar a escalabilidade (e responder a um elevado número de pedidos).
- Problema?
  - Manter coerência entre as réplicas e a informação "oficial".

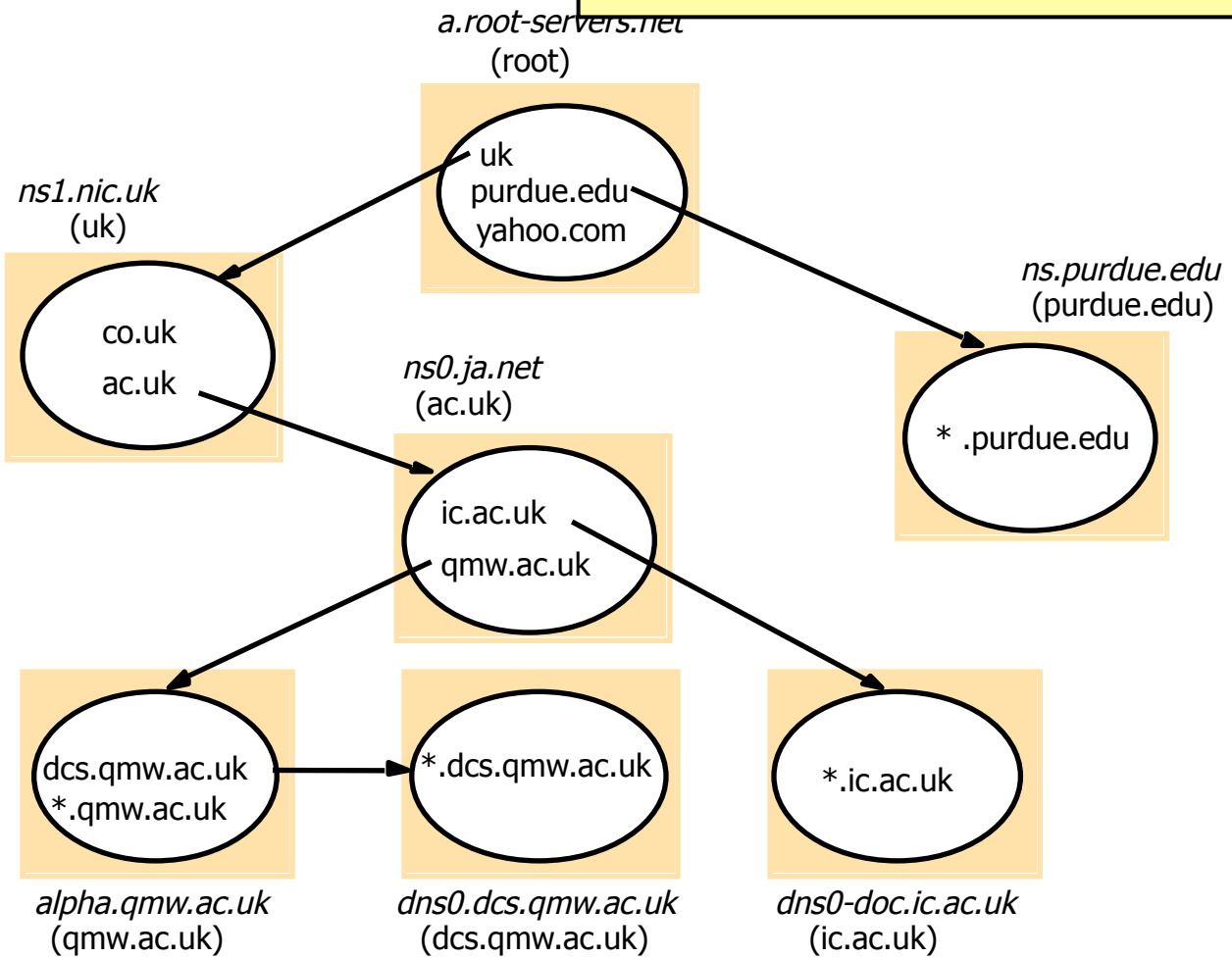
Coerência entre réplicas e das caches

- Muitos serviços de nomes não asseguram coerência total, isto é, permitem que os clientes observem incoerências momentâneas. O sistema só assegura que as incoerências não se vão manter e que todas as réplicas tendem para um estado coerente.
- Propriedades geralmente assumidas:
  - os clientes suportam algum grau de incoerência.
  - os valores registados no serviço evoluem lentamente.
- São frequentes as seguintes soluções:
  - Replicação do tipo primário/secundários com propagação assíncrona das actualizações do primário para o secundário;
  - Mecanismos de *caching* em que os valores são retirados da cache por um mecanismo de "envelhecimento" ou TTL, controlado pelos administradores dos contextos de designação. Trata-se de um mecanismo "optimista".

Exemplo do DNS

DNS:
Sistema que mantém base de dados distribuída com informação de máquinas na internet.
Protocolo de acesso ao sistema.

Note: Name server names are in italics, and the corresponding domains are in parentheses.
Arrows denote name server entries



Os atributos no DNS - *resource records* do DNS

Record type	*Meaning*	*Main contents*
A	**A computer address**	**IP number**
NS	**An authoritative name server**	**Domain name for server**
CNAME	**The canonical name for an alias**	**Domain name for alias**
SOA	**Marks the start of data for a zone**	**Parameters governing the zone**
WKS	**A well-known service description**	**List of service names and protocols**
PTR	**Domain name pointer (reverse lookups)**	**Domain name**
HINFO	**Host information**	**Machine architecture and operating system**
MX	**Mail exchange**	**List of < *preference, host* > pairs**
TXT	**Text string**	**Arbitrary text**

Registos DNS (DNS Resource Records)

DNS: db distribuída com registos (RR)

Formato de um RR: `(name, type, value, ttl)`

- Type=A
  - `O nome é um` hostname
  - `O valor é um endereço IP do host`
- Type=NS
  - `O nome é um domínio` (e.g. foo.com)
  - `O valor é o hostname de um servidor do domínio`
- Type=CNAME
  - `O nome` é um alias para o nome "canónico" (o nome real)
  - `O valor é o nome canónico`
- Type=MX
  - `O valor é o nome de um mail server do domínio e a respectiva prioridade`

Arquitectura DNS

- Base de dados particionada por múltiplos servidores
  - Organização hierárquica dos servidores
    - Conjunto de servidores replicam informação da raiz da árvore

Domínios de topo

- Domínios genéricos: com, edu, org, mil, etc.
- Domínios nacionais: pt, uk, etc.

Arquitectura DNS

- Base de dados particionada por múltiplos servidores
  - Organização hierárquica dos servidores
    - Conjunto de servidores replicam informação da raiz da árvore
  - Informação sobre cada zona mantida em pelo menos dois servidores (de forma authoritative – em princípio actual)
    - Replicação primary/backup: backup lê informação periodicamente do primário
    - Frequênca da verificação é parâmetro da configuração
  - Qualquer servidor pode fazer cache de informação de outros servidores
    - Entradas têm time-to-live
- Clientes contactam servidores (pode ser dada lista de servidores)
  - Protocolo tipicamente em UDP
  - Cliente pode solicitar navegação recursiva ou iterativa. Servidor é livre de respeitar pedido.
  - Clientes podem fazer caching dos resultados

Esta arquitectura garante coerência da informação?

Porque funciona bem?

DNS: que futuro ?

Introducing Google Public DNS

12/03/2009 08:35:00 AM

When you type www.wikipedia.org into your browser's address bar, you expect nothing less than to be taken to Wikipedia. Chances are you're not giving much thought to the work being done in the background by the Domain Name System, or DNS.

Today, as part of our ongoing effort to make the web faster, we're launching our own public DNS resolver called Google Public DNS, and we invite you to try it out.

Most of us aren't familiar with DNS because it's often handled automatically by our Internet Service Provider (ISP), but it provides an essential function for the web. You could think of it as the switchboard of the Internet, converting easy-to-remember domain names — e.g., www.google.com — into the unique Internet Protocol (IP) numbers — e.g., 74.125.45.100 — that computers use to communicate with one another.

The average Internet user ends up performing hundreds of DNS lookups each day, and some complex pages require multiple DNS lookups before they start loading. This can slow down the browsing experience. Our research has shown that speed matters to Internet users, so over the past several months our engineers have been working to make improvements to our public DNS resolver to make users' web-surfing experiences faster, safer and more reliable. You can read about the specific technical improvements we've made in our product documentation and get installation instructions from our product website.

Directórios e serviços de descoberta

- Um serviço de directório ou descoberta é um serviço que permite obter os atributos de uma entidade que satisfaz uma dada descrição (dada como um sub-conjunto de atributos).
  - Nestes sistemas, o nome é apenas um dos atributos.
  - Exemplo: qual é o endereço IP da impressora a cores do edifício II ?

Designação através de atributos

- Um conjunto de atributos pode funcionar como "apontador" de uma entidade de uma forma mais potente que os nomes clássicos
  - Por exemplo, uma pessoa quando se dirige a uma agência de um banco pode dizer que deseja falar com o "responsável pela agência", ou com o "caixa".
  - Estes "apontadores" são abreviaturas de "a pessoa cujo atributo 'função' tem o valor 'gerente' ou 'caixa' ".
- Um mecanismo de designação baseado em atributos pode facilmente ser usado para designar conjuntos de entidades.

Serviços de nomes vs. serviços de directório

- Serviços de nomes
  - Nomes mais simples
- Serviços de directório
  - Atributos mais poderosos
    - Necessário definir atributos
  - Mais simples obter serviços redundantes
  - Mais simples para integração de um computador num *ambiente novo*

Serviços de descoberta e mobilidade

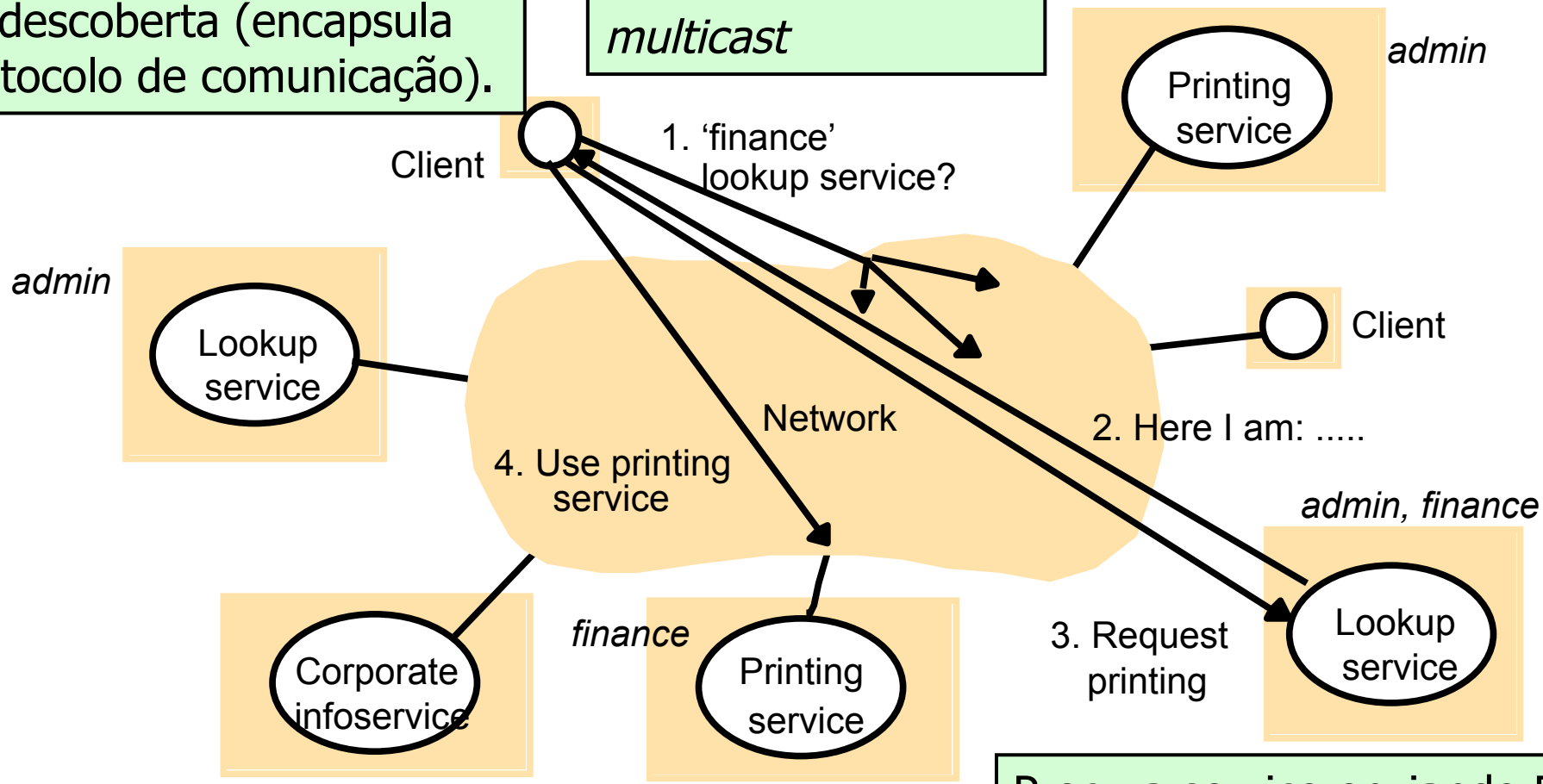
- Num ambiente com mobilidade ou em que a associação entre objectos se faz de forma espontânea, os utilizadores tendem a não conhecerem previamente o nome dos recursos.
- Um serviço de descoberta é um serviço de directório que regista serviços disponíveis numa rede.
- Exemplos?
- Um serviço de descoberta pode ser usado:
  - por um portátil para encontrar uma impressora, ou um projector de slides.
  - por um frigorifico para encontrar o sistema de alarme para comunicar que está sem corrente há mais de 10 minutos.
- Nos serviços de descoberta uma pesquisa está geralmente sujeita a um contexto ou âmbito, por exemplo, uma rede local, uma dada sala, uma localização geográfica, ...

Descoberta de serviços em Jini

Acede ao serviço usando objecto recebido do serviço de descoberta (encapsula protocolo de comunicação).

Procura serviço de descoberta enviando *multicast*

Diversos serviços de *lookup* disponíveis, com vários scopes



Serviços registam-se com um ou mais serviços de *lookup*

Procura serviço enviando RMI para serviço de descoberta.

Serviços de descoberta: arquitecturas possíveis

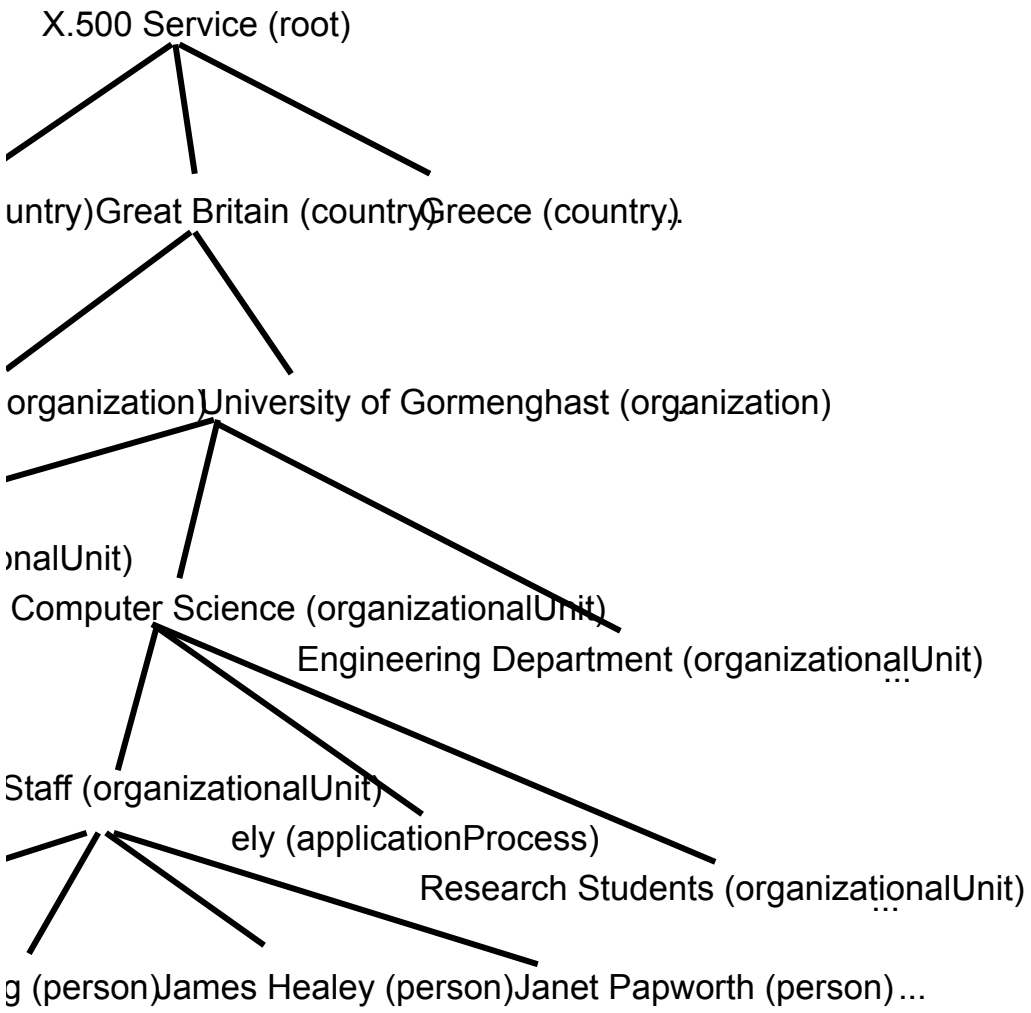
- Com servidor de directório. Como descobrir servidor?
  - Utilização de multicast
    - E.g. não comum: Bluetooth: well-known frequency-hopping sequence: dispositivos e clientes percorrem as frequências disponíveis com diferentes "velocidade"
- Sem servidor de directório
  - Alternativa 1: Cliente envia multicast com pergunta. Servidores respondem directamente caso correspondam à descrição.
  - Alternativa 2: Servidores enviam multicast periódico com a sua descrição. Clientes guardam informação para responder às perguntas locais.

Serviços de descoberta

- Os serviços registados nos servidores de descoberta tendem a ser voláteis. Como lidar com esta volatilidade?
- Arquitectura com servidor:
  - Utilização de *leases*: servidores devem renovar o seu registo periodicamente.
- Arquitectura sem servidor:
  - Sem anúncios periódicos, não é necessário fazer nada
  - Com anúncios periódicos, após um dado período, considera-se que um serviço que não se anunciou deixou de estar disponível

X.500 Directory Service

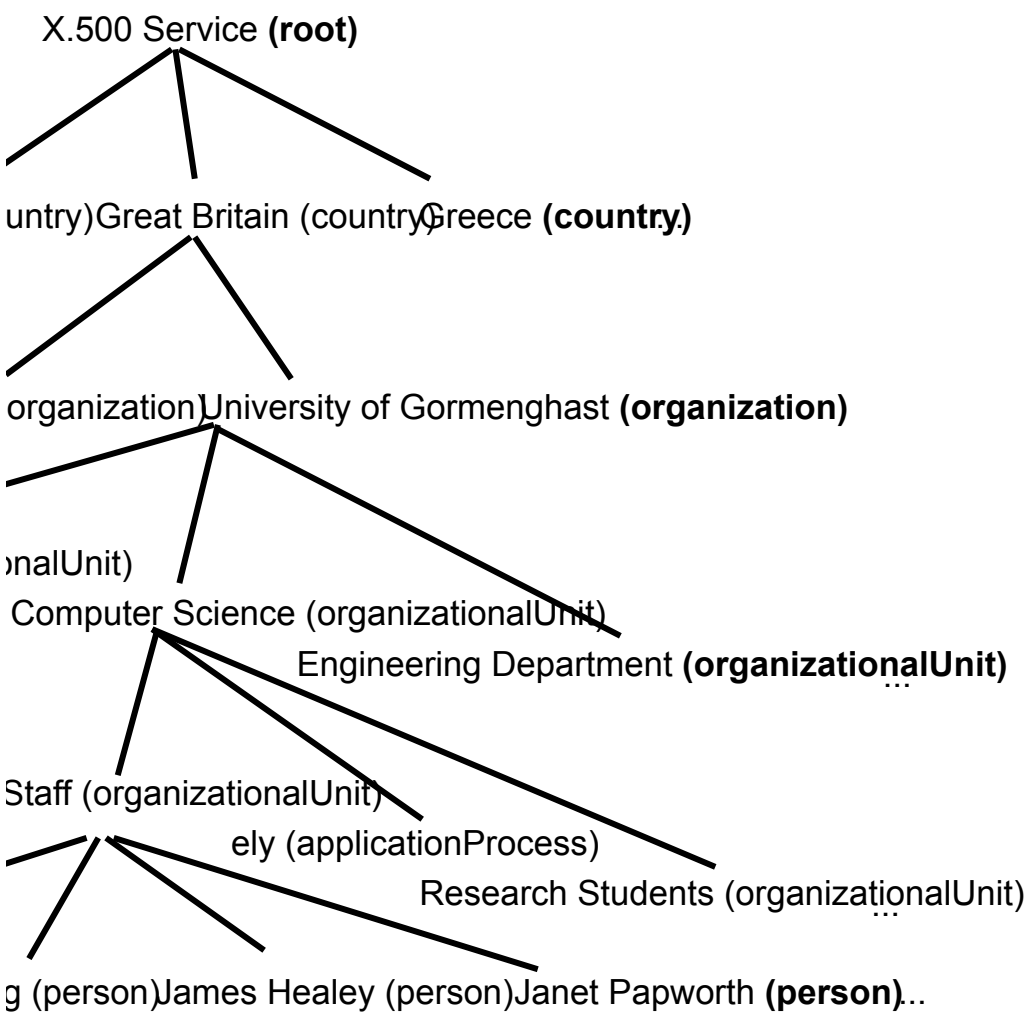
- X.500 é um serviço de directório desenhado para guardar informação sobre recursos e utilizadores
- A informação está organizada hierarquicamente
- À árvore de nomes chama-se *Directory Information Tree*
- À estrutura directório completa, incluindo os dados associados com os vários nós, chama-se *Directory Information Base* (DIB)



X.500 Directory Service

- A DIB (Directory Information Base) é uma colecção de *entries*. Uma entry contém informação sobre objectos, isto é, uma *entry* é uma colecção de atributos de um objecto do mundo real.
  - DIB: entry 1, …, entry N
  - Entry: atribute 1, …, atribute N
  - Atribute: type, value
  - Value: distinguished value, value

- Um dos atributos essenciais de uma *entry* é o atributo *objectClass* que é um atributo obrigatório que define o tipo da *entry*.



Exemplo de uma DIB Entry

info

Alice Flintstone, Departmental Staff, Department of Computer Science, University of Gormenghast, GB

commonName

Alice.L.Flintstone
Alice.Flintstone
Alice Flintstone
A. Flintstone

surname

Flintstone

telephoneNumber

+44 986 33 4604

uid

alf

mail

alf@dcs.gormenghast.ac.uk
Alice.Flintstone@dcs.gormenghast.ac.uk

roomNumber

Z42

userClass

Research Fellow

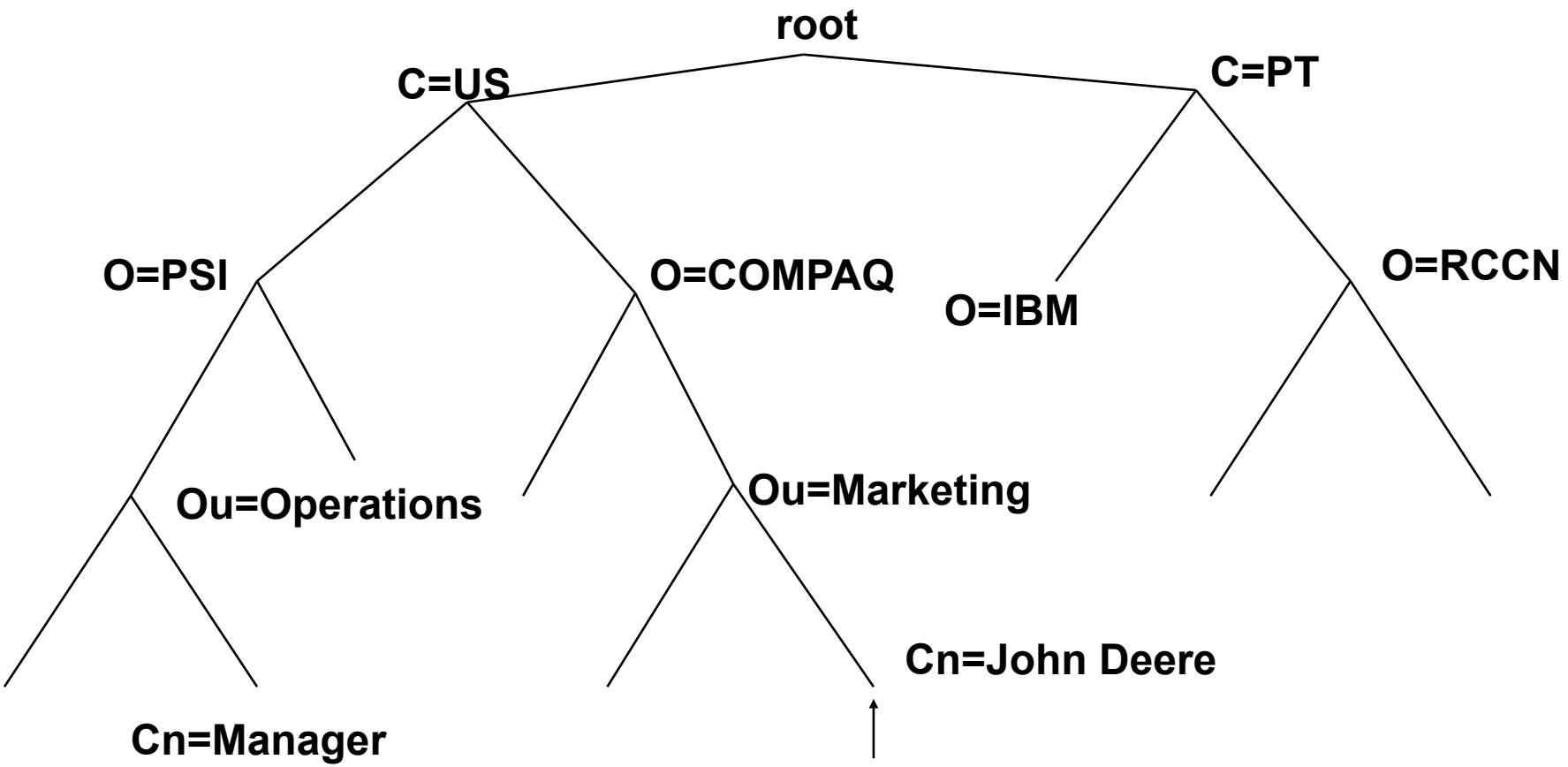
Tipos dos atributos

- Existe um sistema de tipos de entries que utiliza herança e é extensível. O tipo de uma *entry* define, através de ASN.1, os seus atributos obrigatórios e opcionais. O conjunto dos tipos da DIB constitui o esquema da DIB.
- A DIB está organizada hierarquicamente. Cada *entry* é identificada por um DN ou Distinguished Name que é um identificador global. Um DN é um conjunto de atributos que permite identificar a *entry*, localizando-a na árvore.
- Cada *entry* pode também ser designada por um RDN, isto é, Relative Distinguished Name, isto é, um conjunto de atributos que permite distinguir a *entry* das suas irmãs na árvore.

Alguns exemplos de atributos

Nome do atributo	Abreviatura	Sintaxe do valor
commonName	cn	string
countryName	c	ISO3166 code
localityName	l	string
organizationName	o	string
organizationalUnitName	ou	string
postalAddress		special
postalCode		string
userPassword		string
telephoneNumber		special
surname	sn	string

Exemplo



DN = /C=US/o=COMPAQ/Ou=Marketing/Cn=John Deere

(DN = Distinguished Name)

Modelos de segurança

- Permite vários esquema de controlo de acessos
  - Nenhuma
  - Simples (password)
  - Strong (Public Key Encryption) - baseada nos certificados X.509
- Suporta também ACLs para controlo dos acessos e modificações

Modelo organizacional

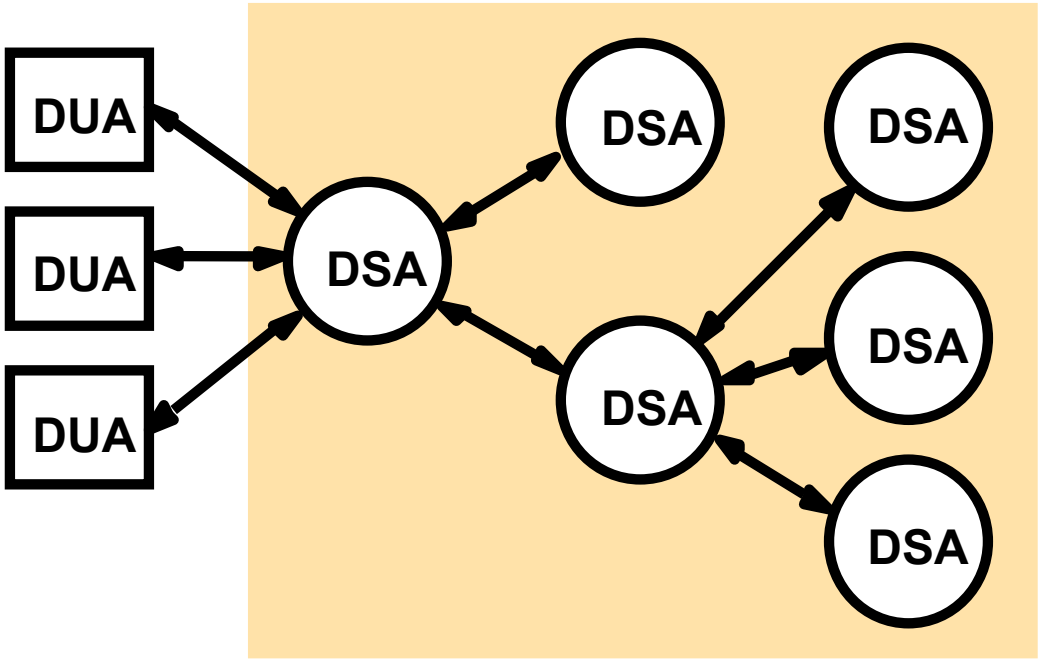
- Existem domínios de gestão da informação – são os *Directory Management Domains* (DMDs)

- Um DMD é materializados por um conjunto de DSAs (Directory System Agents), que gerem uma sub-árvore da árvore de informação (DIT)

- Os DSAs definem o esquema e organização dessa sub-árvore e a forma como a mesma é vista do exterior

Arquitectura

- Conjunto de servidores (DSA) acessíveis pelos User Agents (UA)
- Está prevista a resolução iterativa, recursiva, transitiva server based e por broadcasting dos nomes
- A DIT está particionada em *naming contexts*.
- Um DSA gere uma *information tree* com um ou mais *naming contexts*. Cada DSA conhece os DNs da raíz dos seus *naming contexts* e os RDNs da suas *entries*, logo conhece os DNs das suas *entries*.



Operadores, utilizações e estado

- Read, Compare (verifica o valor de um atributo de uma entry), List, Search, Add, Remove, Modify (atributes), Modify RDN (como move), …
- Utilizações típicas:
  - páginas brancas das instituições
  - DNS para hosts e e-mail
  - servidores de chaves públicas
  - bases de dados bibliográficas
  - listas de serviços, etc.
- Alguns problemas:
  - definição de um conceito de nome legível (e usável)
  - estrutura mundial da DIT e uniformização dos tipos usados
  - algoritmos de pesquisa e navegação genéricos
- A sua generalidade e complexidade tem impedido a sua divulgação a nível mundial para substituir o DNS como era a intenção inicial.

Light-weighted Directory Access Protocol

- A única implementação disponível de forma alargada e com uma utilização generalizada da norma X.500 é como directório interno das organizações.
- O protocolo de acesso é uma versão simplificada do protocolo de acesso entre o DUA e o DSA chamado LDAP ou Light-weighted Directory Access Protocol.
- O LDAP apenas usa uma representação textual dos atributos e dos valores e não ASN.1 nem as regras de codificação associadas como está previsto nas normas X. 500 completas.
- Várias implementações do protocolo LDAP e de serviços de directoria a ele associados são disponibilizadas em "open source" (Open LDAP) e por fabricantes como a Novell e Microsoft ("Active Directory").

Para saber mais

- G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems - Concepts and Design, Addison-Wesley, 4th Edition, 2005
  - Capítulo 9.
  - Capítulo 16 – secção 16.2.1

Sistemas Distribuídos

Capítulo 8

Introdução aos sistemas distribuídos de gestão de ficheiros

Nota prévia

- A apresentação utiliza algumas das figuras livro de base do curso

  G. Coulouris, J. Dollimore and T. Kindberg,
  Distributed Systems - Concepts and Design,
  Addison-Wesley, 4th Edition, 2005

Sistemas de ficheiros distribuídos

- Um sistema de gestão de ficheiros distribuídos fornece um serviço de acesso a ficheiros semelhante ao de um sistema de ficheiros normal, mas estendido a um conjunto de máquinas ligadas em rede. Normalmente permite:
  - Partilha de ficheiros por vários utilizadores
  - Melhor qualidade de serviço do que a disponível localmente
    - Maior dimensão do espaço disponível
    - Melhor tolerância a falhas
    - Desempenho semelhante ou superior

Algumas questões envolvidas na distribuição

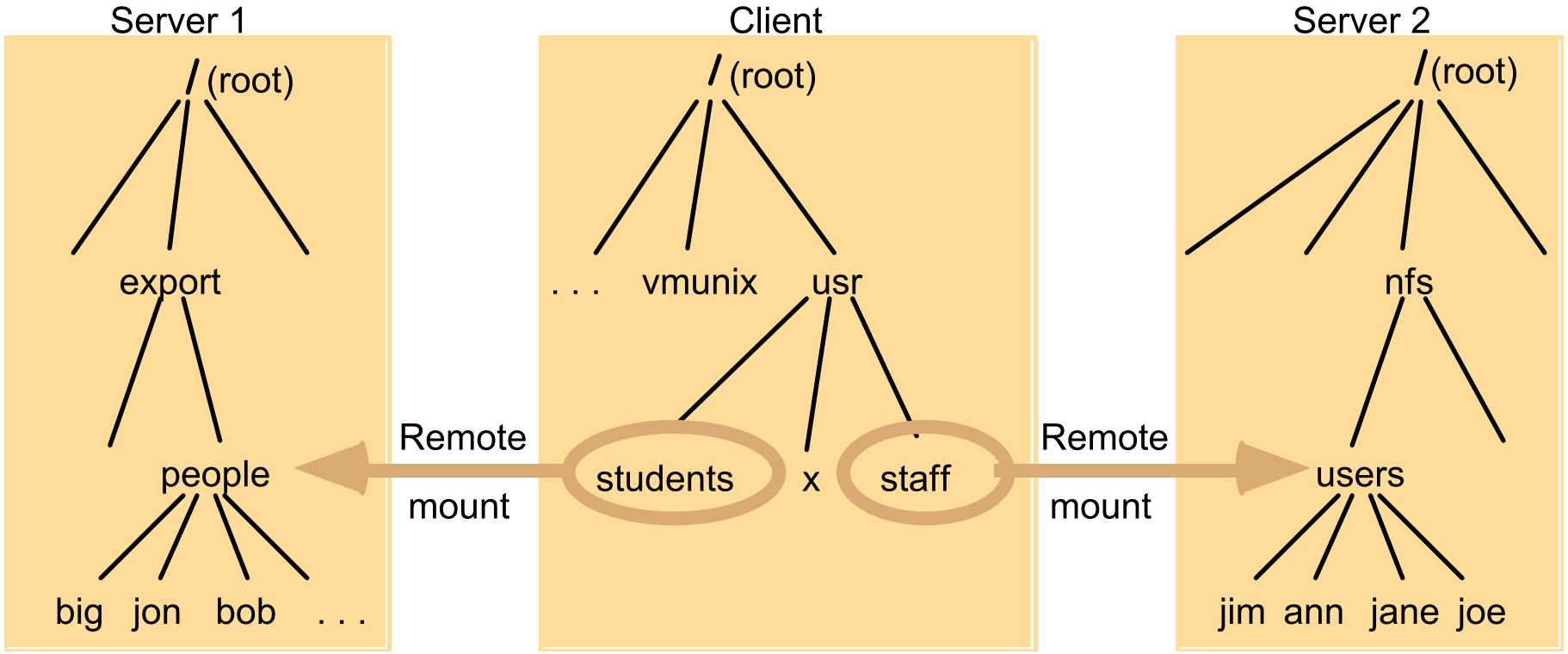
- Integração: como integrar no sistema de ficheiro local?
- Designação: como designar os ficheiros remotos?
  - Externamente: ao nível das aplicações
  - Internamente: no sistema
- Modelo de acesso: dois extremos: acesso completamente remoto ou acesso completamente local (usando *cache*).
  - Partilha de ficheiros: problemas relacionados com o controlo da concorrência e modelo de consistência fornecido

Integração no sistema local

- Os ficheiros dum sistema distribuído de ficheiros devem ser acedidos localmente numa máquina
- Problema: Qual a granularidade da partilha?
- Solução típica: sub-árvore do sistema de ficheiros

- Problema: Como providenciar acesso aos ficheiros?
- Solução: integrar a sub-árvore na hierarquia local
  - Windows: definindo uma nova *drive*
  - Unix-like: através do mecanismo de *mount*

Exemplo: mount remoto e designação no NFS

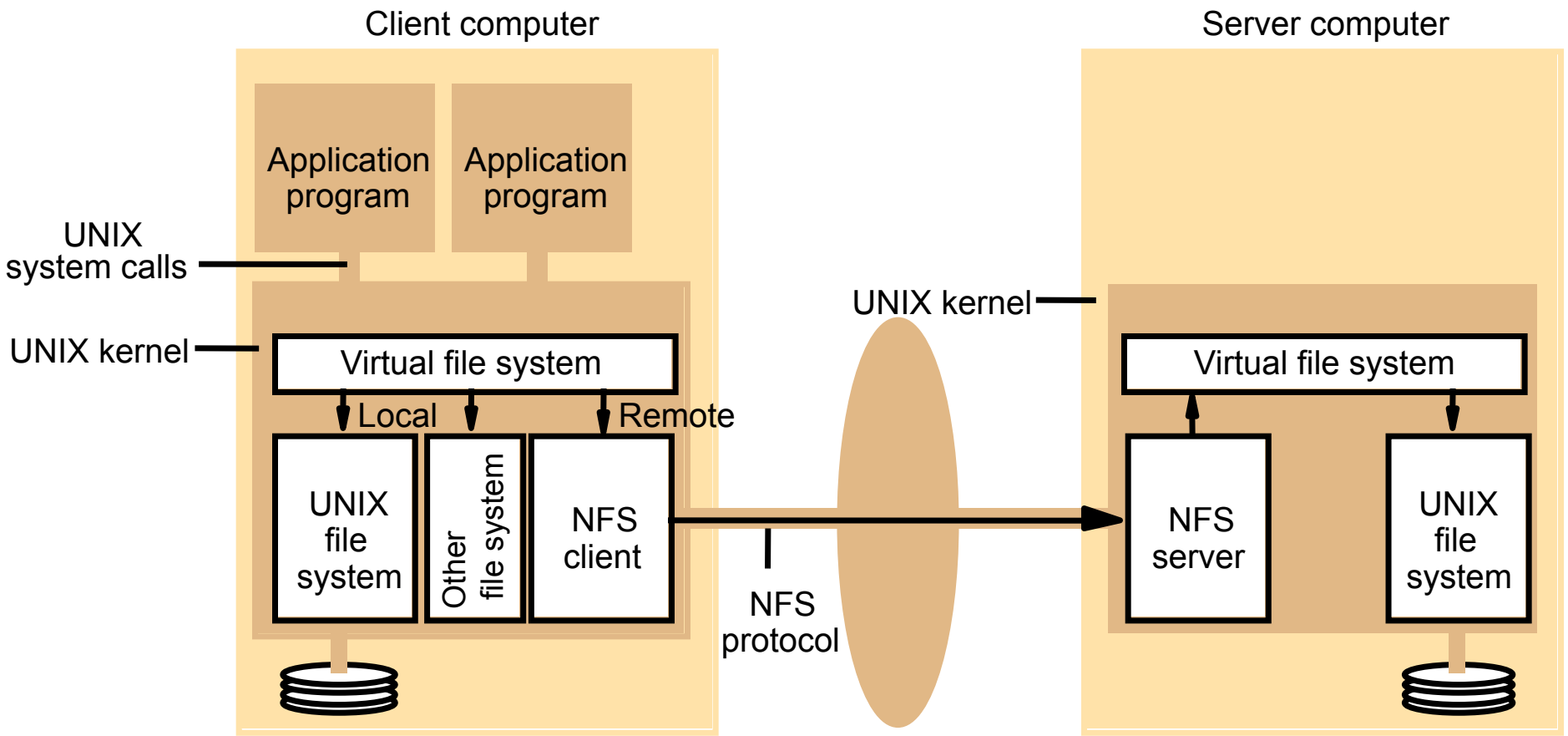


- Cada servidor exporta um conjunto de directorias.
- O cliente pode montar uma directoria remota (caso tenha permissões) numa directoria local (ex.: /usr/students no cliente monta a árvore /export/people no Server 1)

Integração no sistema local (cont.)

- Como fazer a integração ?
- Solução: os sistemas de operação normalmente suportam mecanismos para definir novos sistemas de ficheiros
  - Unix-like: VFS
  - Windows: IFS

Arquitectura



Designação dos ficheiros

Tipos dos nomes dos ficheiros:

- Nomes simbólicos
  (exemplo: /x/y/p/q)

- Nomes internos ou sistema
  (exemplos: FileId, File-handles, UFIDs)

- Endereços internos ao servidor:
  (exemplo: nº device, nº i-node, endereço de bloco, …)

Tipos dos nomes simbólicos

- Nomes globais. Podem ser passados entre máquinas mas implicam a introdução de uma super-raiz do *file system*.
  Uma forma popular é: "//servername/ficheiro"
  - Exemplo: //asc.di.fct.unl.pt/home/foobar
- Nomes contextuais. Todos os nomes são interpretados a partir de um *file system* local.
  - Fáceis de implementar mas difícil passar um nome entre duas máquinas
  - Ligações (mounts)
    - Um mesmo nome em duas máquinas pode referir o mesmo ou diferentes ficheiros
    - Nomes diferentes em duas máquinas podem referir o mesmo ficheiro

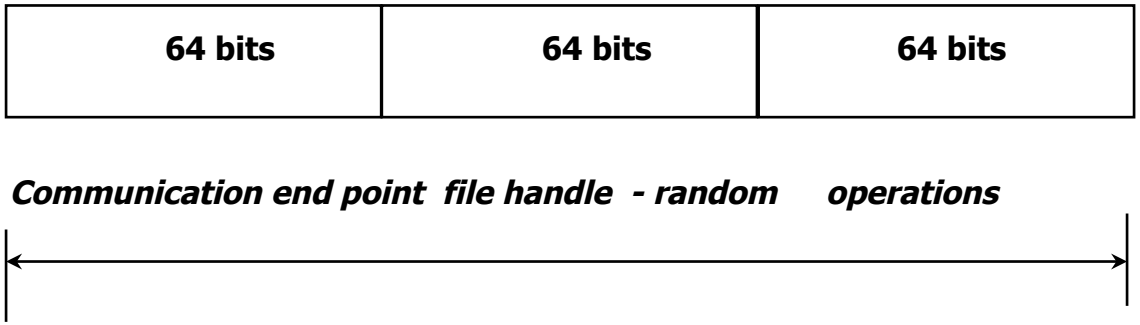
Características dos nomes internos

- Para que os clientes e os servidores possam memorizar UFIDs (ou *file handles*) sem que os mesmos possam dar origem a ambiguidades, estes devem ser:
  - únicos no tempo
  - únicos no espaço

Que propriedades tem um NFS *file handle* ?
[file system number, file i-node, i-node generation number],
endereço IP do servidor, porta do servidor

Capacidades e UFIDS

- É possível definir identificadores de baixo nível cuja posse garante o direito de realizar operações sobre um objecto. Tratam-se de identificadores que podem ser copiados, mas que não podem ser forjados. Este tipo de identificadores têm as vantagens de uma capacidade (e os defeitos também).
- Exemplo:



UFID - Unique File Identifier

Os dois campos à direita têm de ser cifrados ou "assinados" para não serem alteráveis.

Características do método

Vantagens

- Favorece a modularidade e a independência das componentes
- Favorece soluções *stateless*, mas

Desvantagens

- A revogação ou simples modificação dos direitos é impossível ou simplesmente difícil
- Exige cuidados especiais para impedir a cópia das capacidades ou o acesso indevido às mesmas
- Geralmente usam-se como complemento de um sistema baseado em ACLs, através de *short-lived* ou *session capabilities*.

Servidores *stateless* vs. Servidores stateful

- Um sistemas de ficheiros distribuídos pode ser desenhado recorrendo a servidores *stateless* ou *stateful*.
- Um servidor que não mantém informação sobre os clientes chama-se *stateless*.
- Um servidor que mantém informação sobre os clientes chama-se *stateful.*

Servidores *stateless*

- Considere um sistema de ficheiros local. Neste, o sistema mantém informação sobre o estado do cliente, ficheiros abertos, posição de leitura/escrita nos ficheiros, etc. Como desenhar um sistema distribuído *stateless*, em que o servidor não mantém nenhuma desta informação?
- Criar as operações de modo a que incluam toda a informação necessária à sua execução

Exemplo: operações NFS

Nomes são resolvidos através de uma sequência de operações de *lookup*, uma para cada elemento do caminho. Porquê?

lookup(dirfh, name) -> fh, attr	**Returns file handle and attributes for the file *name* in the directory *dirfh*.**
create(dirfh, name, attr) -> newfh, attr	**Creates a new file name in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.**
remove(dirfh, name) status	**Removes file name from directory *dirfh*.**
getattr(fh) -> attr	**Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)**
setattr(fh, attr) -> attr	**Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.**
read(fh, offset, count) -> attr, data	**Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.**
write(fh, offset, count, data) -> attr	**Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.**

fh, newfh são identificadores de ficheiros opacos

Continua ...

Exemplo: operações NFS

lookup(dirfh, name) -> fh, attr	**Returns file handle and attributes for the file *name* in the directory *dirfh*.**
create(dirfh, name, attr) -> newfh, attr	**Creates a new file name in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.**
remove(dirfh, name) status	**Removes file name from directory *dirfh*.**
getattr(fh) -> attr	**Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)**
setattr(fh, attr) -> attr	**Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.**
read(fh, offset, count) -> attr, data	**Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.**
write(fh, offset, count, data) -> attr	**Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.**

Operações *read* e *write* são idempotentes. Como?
Através da adição da posição a ler/escrever

...

Servidores *stateful*

- Nesta aproximação, o servidor mantém informação sobre quais são os clientes e sobre o seu estado (e.g. ficheiros abertos, posição de leitura/ escrita, etc.)
  - Exemplo: protocolo CIFS/SMB

Servidores *stateless* vs. Servidores *stateful*

- Propriedades stateless
  - Mascara mais facilmente as falhas dos servidores e de comunicações. Porquê?
  - Não há limite teórico para o número de utilizadores simultâneos. Porquê?
  - Não existe necessidade de propagar operações OPEN / CLOSE. Porquê?
- Propriedades stateful
  - Operações podem ser mais simples e é mais simples ter semântica equivalente à dos sistemas centralizados. Porquê?
  - Permitem implementar *locks* mais facilmente. Porquê?
  - Tendem a permitir uma política de *caching* também mais eficaz (como veremos a seguir)

Operações idempotentes

- É possível desenhar um interface com (quase todas as) operações idempotentes – e.g. NFS
  - Para algumas operações é fácil: ex.: read, write, etc. Como?
  - Para outras operações não é possível (ex.: rename, mkdir, rmdir). Porquê?
  - Para estas operações, é comum o servidor manter uma pequena *cache* das últimas operações deste tipo, implementando uma semântica "at-most-once" simplificada.

- A utilização de operações idempotentes simplifica o protocolo e a implementação do servidor. Porquê?
  - Estado mantido pelo servidor é nulo (ou mínimo)
  - Protocolo RPC pode implementar semântica "at-least-once". Porque é que é mais simples?

- Se tudo estiver a funcionar bem, um *crash* do servidor seguido da sua recuperação, uma interrupção momentânea da rede, ou a perca de alguns pacotes, são falhas que são mascaradas facilmente pelo conjunto do sistema
  - Como?

Falhas: não tão simples como parece...

- *Cache* no (SO do) servidor levanta alguns problemas na presença de falhas. Quais?

- Soluções possíveis?
  - Exemplo NFS:
  - Versão 2: o *write* apenas retornava após a escrita ser efectuada em disco (cache *write-through*). Porquê? Quais os problemas?
  - Versão 3: o *write* escreve na *cache do file-system* (como qualquer *write* local). A operação *commit* permite garantir que as escritas efectuadas foram escritas em disco (normalmente executado aquando do *close*).

Modelos de *Caching* : informação replicada

- Quase todos os sistemas de ficheiros distribuídos mantém um cache que pode ter diferentes características
- Modelo *caching* por blocos. Guarda blocos de ficheiros (geralmentes da dimensão do bloco dos discos e com dimensão idêntica à usada na transferência entre o cliente e o servidor).
- Modelo *Caching* de Ficheiros "inteiros". Guarda ficheiros completos na cache – sempre que é necessário aceder a um ficheiro, este é transferido para a máquina do cliente. Nota: caching efectuado normalmente em disco.
- Modelo Serviço Remoto ou sem Cache. Cada vez que é necessário aceder a um ficheiro, o cliente invoca o servidor.

Eficiência?

Depende da fracção de operações que podem ser servidas pelos valores guardados na cache.
Caching fich. inteiros > caching blocos > sem caching

Modelos de *Caching* : informação replicada

- Quase todos os sistemas de ficheiros distribuídos mantém um cache que pode ter diferentes características
- Modelo caching por blocos. Guarda blocos de ficheiros (geralmentes da dimensão do bloco dos discos e com dimensão idêntica à usada na transferência entre o cliente e o servidor).
- Modelo Caching de Ficheiros "inteiros". Guarda ficheiros completos na cache – sempre que é necessário aceder a um ficheiro, este é transferido para a máquina do cliente. Nota: caching efectuado normalmente em disco.
- Modelo Serviço Remoto ou sem Cache. Cada vez que é necessário aceder a um ficheiro, o cliente invoca o servidor.

Coerência?

Depende do modo como é verificada a validade da cache
Sem caching > caching blocos e caching fich. inteiros

Acessos concorrentes em sistemas tradicionais

- Nos sistemas de ficheiros centralizados, a semântica dos acessos concorrentes é definida por:
  - política por omissão
  - mecanismo de *locks*
- No sistema Unix (e Linux …), a política por omissão é do tipo *serialização* das operações sobre cada ficheiro: uma leitura lê sempre o resultado da última escrita.
- Existe ainda um sistema de locks que permite garantir acessos sem conflitos (write / write ou read / write) por dois processos distintos.
  - Exemplo de *system call* para manipular locks no sistema Unix:
    result = flock(filedes, offset, range, locktype)
- Nos sistemas centralizados estas políticas são relativamente fáceis de implementar pois só existe um sistema de gestão de ficheiros e uma cache

Acessos concorrentes distribuídos

- Quando o sistema de gestão de ficheiros é distribuído o problema do acesso partilhado e do controlo da concorrência é mais difícil de resolver. Porquê?
- Os métodos usados para lidar com este aspecto podem agrupar-se em "métodos pessimistas" ou "métodos optimistas".
- Os métodos pessimistas tentam esconder que existem várias cópias e procuram emular o melhor possível a semântica do sistema centralizado – a distribuição é mais transparente.
  - Estes sistemas poderiam descrever-se como "sistemas com equivalência a uma só cópia" ("one copy serializability").
- Os métodos optimistas tentam não esconder totalmente a replicação e o "caching" em troca de um maior desempenho, adaptação à escala e flexibilidade.
  - No limite são os únicos aplicáveis em sistemas móveis quando se pretende admitir acesso em escrita com desconexão.

Controlo de concorrência pessimista com servidores *stateless*

- A semântica tradicional centralizada consiste em uma leitura ver sempre o resultado da última escrita (serialização dos acessos).
- A semântica dum esquema de gestão das caches depende do que se faz nas operações de leitura e escrita:
  - Associado às escritas está a propagação das modificações para o servidor (e sua visualização por outros clientes).
  - Associada às leituras está a verificação da coerência da cache em relação ao servidor.

Modelo da cache do NFS

- Os servidores Unix e os respectivos clientes usam sempre *caching* de ficheiros locais na cache, em memória central. A política convencional é satisfazer os pedidos de leitura da cache se possível e usar *read-ahead* e *delayed-write* (30 segundos no máximo) dos ficheiros locais.
- O módulo cliente do NFS coloca também na cache local os blocos que lê do servidor e utiliza um sistema de *timestamps* para controlar a respectiva validade. Há duas *timestamps* associadas a cada bloco na cache:
  - Tc - última hora a que esta entrada na cache foi validada (hora local)
  - Tw-client - estampilha horária da última vez que o ficheiro foi escrito (hora do servidor tal como está registada actualmente no cliente)
- No momento T, considera-se válida uma entrada na cache sse: T-Tc< t Caso contrário, é necessário verificar se o ficheiro não foi alterado no servidor, i.e., se Tw-server = Tw-client
  - Se t é grande pode resultar incoerência, se t é pequeno pode resultar ineficiência.

Porque é que (quase) todas as operações NFS retornam atributos do ficheiro?

Modelo da cache do NFS (2)

- O valor de t é calculado de forma dinâmica e varia entre 3 e 30 segundos para ficheiros (o valor é menor quanto mais frequentemente o ficheiro é escrito) e entre 30 e 60 segundos para directórios (estratégia idêntica).
- Se T - Tc < t, o segundo teste não é realizado, caso contrário é executado um *getattr request* ao servidor e os valores de Tc e Tw são actualizados.
  - Se Tw-server ≠ Tw-client todos os blocos do ficheiro são retirados da cache pois o ficheiro foi modificado.
  - Para optimizar, os valores de Tw são aplicados a todos os blocos do mesmo ficheiro e os atributos do ficheiro são piggybacked sempre que possível com todas as respostas do servidor (ver a definição das operações NFS)
  - Qual o problema de manter apenas um Tw para cada ficheiro?
- Os blocos modificados por uma escrita no cliente são marcados como *dirty* e enviados assincronamente para o servidor ou logo que um *sync* ocorre no cliente.

Como se comporta esta aproximação com um sistema de bases de dados?

Gestão da cache com servidores *stateless*

- Leituras. Para garantir a frescura da informação *cached*, os clientes de servidores stateless têm de testar a validade da mesma:
  - teste antes de cada acesso - aumenta a coerência mas diminui a performance
  - teste periódico - semântica sem garantias totais
- Escrita. A propagação de escritas pode ser efectuada em vários momentos.
  - Escrita "Write-through". A cada escrita no cliente corresponde uma (ou mais) escritas no servidor. Propriedades?
  - Favorece a fiabilidade e a semelhança com a semântica tradicional tipo "sistema equivalente a uma só cópia" à custa do sacrifício do desempenho.
  - Escrita "Delayed-Write". As escritas são inicialmente efectuadas na cache local e, posteriormente, enviadas para o servidor. Propriedades?
  - Estes esquemas diminuem o tráfego de rede mas diminuem a fiabilidade e consistência.
    Quando o servidor é *stateless* e ignora os clientes, não é possível implementar a semântica "equivalência a uma só cópia".

NFS: Gestão da cache com servidores *stateless*

- NFS
  - Escrita:
  - "delayed-write", read-ahead
  - Leitura:
  - testa periodicamente

- Para que tipo de ambiente é apropriado?

- Acesso aos ficheiros num ambiente de desenvolvimento de *software* ou de partilha não simultânea dos ficheiros. O resultado é um sistema com uma semântica de carácter probabilístico, mas com muito sucesso no ambiente alvo e que afinal é mais optimista que pessimista.

Gestão da cache com servidores stateful

- Aproximação controlada pelo servidor: o servidor (stateful) anota cada cliente de um ficheiro e o modo de acesso e gere um sistema de notificações de alterações, assim como mecanismos baseados em *tokens* e *locks* para ajudar os clientes a melhorarem o seu desempenho.

- Quando existe um sistema de *locks* distribuídos, o cliente que possui um *lock* sobre um ficheiro pode usar esse conhecimento para melhorar a gestão da sua cache.
  - Se o cliente possui um *lock* de leitura partilhada, ????
  - Se o cliente possui um *lock* de leitura partilhada, o ficheiro não pode ser escrito por outros e portanto enquanto o *lock* for válido não será modificado – os testes de validade do ficheiro *cached* são inúteis.

  - Se o cliente possui um *lock* de escrita exclusiva, ????
  - Se o cliente possui um *lock* de escrita exclusiva, o ficheiro não pode ser escrito por outros e basta fazer escritas periódicas para minorar problemas em caso de *crash*.

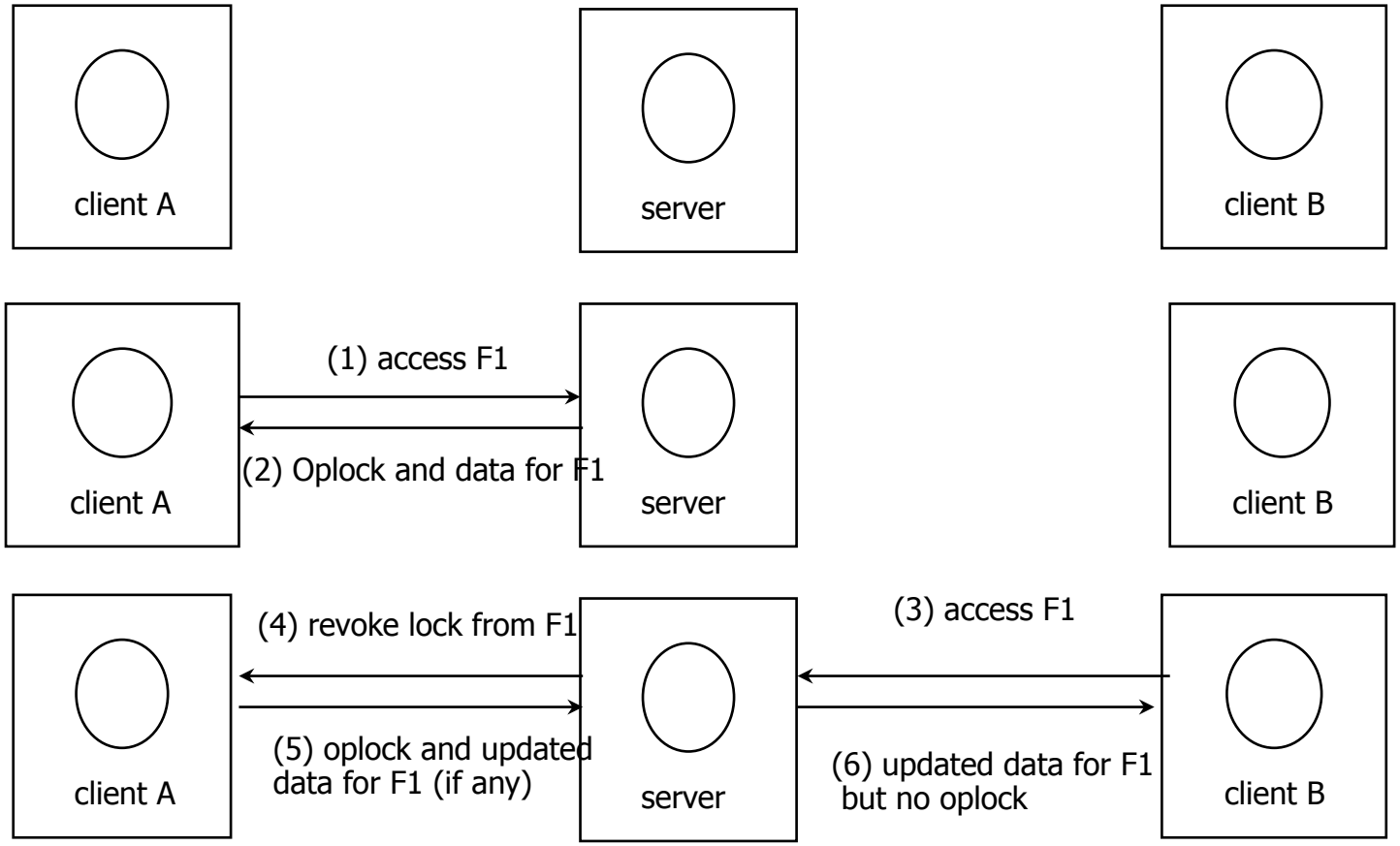
Gestão de cache no sistema CIFS: Opportunistic locks

- Neste esquema, introduzido no sistema CIFS (sucessor do SMB), existem vários tipos de *locks*:
  - *Opportunistic locks (oplocks) exclusivos* que permitem ao cliente ter acesso exclusivo ao ficheiro e fazer *caching* arbitrário do mesmo. Um *oplock* pode ser retirado ao cliente pelo servidor.
  - *Oplocks partilhados* que permitem aos clientes ter acesso ao ficheiro em leitura e fazer *caching* arbitrário do mesmo. Um *oplock* pode ser retirado ao cliente pelo servidor.
  - *Mandatory locks* que permitem acessos exclusivos e *caching* e que não podem ser retirados pelo servidor

- Os clientes fazem ou não *caching* dos ficheiros conforme o tipo de *lock* que têm. Se não têm nenhum podem sempre aceder ao ficheiro mas sem fazer *caching* do mesmo.

Ilustração

Neste exemplo ilustra-se a utilização de *oplocks* para acesso ao ficheiro F1 por dois clientes A e B.



Gestão dos oplocks

1. O primeiro cliente obtém:
   1. Oplock exclusivo caso pretenda escrever o ficheiro
   2. Oplock partilhado caso pretenda apenas ler o ficheiro
2. Quando surgem novos clientes de leitura:
   1. Existe um cliente com oplock exclusivo
      - Cliente perde lock e deve enviar as modifcações efctuadas
      - Ambos os clientes ficam sem oplocks (e portanto sem poder fazer cache)
   2. Existe um cliente com oplock partilhado
      - Ambos os clientes ficam com oplock partilhado
3. Quando surge um novo cliente de escrita
   1. Existe um cliente com oplock exclusivo
      - Cliente perde lock e deve enviar as modifcações efctuadas
      - Ambos os clientes ficam sem oplocks (e portanto sem poder fazer cache)
   2. Existe um (ou mais) clientes com oplock partilhado
      - Clientes perdem o oplock
      - Todos os clientes ficam sem oplocks (e portanto sem poder fazer cache)

Gestão dos oplocks (cont.)

- Quando um ficheiro é aberto para leitura/escrita tenta-se prolongar concorrência, assumindo que o cliente é de leitura até à primeira escrita – nesse momento, efectua-se o procedimento de adição de um novo cliente.
- Se um *oplock* é quebrado, deixa de haver *caching* do ficheiro.

Solução "Callback promise"

- Introduzido no AFS e usado também no Coda.

- Neste sistema os clientes fazem *caching* de ficheiros inteiros, sendo o ficheiro a unidade de transferência entre o cliente e o servidor. Quando um cliente obtém um cópia do ficheiro, o servidor *promete* informar o cliente de qualquer modificação efectuada ao ficheiro – a esta promessa chama-se uma "*callback promise*".

- Desde que o cliente tenha uma "*callback promise*" válida, assume que a sua cópia do ficheiro está actualizada. Neste caso, um ficheiro é aberto no cliente sem nenhuma comunicação com o servidor.

- Quando um programa no cliente fecha um ficheiro que acabou de modificar, o cliente AFS/Coda envia as modificações para o servidor. Nessa altura, o servidor notifica todos os clientes com "callback promises" válidas.

- Um cliente, ao receber a notificação, anula a sua "callback promise" e, se o ficheiro estiver aberto, obtém uma nova cópia do ficheiro.

Funcionamento usando "callback promise"

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
	Open the local file and return the file descriptor to the application.	Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

Solução "Callback promise"

- Com este esquema todos as escritas feitas no cliente são temporárias até ao fecho do ficheiro. Desta forma, a única semântica que pode ser implementada é uma semântica dita "semântica de sessão". Com esta semântica, uma leitura lê o resultado das escritas feitas depois do último *close* do ficheiro.
- Quando um cliente é inicializado, ele tem de revalidar as "*calback promises*" que tem. Porquê?

  Pode ter ter perdido os "calbacks" feitas pelo servidor enquanto o cliente esteve desligado.

Resumo sobre gestão da cache e concorrência nestes sistemas

- Método *write-through*. Semântica equivalente a "uma cópia única com acesso serializado" mas pouco eficaz. É a solução usada quando os *oplocks* são quebrados (em conjunção com a leitura sistemática no servidor).
- Método *delayed-write*. Semântica "aleatória" mas mais eficaz. É a solução usada pelo NFS.
- Métodos com controlo centralizado: permitem a semântica equivalente a "uma cópia única com acesso serializado" mas dependem de o servidor ser *stateful*. Introduzem um ponto central de falha e dependem de métodos de detecção e tratamento das falhas.
- Métodos *"callback promise"* : semântica de sessão – leitura lê o resultado do último close.

Para saber mais

- G. Coulouris, J. Dollimore and T. Kindberg, Distributed Systems - Concepts and Design, Addison-Wesley, 4th Edition, 2005
  - Capítulo 8. Corresponde à matéria aqui apresentada mas não cobre o sistema SMB/ CIFS nem os esquemas de controlo de concorrência baseados em oplocks e tokens.