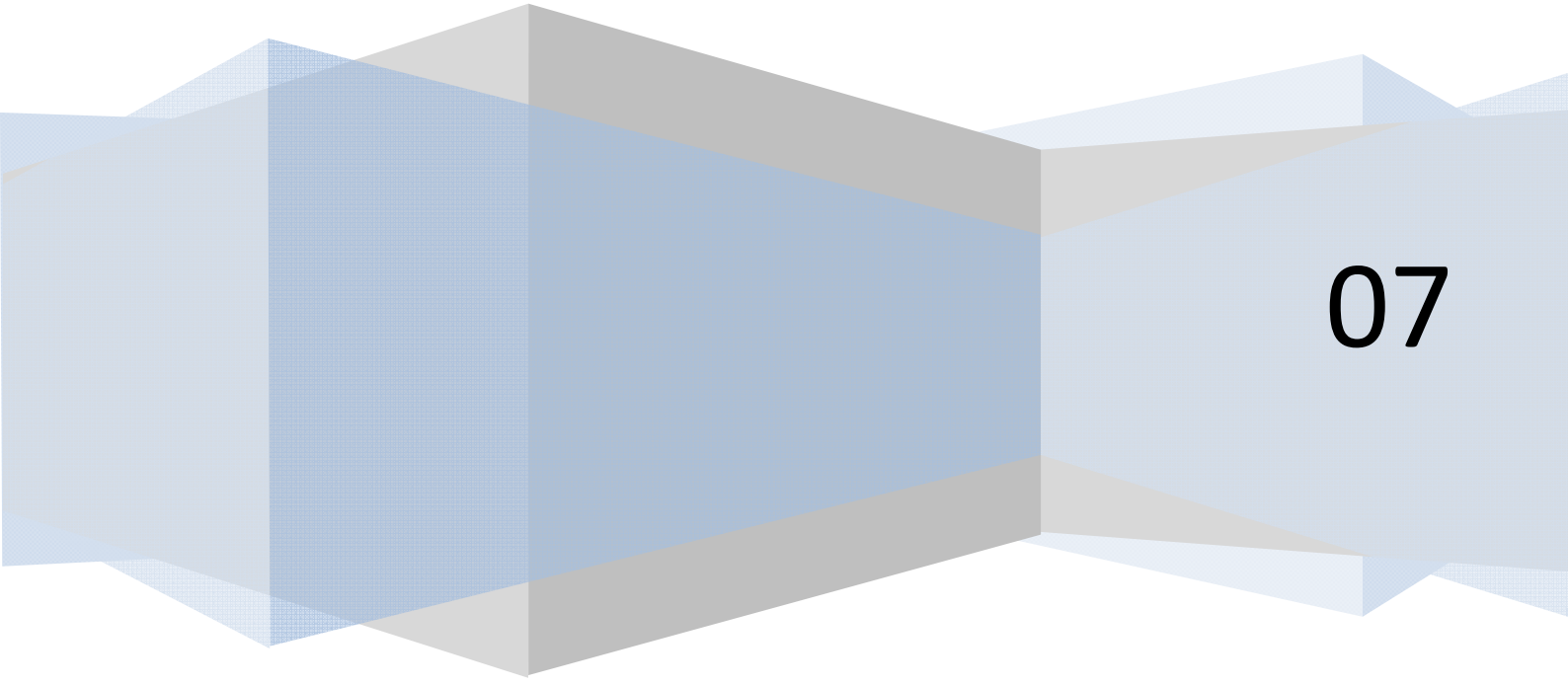
FCT - UNL

Apontamentos Teóricos

SD1 2006/2007

João Pedro



07

I – Introdução

Definição:

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)

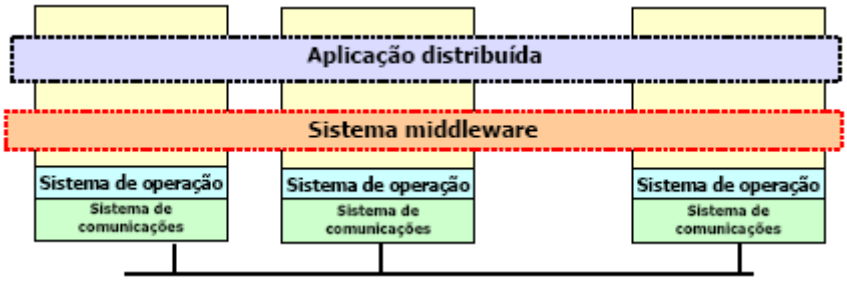
Motivação:

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

Características:

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

Middleware:



Interface homogénea; Serviços mais complexos (invocação remota: RMI, Web-services, Corba; comunicação em grupo: Horus, etc); Verdadeira interoperabilidade requer idênticos *interface* e protocolos.

Segurança:

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

Abertura:

A abertura de um sistema determina o modo como pode ser estendido e re-implementado

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:
  - o recursos e utilizadores
  - o âmbito geográfico (rede local, país, mundo, ...)
  - o âmbito administrativo (uma organização, inter-organizações)

Sist. Escalável:

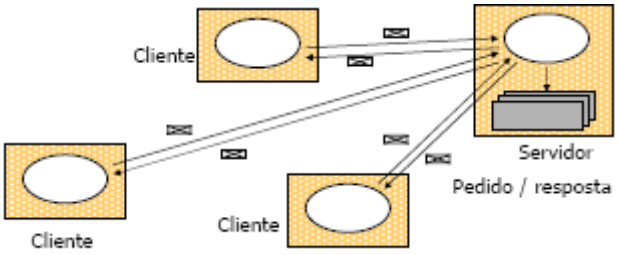
Um sistema capaz de escalar (escalável) é um sistema que continua eficaz quando há um aumento significativo do número de recursos e utilizadores.

Falhas:

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. [Detectar, tolerar e mascarar falhas]

II - Arquitecturas e Modelos de Sistemas Distribuídos

Cliente/Servidor:



Sistema em que os processos podem ser divididos em dois tipos, de acordo com o seu modo de operação:

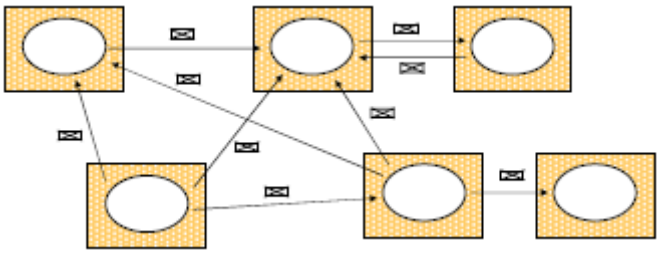
Cliente: programa que solicita pedidos a um processo servidor

Servidor: programa que executa operações solicitadas pelos clientes, enviando o respectivo resultado

Propriedades:

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

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

Propriedades:

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

Variantes:

Cliente/servidor replicado

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

Cliente/servidor particionado

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

Cliente leve (*thin client*)/servidor

- ✓ O cliente apenas inclui um interface (gráfico) para executar operações no servidor (ex: browser)
- ✓ Vantagens:
  - o Cliente pode ser muito simples
- ✓ Desvantagens
  - o Maior peso no servidor

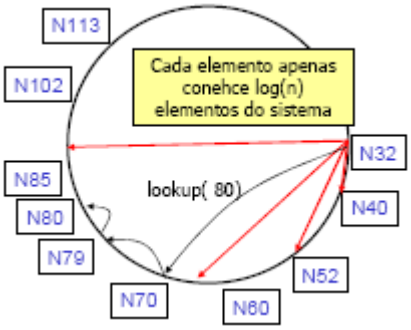
Cliente completo(estendido)/servidor

- ✓ O cliente executa localmente algumas operações que seriam executadas pelo servidor
- ✓ Vantagens:
  - o Permite funcionar quando não é possível contactar o servidor (recorrendo a caching)
  - o Permite diminuir a carga do servidor e melhorar o desempenho
- ✓ Desvantagens:
  - o Implementação do cliente mais complexa
  - o Necessário tratar da coerência dos dados entre o cliente e o servidor

Peer-to-Peer não estruturado:

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

Peer-to-Peer estruturado:



- ✓ Os membros do sistema comunicam de acordo com uma organização definida de forma determinista
  - o E.g. DHT permitem pesquisar valores conhecidos
    - ▪ Nó responsável por uma dada chave depende do identificador do nó
- ✓ Vantagens
  - o Boa latência/escalabilidade - O(log n)
- ✓ Desvantagens
  - o Maior complexidade

Combinação dos modelos com servidor 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 p2p

✓ Propriedades:
- Permite a um host aceder a um serviço disponibilizado por um sistema p2p
- Permite limitar o número de processos que fazem parte do sistema 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, Edonkey

✓ Propriedades:
- Permite combinar as vantagens de ambos os sistemas

Sistemas peer-to-peer hierárquicos

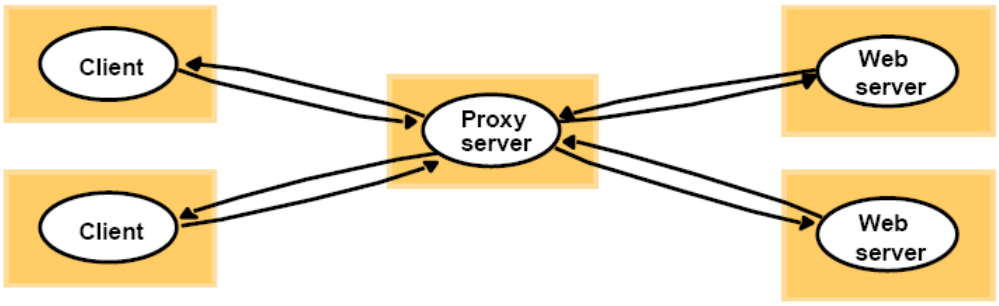
✓ Subconjunto de super-nós que se agrupam como num sistema p2p
✓ Nós ligam-se a um super-nó

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.

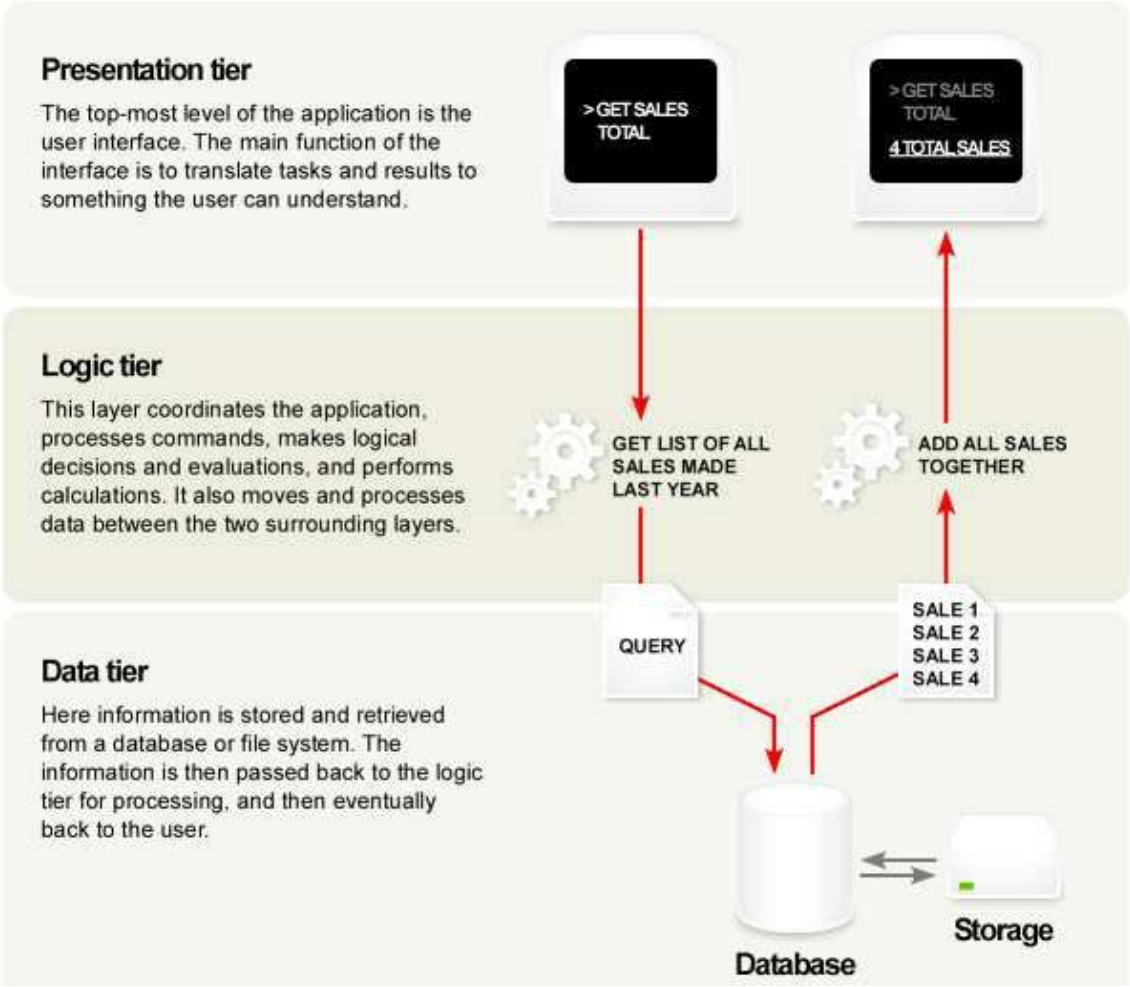
Proxy de um serviço



✓ 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

Arquitectura three-tier



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

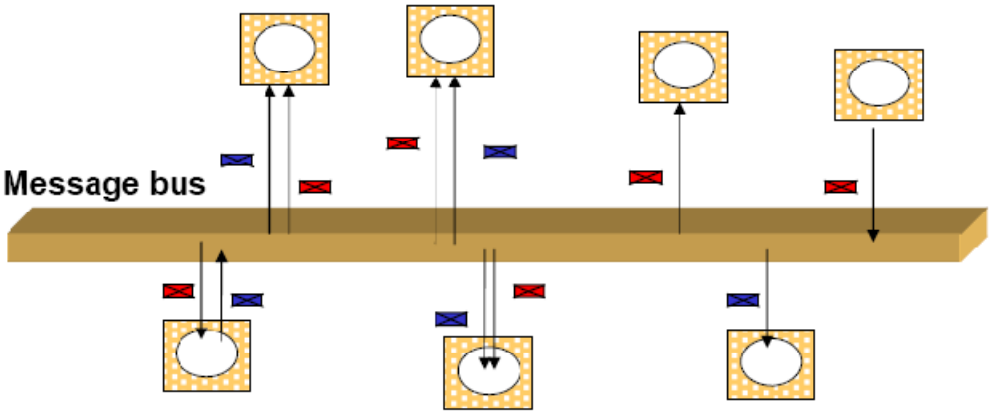
Arq. orientadas a serviços – service-oriented architectures (SOA)

- ✓ Arquitectura em que os recursos são disponibilizados como serviços independentes que podem ser acedidos conhecendo apenas a interface e protocolo de acesso
  - o Existe mecanismo de descoberta de serviços
  - o Serviços tendem a ser stateless
  - o Pode-se implementar usando qualquer tecnologia: RPCs, RMI, Jini, etc.
    - ▪ Surge normalmente associado com Web Services (permitem a invocação remota usando protocolo sobre HTTP)
- ✓ Propriedades desejáveis:
  - o Reutilização, modularidade, possível compor serviços mais complexos, descrição dos serviços
- ✓ Desafios:
  - o Segurança, interoperablidade, estado nos serviços

Modelo de interacção

- ✓ Tipo de interacção
  - o Activa (Processo solicita execução de operação noutro processo)
  - o Reactiva (Evento no sistema desencadeia acção num processo)
  - o Indirecta (Processos comunicam através de um espaço partilhado)
- ✓ Conteúdo da interacção
  - o Informação/dados (1)
  - o Código (2)

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

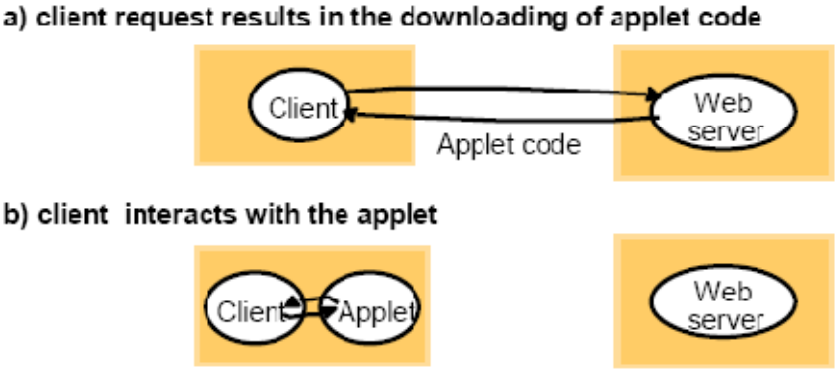
Modelo de interacção indirecta (ex.: memória partilhada distribuída)

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

Conteúdo da interacção (Dados)

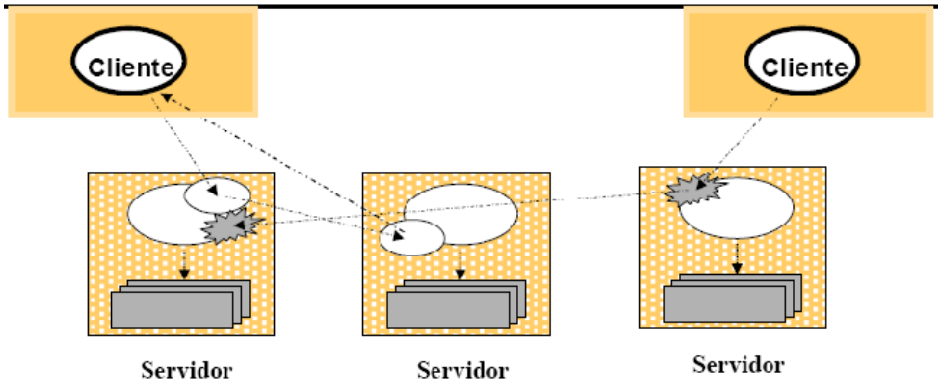
- ✓ Processos trocam dados
  - o Pedido (operação a invocar + parâmetros + inf. utilizador + ...)
  - o Resposta (resultado de operação + ...)
  - o Evento (num sistema de eventos)
- ✓ Propriedades
  - o Parceiros devem conhecer formato das mensagens
  - o Parceiros devem saber processar mensagens (operações)

Conteúdo da interacção (código móvel – cliente/servidor)



- ✓ A execução do código no cliente pode:
  - o Melhorar o desempenho
  - o Ser usado para implementar funcionalidade adicional
- ✓ Segurança
  - o 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
- ✓ Problemas
  - o Proteger informação do agente do ambiente de execução (impossível?)
  - o 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
  - o 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.

Falhas

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.
- ✓ Availability – disponibilidade. Mede a fracção de tempo em que um serviço está a operar correctamente, isto é, de acordo com a sua especificação.
  - o 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.
  - o Um sistema que falha com grande frequência e recupere rapidamente tem baixa fiabilidade, mas alta disponibilidade.
- ✓ 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 Falhas:

- ✓ Uma falha por omissão dá-se quando um processo ou um canal de comunicação falha a execução de uma acção que devia executar
  - o 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
  - o Exemplo: chegou uma mensagem corrompida, um atacante produziu uma mensagem não esperada.
  - o Para lidar com estas falhas é necessário garantir que elas não levam a que outros componentes passem a estados incorrectos

Mascarar tipos de falhas:

- ✓ 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
  - o Ex.: se uma mensagem não chegou dentro de um certo período (que se detecta por timeout), pode-se emiti-la novamente ou duplicar o canal, enviar mais que uma cópia em paralelo e filtrar as mensagens duplicadas.
- ✓ As falhas arbitrárias podem ser difíceis de mascarar. Pode-se tentar transformálas em falhas por omissão
  - o Exemplo: um CRC numa mensagem permite transformar uma falha bizantina do canal numa falha por omissão

Tipos de erro/falha: duração

- ✓ Permanentes: mantêm-se enquanto não forem reparadas (ex: computador avaria)
  - o Mais fáceis de detectar
  - o Mais difíceis de reparar
- ✓ Temporárias: ocorrem durante um intervalo de tempo limitado, geralmente por influência externa
  - o Mais difíceis de reproduzir, detectar
  - o Mais fáceis de reparar
  - o Erros transientes: ocorrem instantaneamente, ficam reparados imediatamente após terem ocorrido (ex.: perda de mensagem)

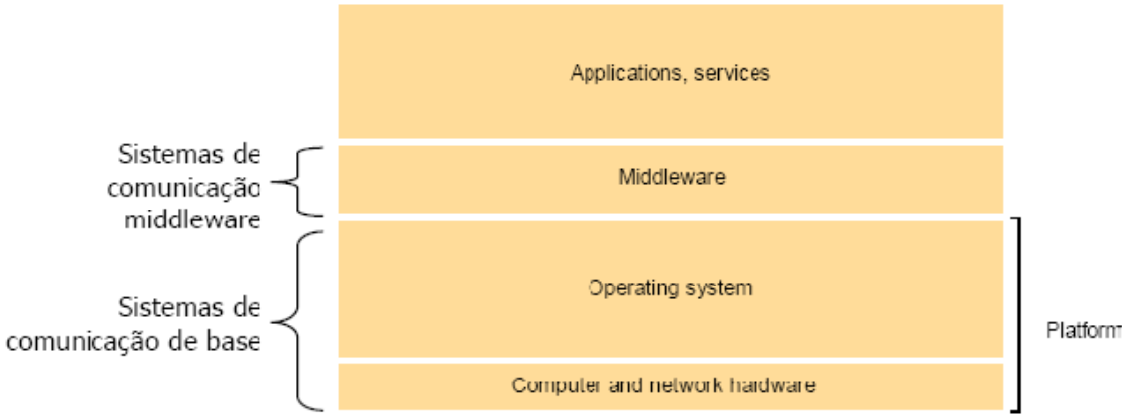
Segurança num sistema distribuído (modelo de segurança)

- ✓ 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:
  - o Protegendo os processos e os canais usados para a interacção entre processos;
  - o Protegendo os objectos (dados) geridos pelos processos contra acessos não autorizados.

Canais Seguros

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

III - Comunicação em Sistemas Distribuídos



Sistemas de comunicação de base

- ✓ Os sistemas de operação suportam a comunicação de dados entre os diferentes computadores envolvidos num sistema distribuído
  - o Interfaces geralmente de baixo nível
  - o Os protocolos mais populares são os protocolos TCP/IP que oferecem os seguintes modos de comunicação de base:
    - ▪ Streams (TCP) – comunicação por fluxo contínuo de dados. Dados transmitidos como fluxo contínuo. Dados chegam de forma fiável a menos que o stream seja quebrado.
    - ▪ Datagramas (UDP) – comunicação por mensagens. Mensagens podem-se perder, duplicar e chegar fora de ordem.
- ✓ Funcionalidades adicionais (modelos de sincronização, suporte para tratamento de falhas, da segurança, da heterogeneidade e as ajudas à programação) devem ser implementadas pelas aplicações ou por um nível intermédio (middleware)

Comunicação no nível middleware

- ✓ Comunicação por streams (fluxos).
  - o Em geral é entre dois pontos e bidireccional (full duplex).
  - o Também pode ser uni-direccional com uma origem e um ou vários destinatários.
- ✓ Comunicação por mensagens. Pode assumir duas formas:
  - o Comunicação ponto a ponto
  - o Comunicação multiponto ou comunicação em grupo (1 para N por exemplo).
- ✓ Comunicação baseada no paradigma pedido / resposta
  - o Invocação de métodos/procedimentos remotos (RMI / RPC)
- ✓ Comunicação baseada no paradigma do código móvel (“agentes”)

Streams

- ✓ Uma mensagem é recebida depois de emitida e a ordem das mensagens num stream é (geralmente) mantida, isto é, se P emite m1, m2, … então Q recebe m1, m2, ...
  - o A dimensão (fronteira) das mensagens deve ser definida pela aplicação
- ✓ Algumas variantes
  - o Um stream pode ser composto por vários substreams (vídeo+audio)
    - ▪ Problemas de sincronização dos substreams
  - o Qualidade de serviço
    - ▪ Condições temporais para propagação entre extremos (síncrono)
    - ▪ Reserva de largura de banda (ex: RSVP)

- ✓ Concorrência: permite concorrência porque apenas bloqueia o emissor se existirem problemas com buffers. Trata-se, no essencial, de uma relação assíncrona do emissor com o receptor do tipo produtor / consumidor
- ✓ Sincronização: o receptor sabe que os dados foram produzidos antes de serem consumidos e o emissor sabe que os dados só podem ser consumidos no futuro.
- ✓ Ordenação: geralmente garante a ordem das mensagens do mesmo emissor.
- ✓ Modelo de falhas: o emissor não sabe quando é que o receptor vai receber os dados. Em caso de falha o emissor não sabe que dados o receptor recebeu exactamente. O processo que usa o canal não sabe se a falha é no canal ou no outro processo.

Sincronização entre o emissor e o receptor: comunicação por mensagens

Comunicação assíncrona:

- ✓ o emissor só fica bloqueado até o seu pedido de emissão ser tomado em consideração
- ✓ o receptor fica bloqueado até ser possível consumir dados
  - o Em geral, o sistema de comunicação do receptor armazena (algumas) mensagens caso não exista nenhum receptor bloqueado no momento da sua recepção
  - o É possível variante em que o receptor não fica bloqueado e devolve erro ou recepção é efectuada em background

- ✓ Concorrência: permite o máximo de concorrência entre o emissor e o receptor. O sistema de comunicações funciona como um “buffer”.
- ✓ Sincronização: o receptor sabe que a mensagem foi emitida antes de ser recebida e o emissor sabe que ela vai ser recebida no futuro.
- ✓ Ordenação: quase sempre não garante a ordem das mensagens do mesmo emissor nem de emissores distintos mas é fácil de executar a ordenação das mensagens do mesmo emissor.
- ✓ Modelo de falhas: geralmente o emissor não sabe quando é que o receptor vai receber a mensagem ou mesmo se este a recebeu.
  - o Fiabilidade da transmissão de mensagens depende do sistema/garantias.

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

Comunicação síncrona unidireccional:

- ✓ Concorrência: limita a concorrência do emissor pois bloqueia-o à espera do receptor.
- ✓ Sincronização: o receptor sabe que a mensagem foi emitida antes de ser recebida e o emissor sabe que ela acabou de ser recebida de facto. O receptor sabe também que o emissor esperou por ele.
- ✓ Ordenação: 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 síncrona pedido/resposta

- ✓ Concorrência: limita a concorrência do emissor pois bloqueia-o 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?

Comunicação e persistência

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

Comunicação persistente síncrona

- ✓ Concorrência: Emissor bloqueia-se até existir garantia que o sistema de comunicação armazenou persistentemente a mensagem. Receptor pode bloquear-se na recepção de mensagem.
- ✓ Sincronização: O receptor sabe que a mensagem foi emitida antes de ser recebida. Nenhum sincronização entre emissor e recepto adicional.
- ✓ Ordenação: geralmente garante a ordem das mensagens do mesmo emissor.
- ✓ Modelo de falhas: Se emissão tiver sucesso, emissor tem garantia que mensagem foi armazenada. Se falhar, emissor não sabe o que aconteceu. Se mensagem 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)

Comunicação persistente assíncrona

- ✓ Concorrência: Emissor nunca se bloqueia. Receptor pode bloquear-se na recepção de mensagens.
- ✓ Sincronização: o receptor sabe que a mensagem foi emitida antes de ser recebida. Nenhuma sincronização adicional.
- ✓ 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)

Tipos comunicação

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

- ✓ Comunicação multicast ou em grupo [1xN] – envio de uma mensagem de um emissor para um grupo de receptores
  - o Pode exigir que o emissor pertença ao grupo ou não
  - o 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

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

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
  - o 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
  - o Membros que falham podem ter entregue a mensagem ou não
  - o 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
  - o 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
  - o NOTA: a ordem de entrega das mensagens de um mesmo emissor pode ser diferente da ordem de emissão
  - o Um sistema de multicast fiável no qual as mensagens são entregues por ordem total chama-se sistema de multicast atómico
- ✓ Ordem causal – se m1 pode causar m2, m1 deve ser entregue sempre antes de m2
  - o Se duas mensagens forem emitidas pelo mesmo processo, considera-se que o envio de uma mensagem pode causar o envio da segunda (Neste caso, a recepção da segunda deve acontecer antes da recepção da primeira)
  - o 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)

Caracterização do multicast: vistas

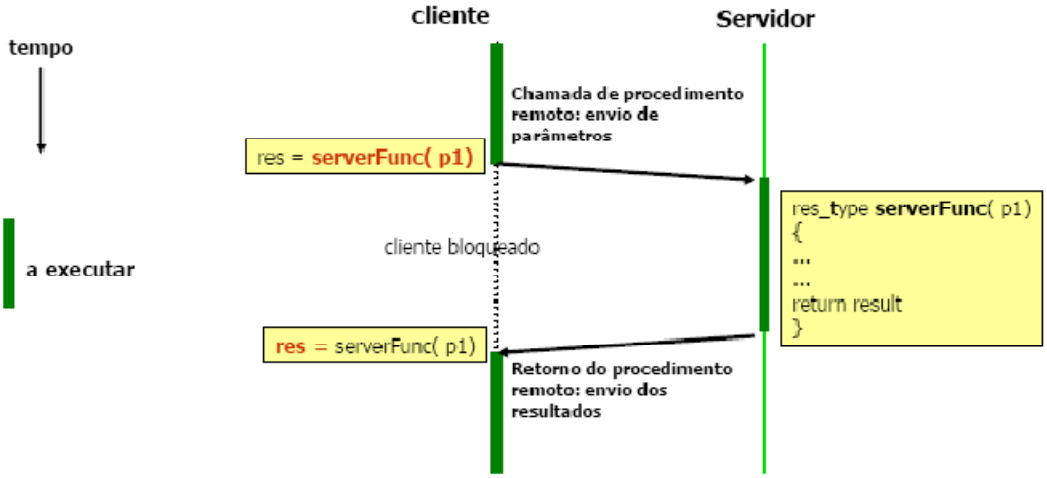
- ✓ Uma vista (view) é o conjunto de elementos de um grupo num dado momento.
  - o Nos sistemas de comunicação em grupo fiáveis, a mudança de vista é efectuada através do envio de uma mensagem multicast
- ✓ Sincronia virtual (virtual synchrony) – um sistema de multicast fiável implementa sincronia virtual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

IV - Invocação de procedimentos e de métodos remotos

Invocação de procedimentos remotos

- ✓ Uma extensão natural a um ambiente distribuído consiste em permitir a execução de procedimentos noutra máquina
- ✓ Se os procedimentos estiverem definidos no âmbito de um objecto, chama-se invocação de métodos remotos (Java RMI, CORBA, .NET)
- ✓ Web services: permitem efectuar invocação remota de procedimentos usando mensagens (geralmente) representadas em XML e enviadas usando HTTP
    - o Web services permitem interacções mais complexas que simples pedido/resposta

Invocação de procedimentos remotos (RPCs)



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

Propriedades

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

Implementação

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

Nota: *Stub* ou *skeleton* do servidor tem 3. e 2. Enquanto que o *Stub* do cliente ou *proxy* do servidor tem 4. e 1.

Automatização do processo - Stub ou proxy compilers

Nos sistemas de RPC/RMI existe geralmente 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 (ex: *Java*)
- ✓ Algumas linguagens/sistemas incluem suporte para invocação remota, pelo que as comunicações são tratadas pelo runtime de suporte (ex: *.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 transmissão 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; Cabeçalho (assinatura) das funções / procedimentos

Os IDLs são usados apenas para definir os interfaces, não o código das operações

IDLs – aproximações possíveis

1. Usar sub-conjunto de uma linguagem já existente (Ex.: *Java RMI*)
2. Definir linguagem específica para especificar interfaces dos servidores / objectos remotos (Ex.: *DCE RPCs*)
   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

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*)

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

Aproximação comum nos sistemas de RPC/RMI:

- ✓ Passagem por valor para tipos básicos, arrays, estruturas e objectos não remotos
  - o Apontadores/referências para arrays, objectos, etc. são seguidas
  - o Estado dos objectos é copiado (ex: Java RMI)
- ✓ Passagem por referência para objectos remotos
  - o Quando o tipo de um parâmetro é um objecto remoto, uma referência para o objecto é transferida

Aproximações à codificação dos dados

- ✓ Utilização de formato intermédio independente (network standard representation)
  - o Emissor converte da representação nativa para a representação da rede
  - o O receptor converte da representação da rede para a representação standard
- ✓ Utilização do formato do emissor (receiver makes it right)
  - o Emissor envia usando a sua representação interna e indicando qual ela é
  - o Receptor, ao receber, faz a conversão para a sua representação
- ✓ Utilização do formato do receptor (sender makes it right)
- ✓ Propriedades:
  - o Desempenho: rep. intermédia tem pior desempenho - exige duas transformações
  - o Complexidade (número de transformações a definir): rep. intermédia exige apenas que em cada plataforma se saiba converter de/para formato intermédio

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 porque poderia abrir problemas de segurança, por exemplo, permitia acesso a campos private)
- ✓ Os campos static e transient não são serializados
- ✓ Usa *reflection* – permite obter informação sobre os tipos em runtime
    - o Assim, não necessita de funções especiais de marshalling e unmarshalling

Representações dos dados: classificação

- ✓ Conteúdo da representação
    - o Formato binário – Corba, Java
    - o Formato de texto – XML
- ✓ Integração com linguagem
    - o Independente – Corba, XML
    - o Integrado – Java
- ✓ Informação de tipos
    - o Incluída – Java, XML
    - o 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:
    - o Passando como parâmetro/resultado uma referência remota – neste caso, uma cópia da referência remota é enviada
    - o 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)

Ligação do cliente ao servidor (binding)

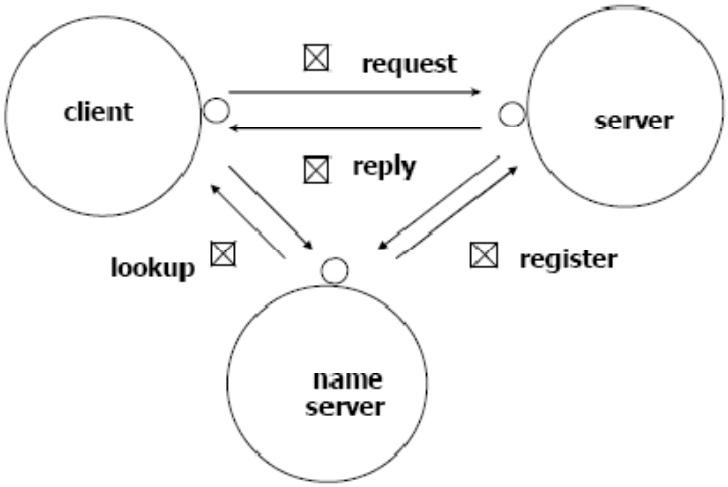
Para poder invocar o servidor, o cliente tem de obter uma referência para o servidor

- ✓ 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)

Obter referência para o servidor

- ✓ Servidor de nomes regista associação entre nome e referência remota
- ✓ Servidor de nomes e directório regista informação sobre servidores
- ✓ Cliente procura servidor usando multicast/broadcast

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.

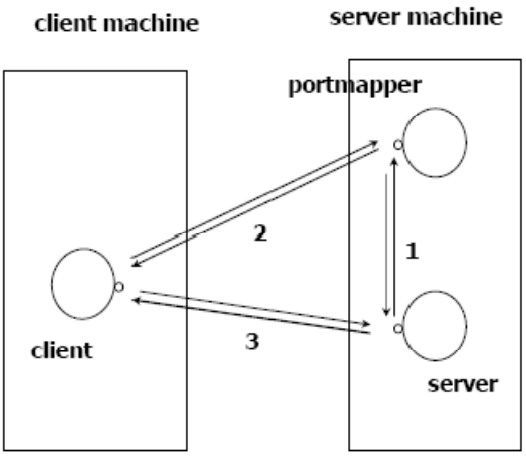
Remote lookup(String name)

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.

Solução RMI e Problemas



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

Protocolo connection-less vs. connection-oriented

- ✓ Motivações para o uso do UDP:
  - o Estabelecer uma conexão tem um peso que se pode revelar demasiado pesado (para pedidos pontuais e de pequenas dimensões)
  - o 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):
  - o Dimensão dos parâmetros/resultados não influencia complexidade da implementação
  - o No caso de usar UDP pode ser necessário enviar um pedido/resposta em mais do que uma mensagem

Filtragem de duplicados (por re-emissão)

- ✓ 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.
  - o Para tolerar *crashs* e recuperações do servidor, a cache com resultados deve ser mantida em memória estável
- ✓ O servidor pode remover a informaçãosobre resultados de pedidos antigos quando souber que já não vai se necessária:
  - o Cliente pode informar o servidor que recebeu o resultado (ack)
  - o 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)

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

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

Protocolos de invocação remota (resumo)

- ✓ Com transporte connection-oriented (TCP)
  - o Request / Reply protocol (RR): Semântica: Implementa facilmente a política no máximo uma vez.
- ✓ Com transporte connectionless (UDP)
  - o Request protocol (R): Semântica: *maybe*
  - o Request / Reply protocol (RR): Semântica: Implementa facilmente semântica pelo menos uma vez.
  - o Request / Reply / Acknowledge protocol (RRA): Semântica: usado para melhorar RR no caso de de se estar a implementar a semântica no máximo uma vez.

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.

Falhas

Independentemente do protocolo usado, para mascarar os crashs dos servidores e dos clientes e implementar uma semântica "exactly-once", é necessário registar em memória estável o estado do cliente e do servidor. Porque:

1. Quando o cliente recupera de uma falha, deve retomar a execução da invocação
2. 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

- ✓ Para se mascarar completamente as falhas dos servidores, clientes e canais e garantir um elevado nível de disponibilidade é necessário utilizar alguma forma de replicação dos mesmos, o que é mais complexo

Utilização de diferentes protocolos

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

Protocolos connection-less

Usados quando se pretende optimizar as comunicações num ambiente com pouco erros nas comunicações e mensagens de dimensão reduzida

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.
  - o Números únicos de serviço / interfaces, números únicos de objectos
  - o Números únicos de invocações
  - o Números únicos de incarnação dos clientes e servidores se necessário
- ✓ Ideia para geração de números únicos
  - o Concatenar identificador único num dado contexto com um contador
  - o Números únicos numa máquina (Exemplo de prefixo único: pid + tempo)
  - o 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

- ✓ Servidor iterativo: o servidor serializa os pedidos dos diferentes clientes, executando um de cada vez (Modelo simples e claro)
- ✓ Para alguns tipos de serviços, esta aproximação pode ter um desempenho inadequado
  - o Exemplos: servidores de bases de dados, de ficheiros, etc. Porquê?
  - o Exemplo: serviços que chamam outros serviços em ambos os sentidos (A→B e B →A).
- ✓ Em geral, quando a execução de uma operação remota pode ser longa é interessante introduzir concorrência no servidor.

Utilização de threads num servidor

- ✓ 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

Nos modelos thread-per-request e thread-perconnection 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

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
  - o Em cada momento, o sistema mantém informação sobre os threads que não estão a processar nenhuma operação, os quais se encontram a dormir
  - o 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
  - o No fim de processar o pedido, o thread volta à pool

Tipo de invocação em RMI

- ✓ Invocação estática – operação a invocar definida em tempo de compilação
  - o Ex: server.someOp(param1, param2)
- ✓ Invocação dinâmica – operação a invocar pode ser definida em tempo de execução
  - o 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.
- ✓ RPCs/RMIs one-way – neste caso, o cliente prossegue imediatamente a seguir a enviar o pedido, não esperando pelo ACK

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
- ✓ Activator: servidor responsável por:
  - o Manter informação sobre os objectos activáveis
  - o Activar os objectos remotos quando solicitado por um cliente
  - o Manter informação sobre localização dos objectos activados
- ✓ Objecto remoto passivo (quando não activado)
  - o Código
  - o Estado do objecto marshalled
- ✓ Referência remota mantém informação necessária para solicitar a activação do objecto

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 por referência ou valor, 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 fazerse remotamente através de URLs.

- ✓ 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 referenciados via interfaces que herdam da interface Remote
- ✓ Todos os objectos serializáveis podem ser passados em parâmetro
- ✓ Os objectos stub implementam a mesma interface que o objecto remoto
  - o pode fazer-se casting
  - o pode usar-se instanceof para testar o tipo das interfaces
- ✓ Os problemas de comunicação e outros são comunicados através de excepções
- ✓ Os argumentos e o valor de retorno podem ser de qualquer tipo desde que seja derivado de Serializable (or RemoteObject, como o UnicastRemoteObject)
- ✓ Os objectos do tipo Serializable são copiados através da serialização de objectos
  - o os stubs (referências remotas) são serializáveis
  - o 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 o stub tem embebida a referência remota (a serialização é especial)
  - o dá o efeito da passagem por referência
- ✓ O código das classes é carregado a pedido

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

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 garbagecollection distribuído para remover (apagar) objectos remotos para os quais não existem referências
    - o Cada máquina virtual (VM) contabiliza as referências que existem para cada objecto remoto e informa a VM do objecto
    - o Quando aparece a primeira referência, a VM do objecto remoto é informada
    - o 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

Web services

Motivação

- ✓ Garantir inter-operabilidade
- ✓ Usar protocolos web (HTTP e HTTPS) para transporte
    - o 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 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
- ✓ No SOAP toda a informação está incluída no envelope da mensagem
- ✓ O envelope inclui elementos de cabeçalho e de corpo
- ✓ Interacções:
    - o Oneway: mensagem undireccional do cliente para o servidor
    - o Pedido-resposta: interacção cliente-servidor-cliente
    - o Notificação: interacção unidireccional servidor-cliente
        - ▪ E.g. callback, notificação
    - o 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 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
  - o Ex. JAX-WS tem ferramentas para criar especificação a partir de interfaces Java

```
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();
        }
```

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 responde às questões
  - o Onde é que o Web service está localizado?
  - o O UDDI permite encontrar o serviço baseado na sua definição – *capability lookup*

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)

V - Segurança em sistemas distribuídos

Motivação

- Evitar acesso indevido aos dados;
- Proteger operações com alguma espécie de valor;
- Impedir acções de vandalismo;

Os sistemas (nomeadamente os distribuídos, que se tornam mais acessíveis), devem providenciar um conjunto de mecanismos que possa ser usados, em cada caso, para a implementação das políticas de segurança mais adequadas.

As características distribuídas destes sistemas requerem mecanismos de segurança mais sofisticados.

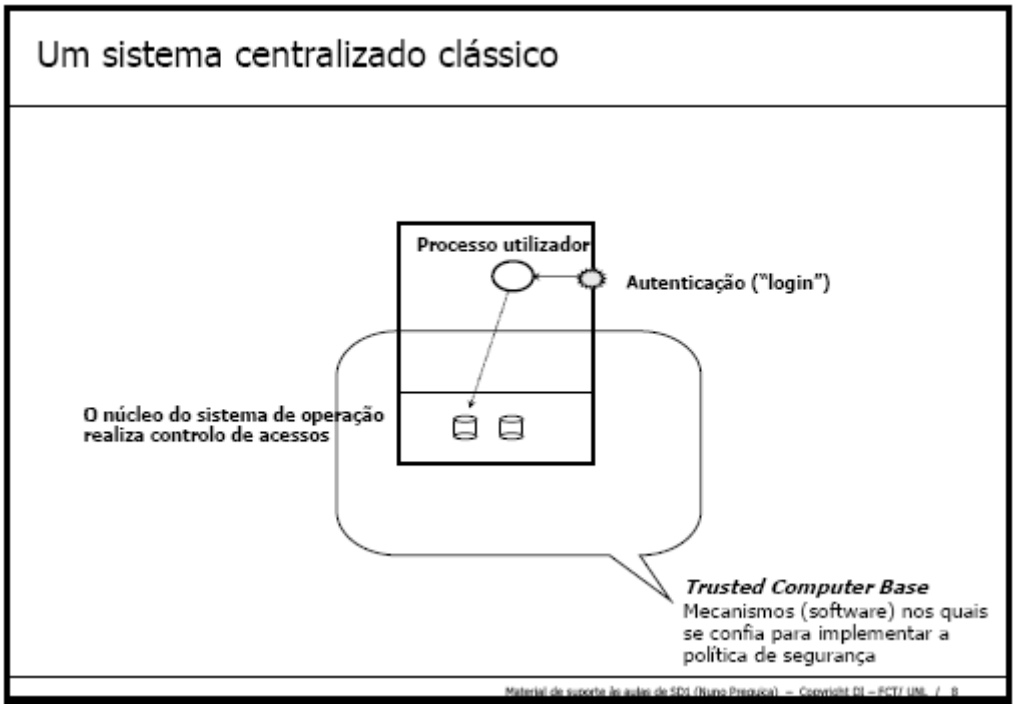
Modelo de Segurança

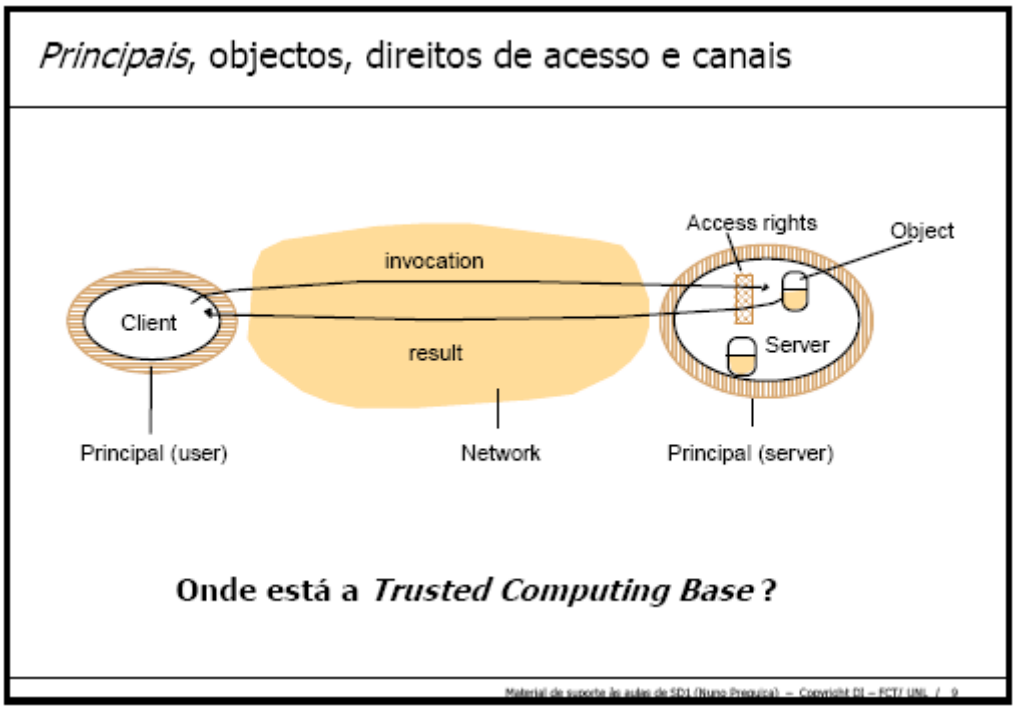
A segurança de um SD passa por:

- ✓ Autenticar os principais (autenticação);
- ✓ Verificar os seus direitos de acesso aos objectos (controlo de acessos);
- ✓ Utilizar canais seguros para impedir o acesso, alteração ou destruição indevida da informação

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).



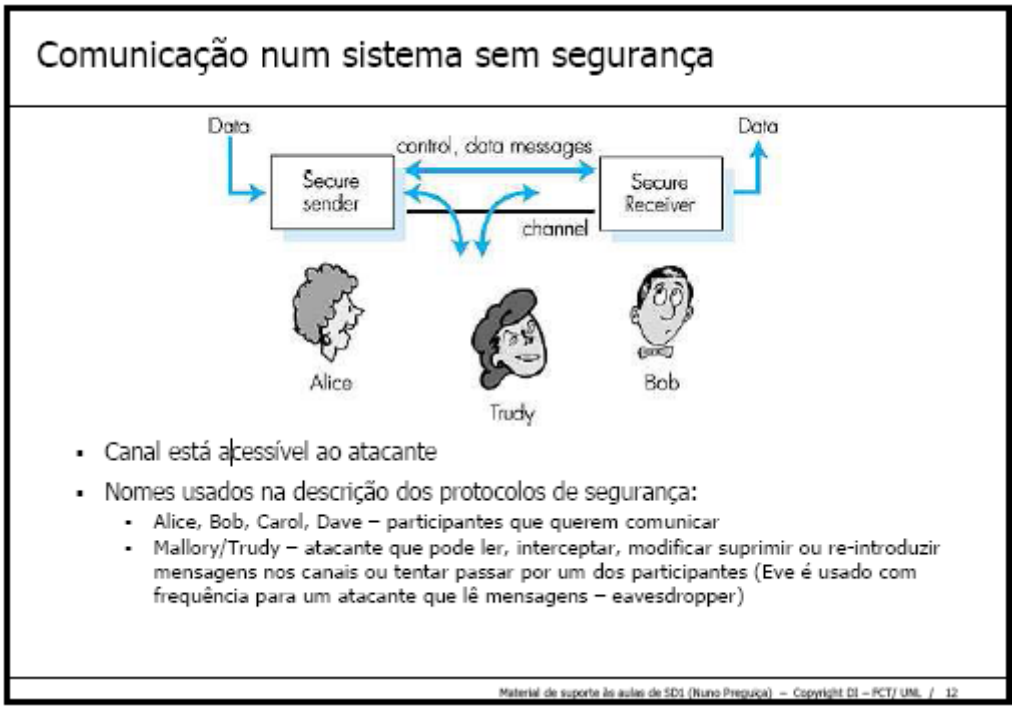


→ A segurança deve depender de um conjunto de mecanismos de segurança e de uma Trusted Computing Base mínima, porque assim:
- é mais fácil compreender os mecanismos de segurança,
- é mais fácil verificar a sua correcção,
- é mais fácil de implementar,
- o número de intervenientes é menor,
- só se paga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

Métodos de Ataque

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

É impossível enunciar todos os tipos de ataques, já que estão constantemente a descobrir-se novas.

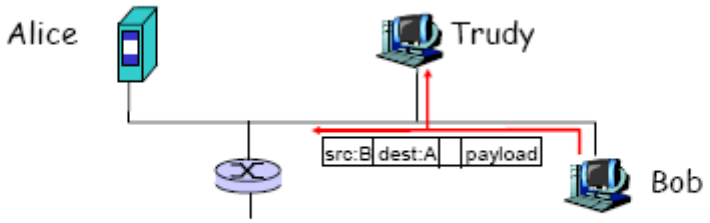


Tipos de Ataques em SD's através da comunicação

- 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)

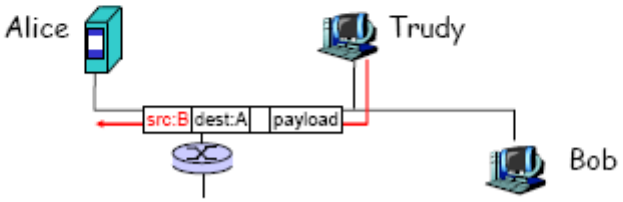
Eavesdropping (exemplo): cópia dos pacotes que transitam na rede

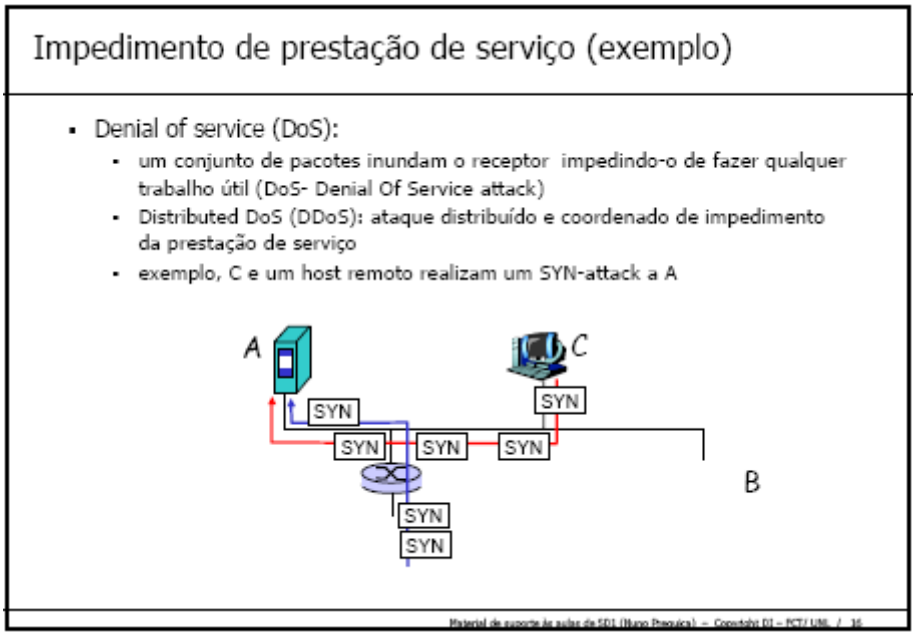
- *Packet sniffing*:
  - *broadcast media*
  - modo *promiscuous* da placa *ethernet* (NIC) permite copiar todos os pacotes
  - permite a leitura de todos os dados não cifrados (exemplo: *passwords*)
  - exemplo: *Trudy sniffs Bob's packets*



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





Canais Seguros

Objectivo: Trocar dados com confidencialidade, integridade e autenticidade

Num canal seguro,:

- Os interlocutores estão autenticados;
- o inimigo não pode copiar, alterar ou introduzir mensagens;
- o inimigo não pode fazer replaying de mensagens;
- o inimigo não pode re-ordenar as mensagens

Problemas do código móvel

O código obtido remotamente pode conter operações que comprometam a segurança de um sistema

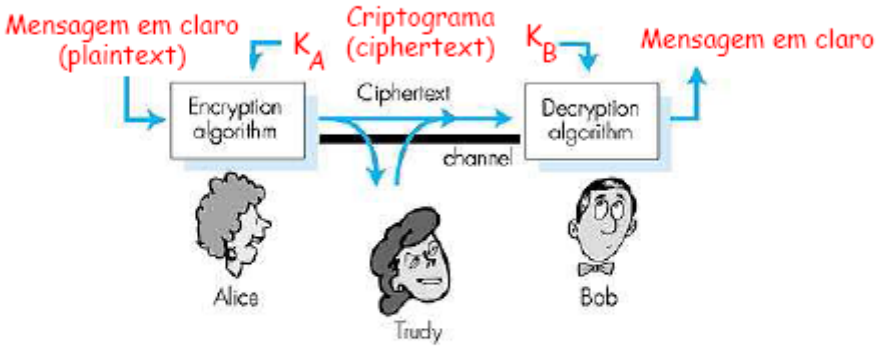
Como se pode minimizar o risco (exemplo Java VM):
- Verificar que o código apenas contém operações legítimas
- Manter o código obtido remotamente separado do código local
- Associar permissões ao código obtido remotamente. As permissões dadas podem depender da origem do código, o que pode ser verificado através da assinatura do mesmo

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_{Apriv}	Alice's private key (known only to Alice)
K_{Apub}	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

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 de Força Bruta

Chaves geradas de forma "totalmente" aleatória e com um número de bits suficiente são relativamente seguras quando sujeitas a ataques do tipo "força bruta"...

... pressupondo 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.

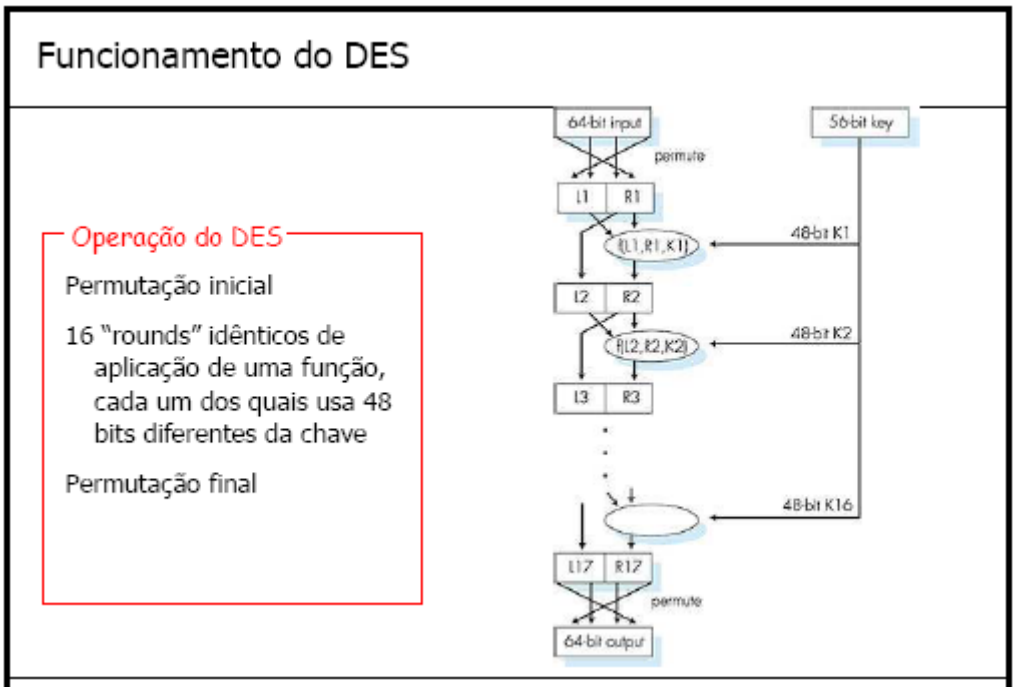
Criptografia Simétrica

Parceiros devem partilhar chave secreta

- ✓ Mensagem é cifrada e decifrada com a mesma chave
  - o E(k,M) = X
  - o D(k,X) = M
  - o Garante confidencialidade das mensagens
  - o 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.
  - o História atribulada. Está a cair em desuso pois a chave é demasiado pequena para a potência de cálculo actual. Uma forma de o tornar mais seguro usado em combinação com *cipher-block chaining*
- ✓ Triple DES - DES reforçado - chave com 128 bits.
  - o Aplica três vezes o algoritmo DES
- ✓ IDEA - International Data Encryption Algorithm - chave com 128 bits.
  - o Método desenvolvido na Europa. Após vários anos de utilização e divulgação pública não se conhecem ataques com êxito.
- ✓ AES - Advanced Encryption Standard
  - o Nova norma U.S.A. Definida por concurso



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.
  - o 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.
  - o Não está dependente da factorização de números

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:
  - o Calcular H(M) é fácil (computacionalmente)
  - o Dado H(M) é computacionalmente impossível calcular M
  - o 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”.
- ✓ 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.
- ✓ Função SHA-1 (Norma USA. Conhecida publicamente)
- ✓ Está-se a migrar para SHA-2
  - o Produz uma síntese de 160 bits

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.
- ✓ As chaves públicas são utilizadas para autenticação e para passar uma chave de sessão para utilização posterior de criptografia simétrica entre os dois principais.

Exposição das chaves secretas assimétricas

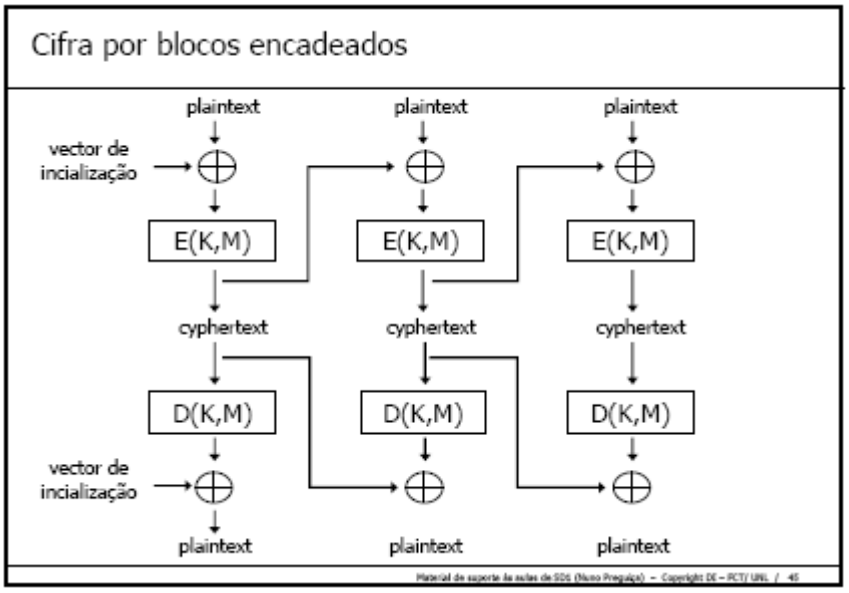
Se as chaves públicas e secretas apenas forem usadas para a autenticação e troca de uma chave simétrica de sessão com o outro parceiro, então:

1. as chaves públicas estão sempre expostas, mas
2. as chaves secretas 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 secretas 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 secreta pode estar gravada num cartão inteligente 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)



- ✓ 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.

- ✓ 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
- ✓ 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)
- ✓ 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:
  - o colocar um CRC
  - o colocar um número de ordem
  - o colocar um valor fixo pré-estabelecido, etc.

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:
     a. Calcula h = H(M+K)
     b. Envia M,h
  3. Ao receber "M,h", B calcula h'=H(M+K) e verifica integridade da mensagem se h=h'
- ✓ Só A conhece K, logo só A consegue gerar H(M+K) – autentica emissor de M e garante que a mensagem não foi modificada

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)

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.

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

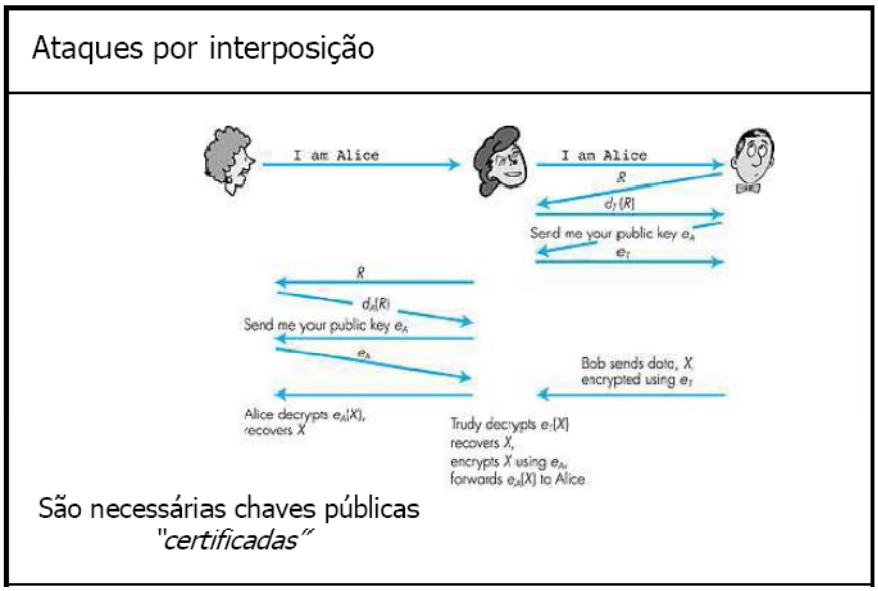
Protocolo de Needham-Schroeder com chaves públicas

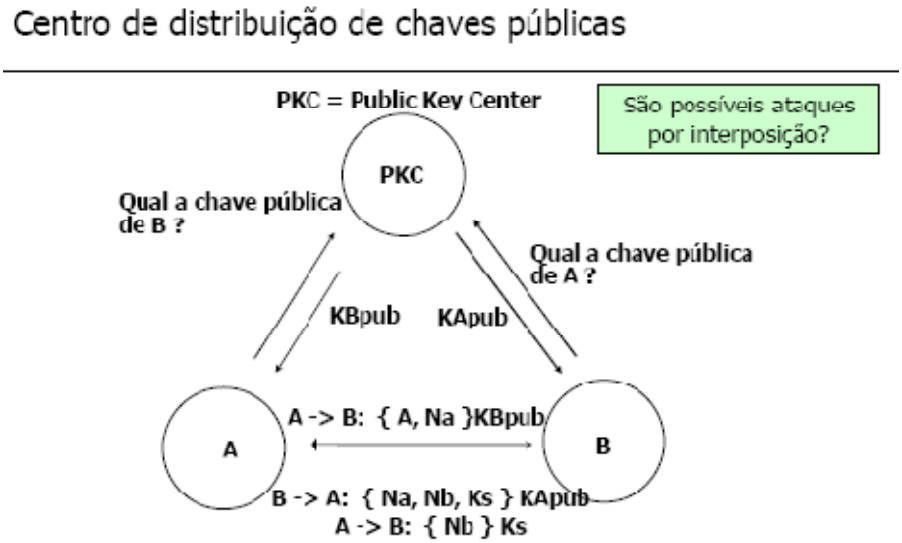
- ✓ Pressupondo que A e B conhecem as chaves públicas um do outro de forma certificada, podem estabelecer um canal seguro e autenticarem-se mutuamente através de:

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

- ✓ O que garante a A estar a falar com B?
  - o Receber Na em 2 – apenas B consegue obter Na, porque apenas B tem KBpriv
- ✓ O que garante a B estar a falar com A?
  - o Receber Nb em 3 – apenas A consegue obter Nb, porque apenas A tem KApriv
- ✓ O que garante que Ks é seguro?
  - o 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?
  - o Negoceia chave simétrica para usar durante a comunicação

Ataques por interposição (middleman)





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"

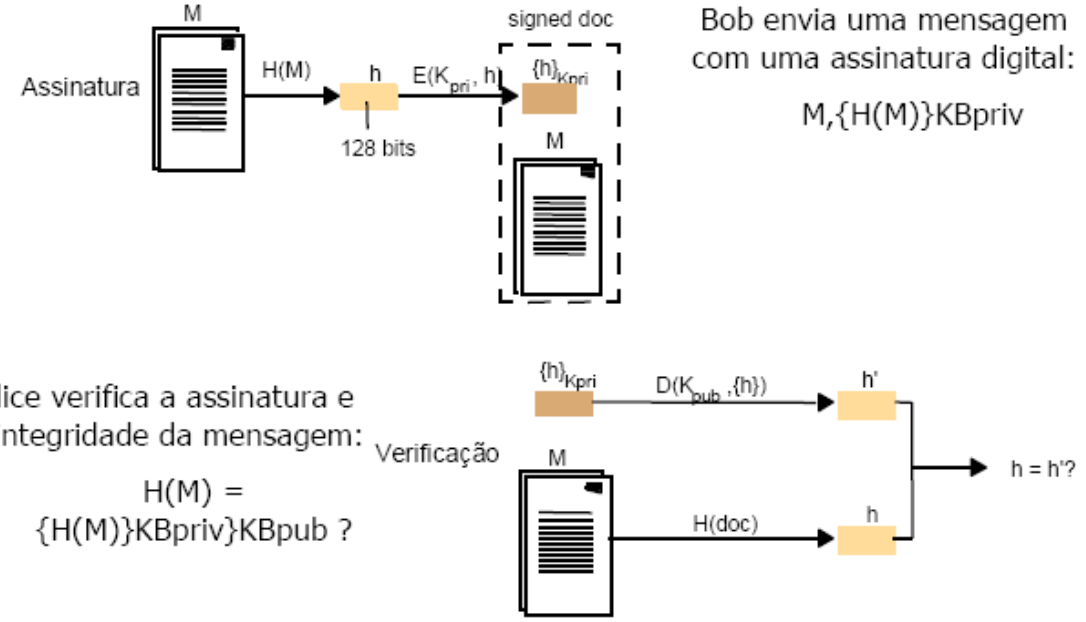
Distribuição das chaves públicas

- ✓ 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:
  - o 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.
  - o 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

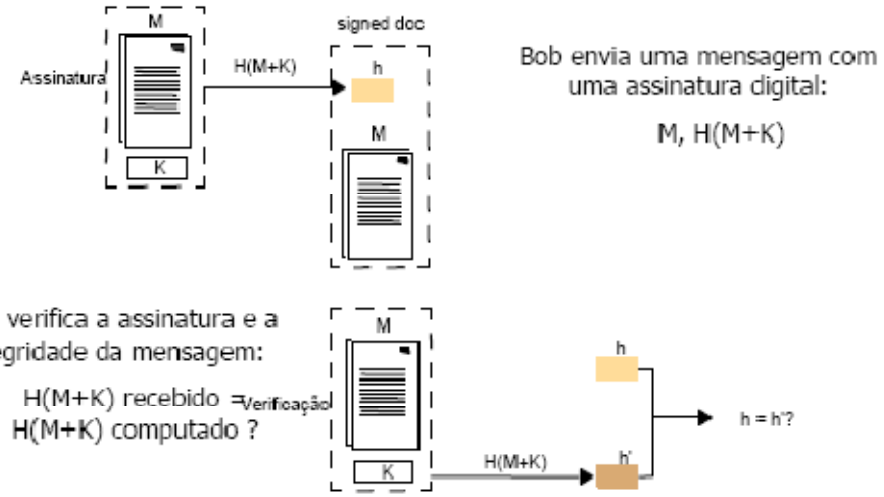
- ✓ Objectivo: desenvolver um mecanismo digital que substitua as assinaturas efectuadas num documento em papel
- ✓ Propriedades:
- ✓ Autenticação: o receptor deve poder verificar que a assinatura é autentica
- ✓ Integridade: a assinatura deve garantir que a mensagem assinada não foi alterada, nem durante 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

Utilização de assinaturas digitais com chaves públicas



- ✓ Assinatura digital da mensagem M efectuada por Bob: {H(M)}Kbpriv
  - o Mensagem a enviar: M,{H(M)}Kbpriv
  - o H(M): função de síntese segura
- ✓ Propriedades:
  - o Verificação ao receber M',{H(M)}KBpriv : H(M') = {{H(M)}KBpriv}Kbpub
  - o 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)
  - o Dimensão da assinatura pequena e constante
  - o Apenas assinatura é cifrada – mensagem pode passar em claro

Utilização de assinaturas digitais com chaves simétricas

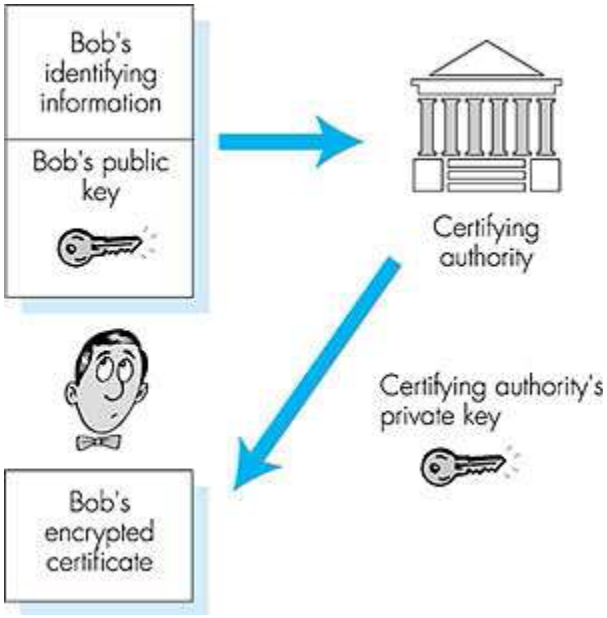


- ✓ Assinatura digital da mensagem M efectuada por Bob: h=H(M+K)
  - o Mensagem a enviar: M,h
  - o H(M): função de síntese segura
- ✓ Propriedades:
  - o Verificação ao receber M', h' : H(M'+K) = h'
  - o Autenticação, integridade e não repudiamento garantidos porque apenas B consegue criar H(M+K) e 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:
  - o **Certificate type:** Public key
  - o **Name:** Bob's Bank
  - o **Public key:** Kbpub
  - o **Certifying authority:** Fred – The Bankers Federation
  - o **Signature:** {Digest(field 2 + field 3)}$_{Kfpriv}$

Certificate Authority



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.
  - o 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

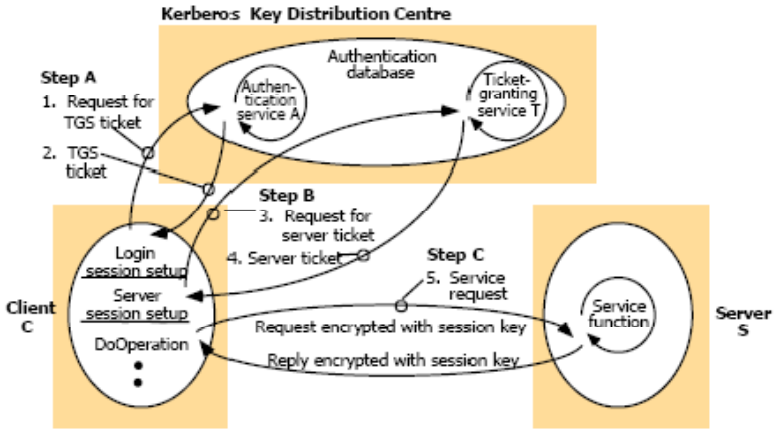
- ✓ Alice e Bob pretendem trocar entre si um senha ou chave de sessão (K) sem que esta passe em claro na rede (nota: não se pretendem autenticar).
- ✓ Sejam dois números aleatórios m e n que são parâmetros públicos do algoritmo:
  1. A gera um número secreto Xa e calcula: Ya = m ^ Xa mod n
  2. B gera 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 em inverso (one-way function).
- ✓ E.g. m = 2; n = 64;
- ✓ Xa = 2; Ya= 4; Xb =3; Yb = 8

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
  - o Trudy pode ficar a conhecer o conteúdo das mensagens trocadas para todas as sessões que tenha gravado
  - o 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.
  - o Xa, Xb, Ka, Kb são segredos apenas enquanto dura a sessão: depois podem (devem) ser descartados

O sistema Kerberos

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



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)
  - o Permite aplicar tempo limite a tickets, revogando autorizações dadas
  - o 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.

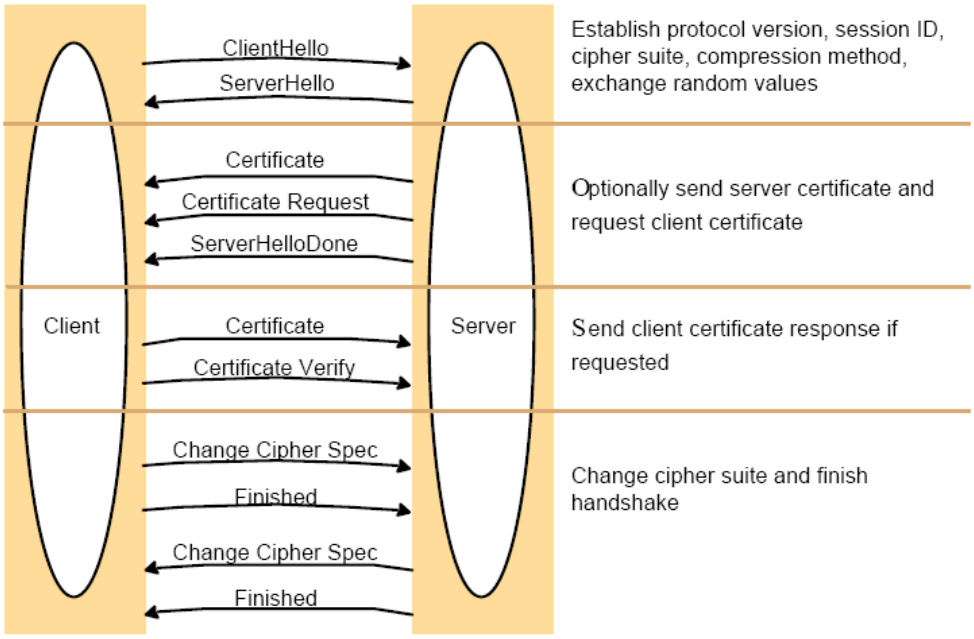
Vantagens da variante Kerberos sobre o protocolo Needham-Schroeder simples

- ✓ 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)
  - o Validade dos tickets é geralmente de algumas horas
  - o 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 (reparese que os clientes são muito mais numerosos que os servidores)

SSL - Secure Socket Layer

- ✓ 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).
- ✓ Segurança:
  - o autenticação do servidor
  - o cifra dos dados
  - o autenticação do cliente (opcional)

SSL handshake protocol



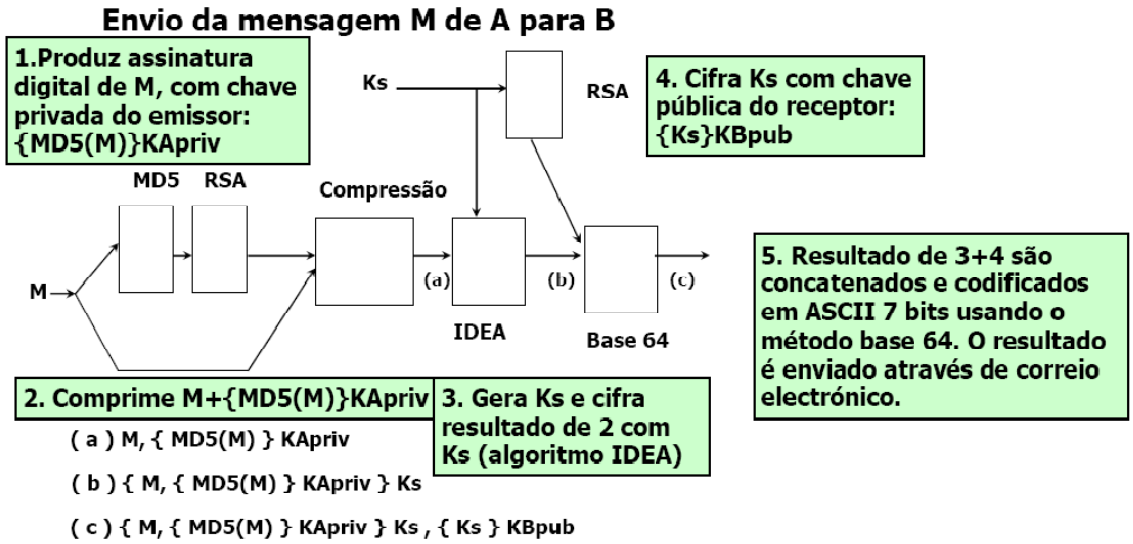
Component	*Description*	*Example*
Key exchange method	the method to be used for exchange of a session key	RSA with public-key certificates
Cipher for data transfer	the block or stream cipher to be used for data	IDEA
Message digest function	for creating message authentication codes (MACs)	SHA

- 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

Pretty good privacy (PGP)

- ✓ Esquema de cifra e assinatura de email.
- ✓ 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.

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).
- ✓ 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).

VI - Introdução à sincronização e algoritmos distribuídos

Ordenação de eventos

- ✓ Dados dois eventos A e B, executados em locais distintos, como estabelecer uma relação temporal ou uma relação de ordem entre A e B?
- ✓ Uma solução é associar uma estampilha temporal a cada evento e comparar as estampilhas. A estampilha temporal pode ser o "tempo" ("hora") em que o evento ocorreu.
- ✓ Esta aproximação exige que os relógios estejam sincronizados.
- ✓ Este requisito é válido em qualquer caso, em particular, num sistema distribuído em que os eventos A e B são executados em computadores distintos.

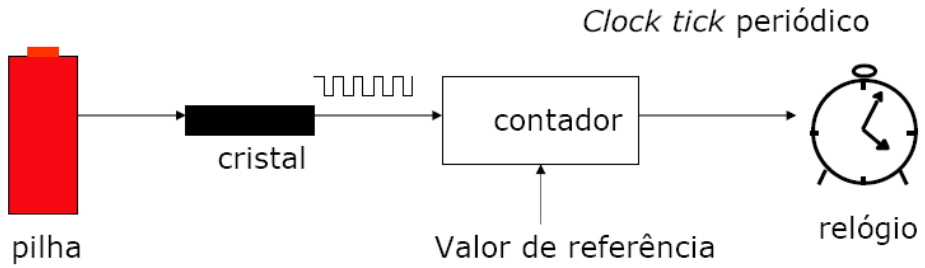
Sistemas oficiais de contagem do tempo

- ✓ Desde 1967, existe um standard para medir o (passar do) tempo: Tempo Atómico Internacional ou TAI, que é definido como:
  - o 1 s = 9 192 631 770 períodos de oscilação do átomo de Cesium 133
  - o 1 dia = 3 000 * 12 = 86 400 segundos
  - o 1 ano = 365,25 dias
- ✓ O TAI oficial é a média de 30 relógios atómicos.
- ✓ Como a terra está a desacelerar, o dia solar tem actualmente mais 3 ms que o dia TAI.
- ✓ O UTC (tempo universal coordenado) é um standard para determinar o valor do tempo ("hora"). É baseado no tempo TAI, sincronizado com o dia solar, através da adição de "leap seconds".

Acesso ao tempo UTC

- ✓ A qualidade com que se consegue obter o valor do UTC depende da qualidade da fonte original e da estabilidade do tempo de propagação
- ✓ Métodos de obtenção do UTC
  - o Via GPS : cerca de 1ns-10Ts; GLONASS: 20ns
  - o Via rádio com um erro de 10 ms
  - o Via telefone, com um erro de 10+ ms
- ✓ Conseguem-se sincronizar relógios baseados em cristais de quartzo com o UTC com um erro de cerca de 10 Ts, fazendo variar o valor de referência que determina o número de oscilações do cristal por clock tick, atrasando ou adiantando de forma progressiva o relógio.

A estabilidade e qualidade do relógio



- ✓ Não há dois cristais cujas frequências de oscilação sejam absolutamente idênticas. Por isso os relógios derivam ("clock drift"). Por exemplo, se a frequência é de 1 MHz, um erro de 1/10**6 conduz à deriva de 1 s em cada 10**6 segundos ou 1 segundo em cada 11.6 dias.
- ✓ Os relógios de computador banais têm um erro relativo que pode chegar a 1/100.000, o que conduz a 0,8 segundos / dia!

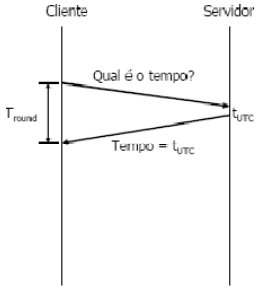
Sincronização de relógios

- ✓ Caso seja a hora oficial, os mesmos dizem-se externamente sincronizados.
- ✓ Se um conjunto de relógios está externamente sincronizado, então também está sincronizado internamente.
- ✓ Um conjunto de relógios diz-se internamente sincronizado se estes marcam a mesma hora (isto é, se a diferença entre as respectivas horas é inferior a um limite positivo conhecido), independentemente de se esta é ou não a hora "oficial".
- ✓ A sincronização externa também se diz sincronização temporal.
- ✓ A sincronização interna também se diz sincronização das frequências.

Sincronização externa: algoritmo de Cristian

- ✓ Pressuposto: latência da rede é pequena quando comparada com a precisão desejada
- ✓ Existe um servidor de tempo que tem acesso a uma fonte UTC
- ✓ Cada computador pergunta periodicamente ao servidor o tempo
- ✓ Cliente fixa $T_{Cliente} = T_{UTC} + T_{round}/2$
- ✓ Seja *min* o tempo mínimo de propagação, cliente pode receber resultado entre:
  - o $[T_{UTC}+min, T_{UTC}+T_{round}-min]$
- ✓ Erro: $\pm(T_{round}/2-min)$

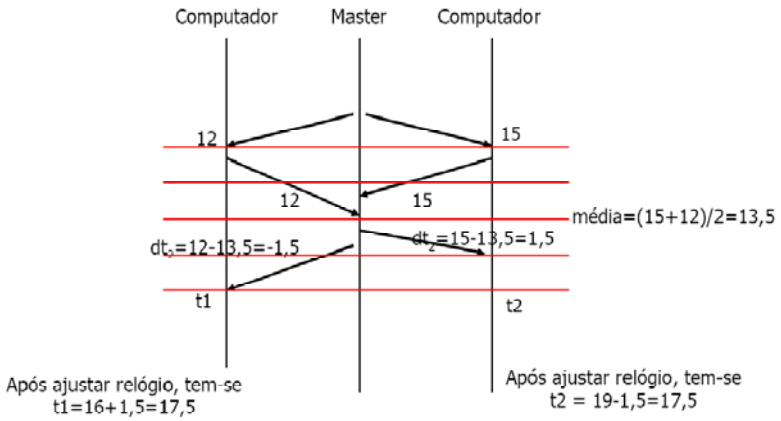
- ✓ Não lida com relógio avariados ou servidores malévolos
- ✓ Lida com a variabilidade das condições da rede fazendo vários pedidos ao servidor e usando aquele com menor T_{round}



Sincronização interna: algoritmo de Berkeley

Pressuposto: nenhum computador tem acesso a uma fonte UTC. Computadores pretende-se sincronizar mutuamente

- ✓ Existe um computador que actua como master
- ✓ Master periodicamente obtém o valor do relógio dos outros computadores
- ✓ Master calcula média e envia a correcção a efectuar a cada computador
    - ▪ ti=msg.ti+Tround_i/2
    - ▪ Correcção: dti=ti - avg ti
  - o Valores muito distantes da média são ignorados
- ✓ Quando um master falha é eleito outro



- ✓ O cálculo da média de um número elevado de máquina deve permitir cancelar o que uns relógios se adiantam pelo que outros se atrasam
- ✓ Ignorar valores muito distantes da média permite tolerar relógios avariados
- ✓ Envio do ajuste a efectuar permite não introduzir mais incerteza devido à propagação de mais uma mensagem

NTP - Network Time Protocol

- ✓ Permite a um conjunto de computadores na Internet (uma WAN) obterem o tempo UTC sincronizando-se directamente com fontes primárias de tempo
- ✓ Permite a sincronização mesmo na presença de grandes variações do RTT e emprega técnicas estatísticas para filtrar respostas inadequadas de servidores pois memoriza a história dos mesmos
- ✓ Tolera falhas e partições da rede mesmo prolongadas e incorpora reconfiguração dinâmica, redundância e escalabilidade

Organização NTP

- ✓ Existem servidores que têm receptores de UTC
- ✓ Os outros servidores organizam-se hierarquicamente, sincronizando-se com os níveis superiores
- ✓ Organização dinâmica
- ✓ Mensagens UDP
- ✓ Três tipos de sincronização:
  - o Multicast: numa rede local, um servidor emite o tempo em multicast
  - o Procedure-call: similar ao algoritmo de Cristian
  - o Simétrico: servidores de níveis superiores trocam o seu tempo para melhor precisão

- ✓ Os ataques possíveis a um serviço destes são os seguintes:
  - o A. Servidor falso
  - o B. Modificação das mensagens
  - o C. Replaying
  - o D. Supressão de mensagens
  - o E. Introdução de atrasos nas mensagens
- ✓ A) e B) são combatidos através de assinaturas digitais.
- ✓ D) e E) são combatidos através de servidores redundantes.
- ✓ C) é combatido através da parametrização adequada dos intervalos de polling pois através da forma como o protocolo está construído é possível detectar esta forma de ataque.

Utilização de relógios num sistema distribuído

- ✓ O valor do relógio pode usar-se para indicar o momento em que um dado evento ocorreu
- ✓ Ao valor do relógio associado a um evento chama-se estampilha temporal

Relação *aconteceu antes* (Lamport 1978)

- ✓ Objectivo: criar relação similar à causalidade física
- ✓ Se o evento e1 pode ser a causa do evento e2, então e1 aconteceu antes de e2
- ✓ A relação aconteceu antes ou precede (→) é definida por:
  - o e1→e2, se e1 e e2 ocorreram no mesmo processo e e1 ocorreu antes de e2
  - o e1→e2, se e1 e e2 são, respectivamente, os eventos de enviar e receber a mensagem m
  - o e1→e2, se ∃e3: e1→e3 e e3→e2 (relação transitiva)
- ✓ Dois eventos e1, e2 dizem-se concorrentes (e1||e2) se ¬ e1→e2 e ¬ e2→e1

Relógios lógicos (Lamport 1978)

- ✓ Um relógio lógico (de Lamport) é um contador monotonicamente crescente, usado para atribuir uma estampilha temporal a um evento. Cada processo i, mantém um relógio lógico Li que actualiza da seguinte forma:
  - o Seja e um evento executado no processo i, faz-se:
    - Li := Li + 1
    - C(e):=Li, com C(e) a estampilha temporal atribuída ao evento e.
  - o Se e=send(m), aplica-se a regra anterior e envia-se (m,t), com t=C(send(m))
  - o Se e=receive(m,t), faz-se Li=max(Li,t) e, de seguida, aplica-se a regra base

Ordenação total usando relógios lógicos

- ✓ Por vezes, é interessante ordenar totalmente os eventos que ocorrem num sistema distribuído
- ✓ Dado um evento e executado no processo pi (i=1..n) com estampilha temporal Le, a estampilha temporal global é (Le,i).
- ✓ A ordem global das estampilhas temporais é definida por:
  - o (Le1,i)<(Le2,j), sse Le1<Le2 ou Le1=Le2 e i < j

Relógios vectoriais

- ✓ Um relógio vectorial num sistema com n processos é um vector de n inteiros, V[1..n]. Casa processo i mantém um relógio vectorial Vi que usa para atribuir uma estampilha temporal aos eventos locais e actualiza da seguinte forma:
- ✓ Inicialmente, Vi[j]=0, ∀i,j
- ✓ Antes de etiquetar um evento e no processo i, faz-se: Vi[i] := Vi[i]+1. C(e)=Vi
- ✓ Se e=send(m), aplica-se a regra anterior e envia-se (m,t), com t=C(send(m))
- ✓ Se e=receive(m,t) no processo i, faz-se Vi[j]=max(Vi[j],t[j]) e, de seguida, aplica-se a regra base
- ✓ Dados dois relógios vectoriais V1 e V2, tem-se:
  - o V1=V2, sse V1[i]=V2[i], ∀i
  - o V1<=V2, sse V1[i]<=V2[i], ∀i
  - o V1<V2, sse V1<=V2 ∧ V1≠V2
- ✓ Dados dois eventos, e1 e e2, tem-se e1→e2 ⍰ C(e1)<C(e2)
  - o Intuitivamente, como um relógio vectorial resume os eventos vistos, se e1→e2 então necessariamente o resumo dos eventos vistos aquando de e2 tem de incluir todos os eventos vistos aquando de e1(incluindo e1). Pode-se demonstrar por indução no comprimento da sequência de eventos que estabelece a causalidade.
- ✓ Dados dois eventos, e1 e e2, tem-se C(e1)<C(e2) ⇒ e1→e2
  - o Intuitivamente, como um relógio vectorial resume os eventos vistos, se o resumo de eventos vistos aquando de e2 inclui o evento e1, necessariamente e1→e2
- ✓ Dados dois eventos, e1 e e2, e1 e e2 são concorrentes sse nem C(e1)<C(e2) nem C(e2)<C(e1)
  - o Intuitivamente, se e1 e e2 são concorrentes, então nem o resumo de e1 pode incluir e2 nem o resumo de e2 pode incluir e1. Se cada resumo não inclui o outro evento é porque nenhum evento aconteceu no passado do outro.

Ordenação da entrega das mensagens

Um elementos pode enviar mensagens para o grupo

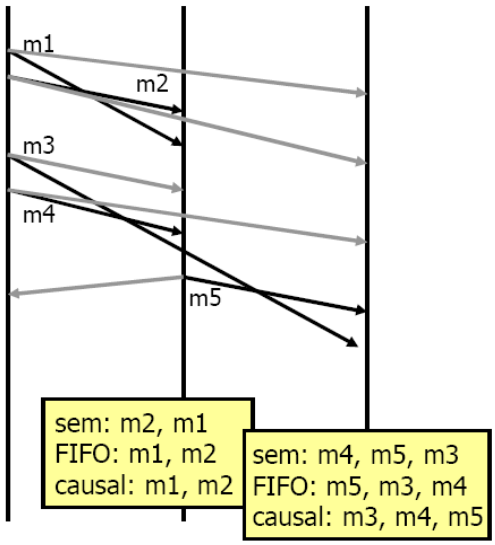
- ✓ send(m) : envia mensagem m para o grupo
- ✓ deliver(m) : entrega mensagem m à aplicação

Para simplificar, assume-se que as mensagens são entregues de forma fiável

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

Ordenação da entrega das mensagens

- ✓ Sem ordem: mensagens são entregues à aplicação pela mesma ordem que são recebidas
- ✓ Ordem FIFO: as mensagens da mesma origem são entregues pela mesma ordem que foram enviadas
- ✓ Ordem causal: as mensagens que estão potencialmente relacionadas causalmente são entregues pela ordem causal em todos os processos
  - o Duas mensagens m1 e m2 são entregues por ordem causal sse: send(m1)<send(m2) => deliver(m1)<deliver(m2)
- ✓ Ordem total: todos os processos entregam as mensagens pela mesma ordem
  - o Existe um processo (sequenciador) responsável por dar um número de ordem às mensagens
  - o Mensagens são entregues pela ordem definida pelo sequenciador
  - o Assumindo propagação por ordem FIFO
    - Cada processo mantém um relógio lógico e estampilha mensagem com o seu relógio
    - Cada processo actualiza relógio e difunde ACK ao receber mensagem (com estampilha superior à original)
    - Cada processo mantém uma fila de mensagens ordenadas pela estampilha. A primeira mensagem pode ser entregue quando o processo recebeu ACKs de todos os processos (com excepção da origem e do próprio)



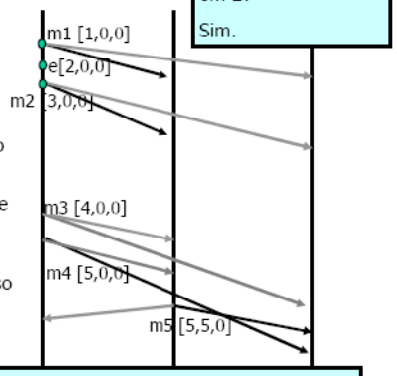
Ordem total com relógios de Lamport

- ✓ Hipóteses
  - o Todas as mensagens emitidas pelo mesmo emissor são entregues por ordem em qualquer receptor (FIFO)
  - o Não se perdem mensagens
  - o Todas as mensagens emitidas transportam a etiqueta temporal de Lamport do emissor e são também enviadas para o processo local (colocadas na fila local)
- ✓ Protocolo
  - o O receptor de uma mensagem difundida emite um ACK para o grupo por cada mensagem recebida; esse ACK tem uma etiqueta superior à mensagem a que diz respeito
  - o Cada participante tem uma fila de mensagens recebidas ordenada pelas etiquetas temporais (se duas mensagens têm a mesma etiqueta usa-se o endereço para desempatar por exemplo)
  - o Sempre que uma mensagem está na cabeça da lista e já se receberam todos os seus ACK, ela pode ser entregue localmente e os ACKs suprimidos
- ✓ Resultado
  - o As mensagens são entregues segundo uma ordem total que respeita necessariamente a causalidade

Como ordenar as mensagens: hip 1

- Ordem causal: as mensagens que estão potencialmente relacionadas causalmente são entregues pela ordem causal em todos os processos
  - Duas mensagens m1 e m2 são entregues por ordem causal sse: send(m1)<send(m2) => deliver(m1)<deliver(m2)
- Assume-se que mensagens do mesmo processo são propagadas por ordem FIFO
- Cada processo mantém um relógio vectorial que actualiza como explicado anteriormente
- Seja $\mathbf{v_m}$ o relógio vectorial da mensagem **m** recebida de **i** e $\mathbf{v_j}$ o relógio vectorial do processo **j**, **m** pode ser entregue se:
  - $v_m[n] <= v_j[n]$ para n <> i
  - nenhum requisito para $v_j[i]$. Porquê?

Todas as mensagens conhecidas em i aquando do envio de m já são conhecidas em j também



Pode entregar m5 em 3 antes de m4?
Não, porque $v_3[1]=4$ e $v_{m5}[1]=5$. m5 fica em fila de espera até se receber m4.

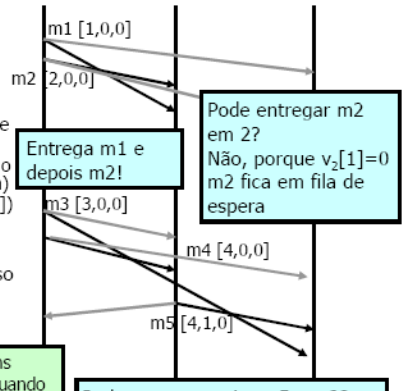
Como ordenar as mensagens: hip 2

- Ordem causal: as mensagens que estão potencialmente relacionadas causalmente são entregues pela ordem causal em todos os processos
  - Duas mensagens m1 e m2 são entregues por ordem causal sse: send(m1)<send(m2) => deliver(m1)<deliver(m2)
- Cada processo mantém um relógio vectorial que actualiza:
  - No envio, de forma normal. A mensagem inclui o relógio da origem (e informação sobre a origem)
  - Na recepção,fazendo apenas Vi[j]=max(Vi[j],t[j]) (i.e., não se incrementa o contador local)
- Seja $\mathbf{v_m}$ o relógio vectorial da mensagem **m** recebida de **i** e $\mathbf{v_j}$ o relógio vectorial do processo **j**, **m** pode ser entregue se:
  - $v_m[n] <= v_j[n]$ para n <> i
  - $v_m[i] = v_j[i]+1$

Entrega m1 e depois m2!

Pode entregar m2 em 2?
Não, porque $v_2[1]=0$ m2 fica em fila de espera

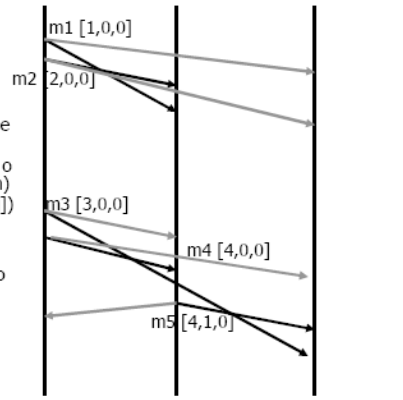
Todas as mensagens conhecidas em i aquando do envio de m já são conhecidas em j também



Pode entregar m4 e m5 em 3?
Não, porque $v_1[1]=2$ e $v_m[1]=4$
m4 e m5 ficam em fila de espera

Como ordenar as mensagens: hip 2

- Ordem causal: as mensagens que estão potencialmente relacionadas causalmente são entregues pela ordem causal em todos os processos
  - Duas mensagens m1 e m2 são entregues por ordem causal sse: send(m1)<send(m2) => deliver(m1)<deliver(m2)
- Cada processo mantém um relógio vectorial que actualiza:
  - No envio, de forma normal. A mensagem inclui o relógio da origem (e informação sobre a origem)
  - Na recepção,fazendo apenas Vi[j]=max(Vi[j],t[j]) (i.e., não se incrementa o contador local)
- Seja $\mathbf{v_m}$ o relógio vectorial da mensagem **m** recebida de **i** e $\mathbf{v_j}$ o relógio vectorial do proceso **j**, **m** pode ser entregue se:
  - $v_m[l] <= v_j[l]$ para l <> i
  - $v_m[i] = v_j[i]+1$

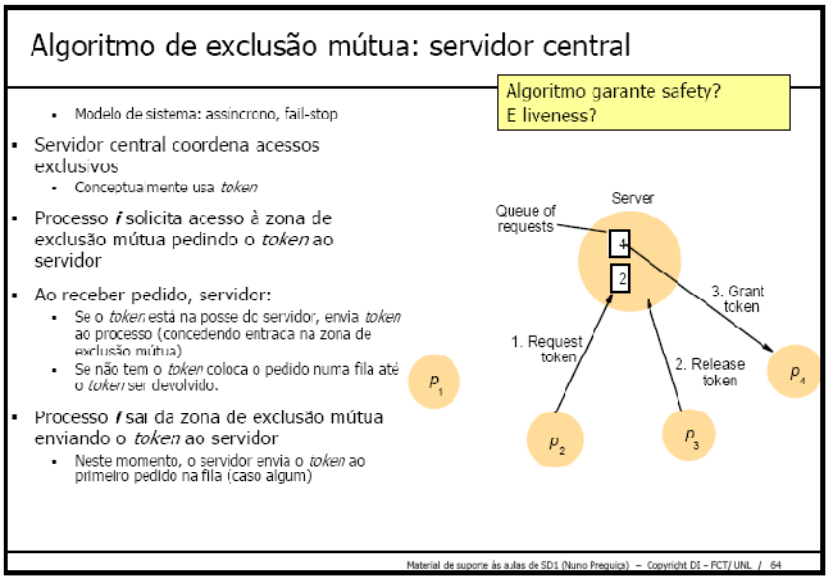


Algoritmos distribuídos

- ✓ Coordenação e consenso
  - o Processos devem-se coordenar or concordar num (ou mais) valores
  - o Alguns problemas
    - ▪ Eleição
    - ▪ Consenso
    - ▪ Exclusão mútua distribuída
  - o Modelo de sistema
    - ▪ Síncrono (Existe limite temporal para: latência da propagação, tempo para executar cada passo, ritmo de desacerto dos relógios) -> permite usar timeouts para detectar falhas dos processos
    - ▪ Assíncrono (Não existem limites temporais)
  - o Modelo de falhas
    - ▪ Falhas dos processos: fail-stop
    - ▪ Canais fiáveis (mensagens não se perdem; pode ser construído por retransmissão das mensagens)

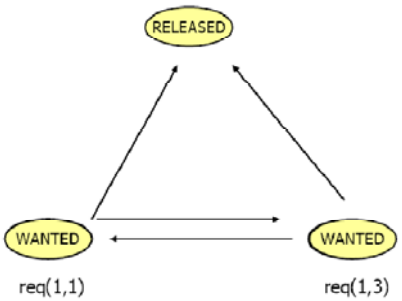
Exclusão mútua

- ✓ Um algoritmo de exclusão mútua distribuída é um algoritmo que permite coordenar os acessos a um recurso, garantindo que em cada momento apenas um processo tem acesso ao recurso
- ✓ Um processo executa as seguintes operações para iniciar e terminar o acesso exclusivo ao recurso:
  - o enter: solicita o acesso exclusivo. Esta operação bloqueia até o pedido ser concedido
  - o exit: termina o acesso exclusivo
- ✓ Requisitos de um algoritmo:
  - o E1 (safety): em cada momento, no máximo, um processo tem acesso exclusivo
  - o E2 (liveness): as operações enter e exit terminam.



Algoritmo de exclusão mútua: Ricart e Agrawal

- ✓ Permitir a entrada na zona de região crítica usando a ordem global definida por (rel. lógico, id processo)



- ✓ Inicialização:
  state:= RELEASED
- ✓ Para entrar na região crítica:
  State := WANTED
  multicast pedido (T=rel.lógico)
  Espera até receber N-1 respostas
  State := HELD
- ✓ Ao receber pedido <Ti,pi> em pj
  IF State=HELD OR (State=WANTED AND (T,pj)<(Ti,pi))
  coloca pedido em fila
  ELSE
  responde imediatamente a pi
- ✓ Ao sair da região crítica:
  State := RELEASED
  Responde a todos os pedidos em fila

Eleição

- ✓ Algoritmo de eleição é um algoritmo para escolher um processo para exercer um papel particular (coordenador) num sistema.
- ✓ Quando se detecta a falha do coordenador, executa-se o algoritmo de eleição. Num dado momento, um processo que está a participar num algoritmo de eleição diz-se participante. Os outros dizem-se não-participantes.
- ✓ Cada processo p_i tem uma variável $eleito_i$ que indica o processo que é o coordenador. Quando não existe coordenador $eleito_i = \perp$
- ✓ Cada processo tem um identificador e o algoritmo de eleição escolherá o processo com maior identificador
- ✓ Requisitos:
  - o E1 (safety): um processo participante p_i tem $eleito_i = \perp$ ou $eleito_i = P$ com P o processo com maior identificador
  - o E2 (liveness): todos os processos participante fazem $eleito_i \neq \perp$ ou falham

Consenso

- ✓ Um algoritmo de consenso permite a um conjunto de processos concordar num valor proposto.
- ✓ Um processo i inicia o protocolo no estado não-decidido e propõe um valor vi. No fim do protocolo, todos os participantes estão no estado decidido com o valor da decisão igual a v.
- ✓ Requisitos de um algoritmo:
  - o Terminação: todos os processos correctos acabam por decidir.
  - o Acordo: todos os processos correctos que chegam ao estado decidido tem o mesmo valor para a decisão
  - o Integridade: se todos os processos correctos propuseram o mesmo valor, esse foi o valor escolhido

- ✓ Mensagens multicast são entregues nos nós que não falham se o emissor não crashar
  - o Podem falhar até f nós
- ✓ Algoritmo com rondas sucessivas
- ✓ Em cada ronda propagam-se os valores recebidos
- ✓ No fim, escolhe-se o menor valor
- ✓ São precisas f+1 rondas porque podem crashar até f nós. O nó que crasha pode ser aquele que propôs o menor valor em cada ronda. Esse nó pode ter enviado a mensagem a apenas um outro nó.

- ✓ Inicialização
  $Values_i^1 := \{v_i\}$; v_i valor proposto por i
  $Values_i^0 := \{\}$
- ✓ Em cada ronda r:
  Cada elemento faz multicast dos valores ainda não enviados ($Values_i^r - Values_i^{r-1}$)
  $Values_i^{r+1} := Values_i^r$
  Para cada mensagem recebida adiciona valores recebidos aos conjunto $Values_i^{r+1}$
- ✓ Após a ronda f+1:
  Decisão := mínimo($Values_i^{f+1}$)

Impossibilidade do consenso na presença de falhas

- ✓ Provou-se a impossibilidade de realizar um algoritmo que garanta resolver o problema do consenso num sistema assíncrono na presença de uma falha num processo
  - o Intuitivamente: é sempre possível construir uma sequência de mensagens em que um processo que é dado como falhado pode acordar no momento errado (Mas pode ser obtido com probabilidade maior que 0)
- ✓ Aproximações para contornar impossibilidade, na prática
  - o Mascarar falhas de processos
  - o Detectores de falhas
  - o Consenso usando aleatoriedade

Necessidade de coordenar operações em vários servidores

- ✓ Num sistema distribuído surge frequentemente a necessidade de coordenar a execução atómica de um conjunto de operações em vários servidores (a que chamaremos transacção apesar de não necessariamente implementar as propriedades ACID)
- ✓ É necessário que os servidores envolvidos concordem relativamente ao resultado da execução da operação
- ✓ Para coordenar a execução de operações em vários servidores deve-se implementar um protocolo de commit atómico
- ✓ Um processo actua como coordenador e é responsável por, no fim, confirmar (commit) ou abortar a transacção
- ✓ Os processos em que são executadas operações actuam como participantes
- ✓ O protoclo é executado apenas após executar as operações e para as confirmar ou abortar.

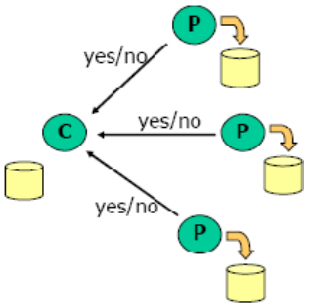
Protocolo *one-phase commit*

- ✓ Cliente
  - o Solicita ao coordenador para confirmar (commit) ou abortar a transacção
- ✓ Coordenador
  - o Informa participantes sobre resultado da transacção
- ✓ Participante
  - o Se a transacção terminar com sucesso, o resultado deve ser escrito em memória estável
  - o Se a transacção for abortada, os resultados intermédios devem ser abortados
- ✓ Problema: Não permite a um participante abortar uma transacção de forma unilateral

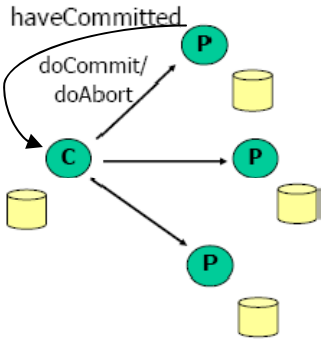
Protocolo *two-phase commit (2PC)*

- ✓ Objectivo: permitir aos participantes abortar uma transacção unilateralmente
- ✓ O algoritmo processa-se em duas fases (quando se pretende confirmar uma transacção)
  - o Fase 1: o coordenador pergunta aos participantes se estão preparados para confirmar a transacção. Um participante que pretenda confirmar a transacção deve ser capaz de o fazer (gravando em memória estável o estado necessário).
  - o Fase 2: os participantes executam a decisão conjunta. Se algum participante tiver decidido abortar a transacção, a transacção será abortada.
- ✓ Modelo de falhas: processos falham por fail-stop. Um processo falhado é substituído por um novo processo com a informação gravada em memória estável.

- ✓ **Fase 1:**
  - o O coordenador grava informação sobre início do protocolo.
  - o O coordenador pergunta aos participantes se pretendem confirmar a transacção.
  - o Um participante grava em memória estável a sua decisão (incluindo todo o estado necessário para confirmar a transacção) e envia o seu voto ao coordenador.
  - o Um participante que tenha votado para abortar a transacção aborta-a imediatamente



- ✓ **Fase 2:**
- ✓ O coordenador recolhe todos os votos. Se algum participante votou não, a transacção será abortada.
- ✓ O coordenador grava decisão.
- ✓ O coordenador informa todos os participantes que votaram sim da decisão.
  - o Não é necessário informar os que votaram não porque estes já abortaram a transacção.
- ✓ Ao receber a decisão, um participante actua de acordo com a mesma e grava o estado final da transacção.



- ✓ Se a transacção foi confirmada, envia uma mensagem *haveCommited* ao coordenador.
- ✓ Quando o coordenador recebe confirmações de todos os participantes, dá a transacção por concluída. A informação sobre a transacção pode ser removida.

Tolerância a avarias

- ✓ Não detecção de mensagens
  - o Detectadas por um temporizador no coordenador e nos participantes
- ✓ Falhas do coordenador ou dos participantes
  - o Um novo processo retoma o processo usando a informação guardada em memória estável
  - o A memória estável é um espaço de memória que sobrevive a um crash do servidor
    - ▪ Pode ser implementada escrevendo o estado em disco (e forçando a escrita imediata e atómica do estado). A escrita é efectuada geralmente em dois passos: 1 – escreve log com operações a executar; 2 – executa operações do log na sua localização definitiva (em cada operação ou periodicamente)

Timeouts

- ✓ Coordenador
  - o Estado esperar (aguarda voto sim/não)
    - ▪ Coordenador não pode unilateralmente confirmar transacção
    - ▪ Coordenador pode unilateralmente abortar a transacção
  - o Estado confirmar (aguarda *haveCommitted*)
    - ▪ Coordenador não pode terminar a transacção (e remover informação sobre a mesma)
    - ▪ Coordenador pode reenviar resultado
- ✓ Participante
  - o Estado iniciar (aguarda *canCommit*)
    - ▪ Participante pode optar unilateralmente por abortar a Transacção
    - ▪ Estado preparado (aguarda *doCommit* ou *doAbort*)
    - ▪ Não pode progredir enquanto não obtiver a decisão
    - ▪ Transacção deve ficar activa e bloqueada até obter decisão
    - ▪ Contacta o servidor para obter a decisão
    - ▪ Contacta outro participante para obter decisão
    - ▪ Quando coordenador falha permanentemente, deve ser eleito um novo coordenador

Recuperação do coordenador

- ✓ Estado inicial e esperar: reenvia mensagem canCommit?
- ✓ Estado confirmar: se ainda não recebeu todas as mensagens haveCommitted reenvia mensagem doCommit
- ✓ Estado abortar: nada a fazer, mas pode existir necessidade de informar que transacção foi abortada

Recuperação de um participante

- ✓ Estado inicial: aborta unilateralmente a transacção
- ✓ Estado preparado: reenvia o seu voto para o coordenador

Problemas do protocolo two-phase commit

- ✓ O protocolo é bloqueante
  - o Obriga os participantes a esperar pela recuperação do coordenador (no estado preparado)
  - o Obriga o coordenador a esperar pela recuperação dos participantes (no estado confirmar)
- ✓ A recuperação não é independente
- ✓ Existem alternativas mais complexas: three-phase commit
  - o 3PC ainda pode bloquear

Transacções

- ✓ Uma transacção é uma sequência de operações que é executada respeitando as seguintes propriedades:
- ✓ Atomicidade: para um observador externo, ou todas as operações são executadas ou nenhuma é executada
- ✓ Consistência: um transacção deve, a partir de um estado inicial válido, atingir outro estado válido
- ✓ Isolamento: cada transacção é executada sem interferência, i.e., os estados intermédios de uma transacção não devem ser visto por nenhuma outra transacção
- ✓ Durabilidade: após uma transacção ter sido completada com sucesso, os seus efeitos são permanentes

2PC e transacções distribuídas

- ✓ O protocolo 2PC é usado para garantir a atomicidade no âmbito de sistemas de bases de dados distribuídas
- ✓ As outras propriedades devem ser garantidas usando outras técnicas
  - o Durabilidade: a memória em que os resultados são gravados deve ser estável
  - o Consistência: o modelo de programação deve garantir esta propriedade
  - o Isolamento:
    - Locks => perigo de existirem deadlocks
    - Timestamping order
    - Optimistic concurrency control => necessidade de validação global

VII - Introducao aos sistemas de designacao, de descoberta e de localizacao de servicos

Serviço de Nomes vs Serviço de Directório

Serviço de Nomes

Um serviço de nomes permite obter dados sobre uma entidade dado o seu nome.

Serviço de Directório

Um serviço de directório ou descoberta permite obter dados sobre as entidades que satisfazem uma dada descrição.

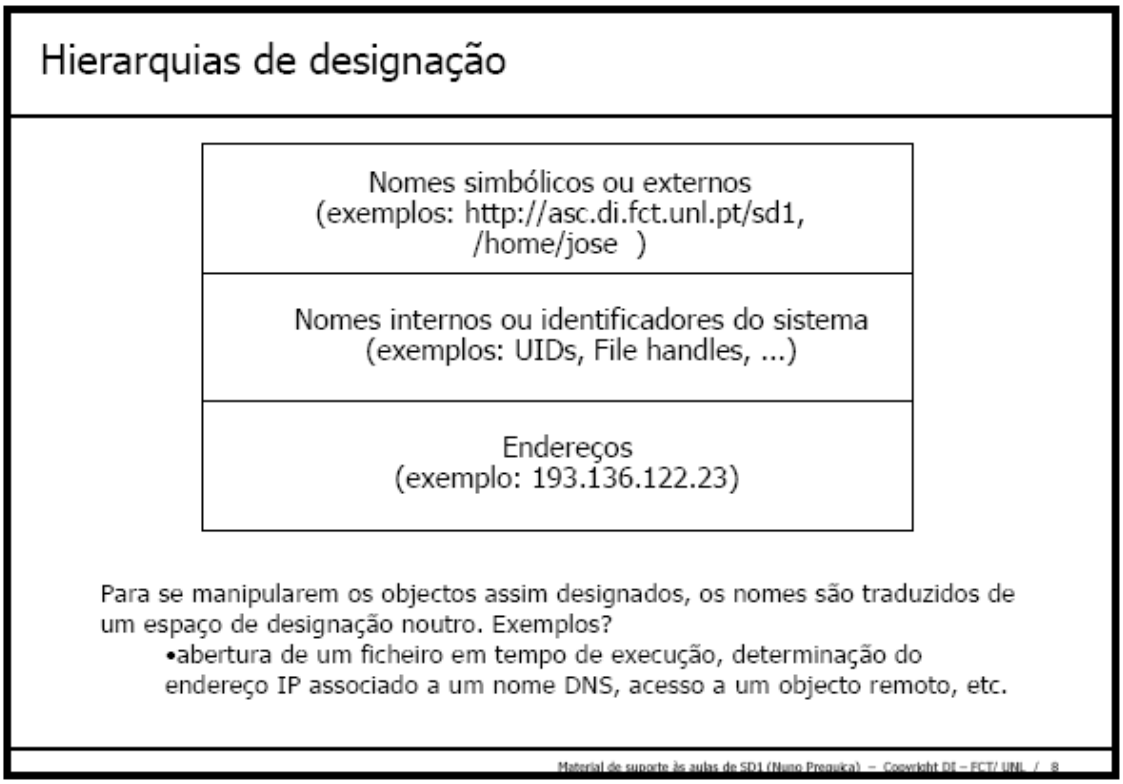
Nomes e identificadores

- ✓ Um **nome** permite designar uma entidade
  - o Num dado contexto, uma entidade pode ser designada por mais do que um nome.
- ✓ Um **identificador** permite identificar uma entidade
  - o Num dado contexto, uma entidade tem um e um so identificador.
  - o Um identificador unico permite identificar uma entidade de forma permanente
    - Se UID1=UID2, entao Ent1=Ent2 para todo o sempre
    - Se UID1≠UID2, entao Ent1 ≠ Ent2 para todo o sempre

- ✓ Um **nome simbolico**, textual, orientado aos utilizadores, ou externo, designa ou identifica entidades atraves de nomes mnemonicos e legiveis para os utilizadores.
  - o Ex: www.di.fct.unl.pt, /home/nmp/myfile

- ✓ Um **identificador** unico sistema (UID) designa sequencias de simbolos, geralmente sem significado mnemonico, que permitem identificar entidades de forma (quase) permanente, ao nivel interno do sistema distribuido.
  - o Ex: AF.65.8F.89.1B.23.FF.45.A5.89.8B (uid de um objecto remoto)

- ✓ Utiliza-se o termo endereco para designar formas especiais de designacao transitoria, volatil ou temporaria das entidades, geralmente associadas a localizacao das mesmas. Um **endereco** e um nome que da acesso imediato a uma entidade.
  - o Ex: 10.0.0.12



URLs, URNs, URIs

- ✓ Um **URL** (Uniform Resouce Locator) é um nome que indica a localização de um objecto. Um URL é basicamente um endereço.
  - o Ex: http://asc.di.fct.unl.pt/sd1

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

Nomes Globais vs Nomes Contextuais

Um sistema distribuido 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 oposicao a nomes relativos a outros contextos mais limitados que se dizem **nomes contextuais**.

Exemplos:

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

- ✓ Nomes contextuais:
  - o 10.200.0.2
  - o /home/joe
  - o C:\database\students

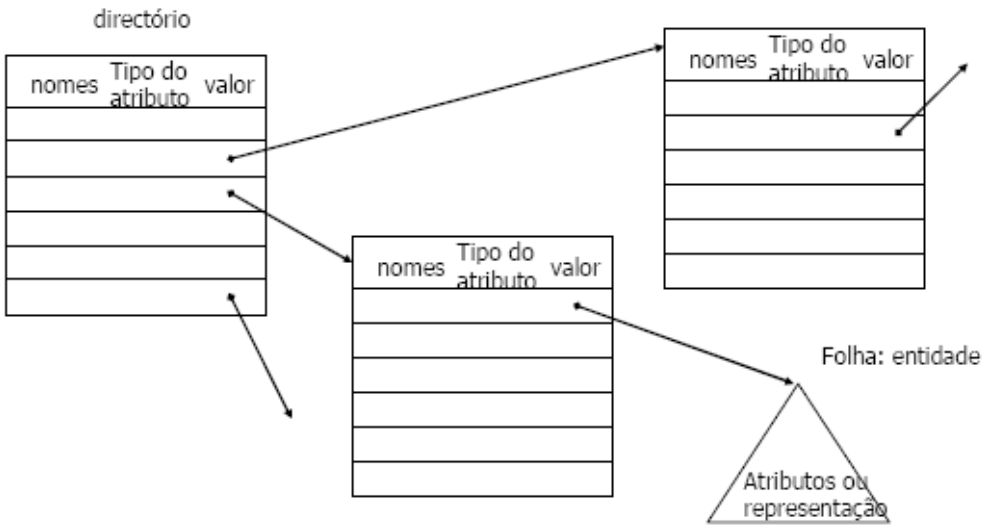
Organização do espaço de nomes

- ✓ Um domínio de nomeação é um espaço de nomes para o qual existe uma única autoridade administrativa para atribuir nomes.

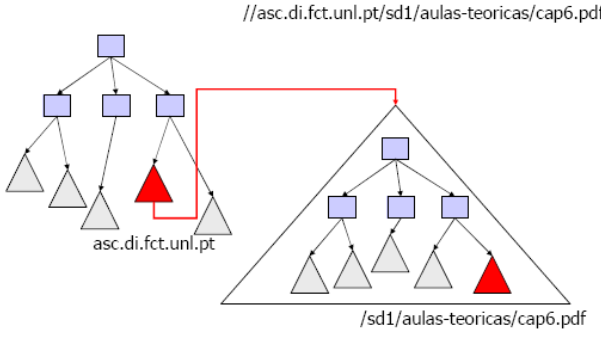
- ✓ Assim, estabelecem-se "ligações" entre os diferentes domínios e procedem-se a várias traduções de nomes até se chegar ao objecto. Exemplo:
  - o http://asc.di.fct.unl.pt/sd1/aulas-teoricas/cap6.pdf

- ✓ 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.

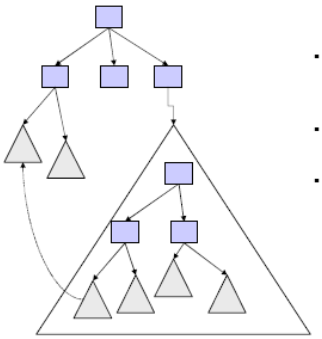
Organização conceptual de um espaço de nomes



Generalização a vários espaços



Conceitos importantes



- Contexto ou directório – lista de nomes e ligações a folhas ou a outros directórios
- Folhas – atributos da entidade designada pelo nome ou a sua representação directa
- *Mounts* – ligações entre espaços distintos
- Links simbólicos, redirecções ou aliases – uma folha que contém um nome (a ser resolvido)

Servico de designação ou de nomes

- ✓ Um servico de designacao ou servico de nomes e um servico que permite obter um conjunto de atributos de uma entidade dado o seu nome. A resolucao do nome e realizada a partir de um contexto de interpretacao do nome.

- ✓ Cada um desses nomes pode designar utilizadores, servidores, servicos, objectos remotos, ficheiros, etc.

- ✓ Um servico de nomes geralmente implementa uma base de dados (distribuida) que associa nomes aos atributos das entidades que estes designam.

- ✓ Existem duas operacoes importantes:
  - o **Lookup**: dado um nome devolve os atributos associados ao mesmo
  - o **Bind**: que associa um nome a um conjunto de atributos

- ✓ O servico de nomes pode ser um servico isolado e especifico ou estar integrado noutro servico. Exemplos?
  - o O DNS e um servico de nomes puro pois nao desempenha nenhum outro papel.
  - o Os servidores de ficheiros integram igualmente um servico de nomes dos ficheiros.

Resolução de Nomes

Resolução Iterativa do nome pelo cliente

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

- ✓ Características
  - o Servidor processa cada pedido rapidamente.
  - o Resolução começa em servidores diferente.

Resolução Recursiva do nome pelos servidores

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

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

Resolução por multicasting

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

- ✓ Características
  - o Apenas possível em ambientes que suportem multicast
  - o Todos os servidores de nomes vêm todos os pedidos

Mascaramento das falhas e caching

- ✓ Para mascarar falhas e fornecer elevada disponibilidade pode recorrer-se à replicação e caching de contextos.
  - o Ao colocar cópias da informação (réplicas) em servidors distinto é possível tolerar falhas dos servidores e da rede.
- ✓ 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

- ✓ Propriedades geralmente assumidas:
  - o os clientes suportam algum grau de incoerência.
  - o os valores registados no serviço evoluem lentamente.

- ✓ São frequentes as seguintes soluções:
  - o Replicação do tipo primário/secundários com propagação assíncrona das actualizações do primário para o secundário;
  - o 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".

Arquitectura DNS

- ✓ Base de dados particionada por múltiplos servidores
  - o Organização hierárquica dos servidores
  - o 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

  - o Qualquer servidor pode fazer cache de informação de outros servidores

- ✓ Clientes contactam servidores (pode ser dada lista de servidores)
  - o Protocolo tipicamente em UDP
  - o Cliente pode solicitar navegação recursiva ou iterativa. Servidor é livre de respeitar pedido.
  - o Clientes podem fazer caching dos resultados

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 subconjunto de atributos).
  - o Nestes sistemas, o nome é apenas um dos atributos.
  - o Exemplo: qual é o endereço IP da impressora a cores do edifício II ?

Serviços de nomes vs. serviços de directório

- ✓ Serviços de nomes
  - o Nomes mais simples

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

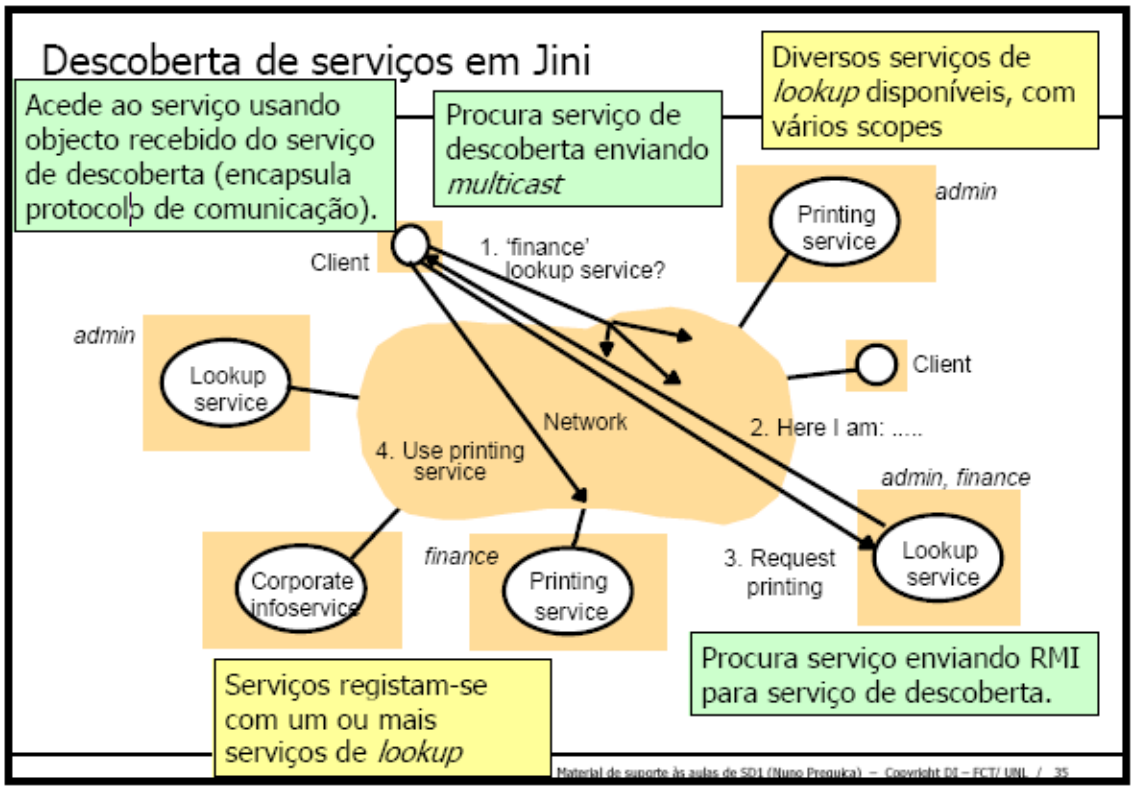
Serviços de descoberta e mobilidade

- ✓ Um serviço de descoberta é um serviço de directório que regista serviços disponíveis numa rede.

Exemplos: Um serviço de descoberta pode ser usado:

- o por um portátil para encontrar uma impressora, ou um projector de slides.
- o 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, ...



Serviços de descoberta: arquitecturas possíveis

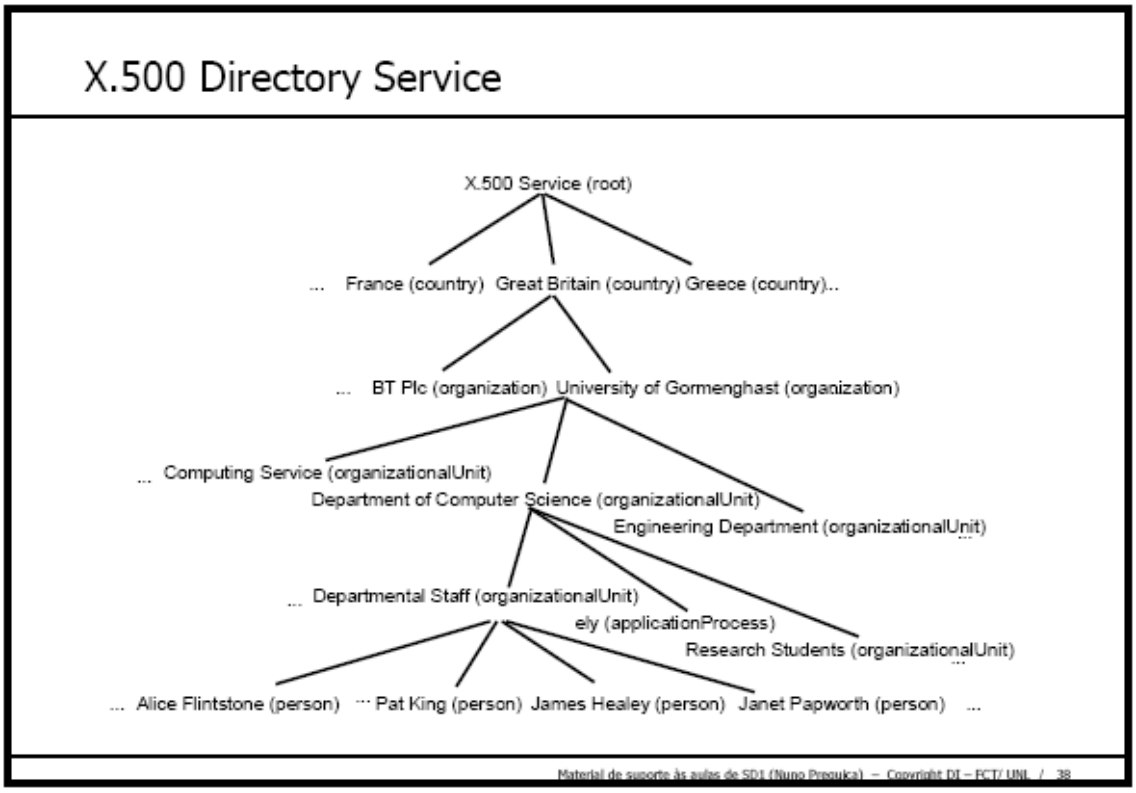
- ✓ Com servidor de directório. Como descobrir servidor?
  - o Utilização de multicast

- ✓ Sem servidor de directório
  - o Alternativa 1: Cliente envia multicast com pergunta. Servidores respondem directamente caso correspondam à descrição.
  - o Alternativa 2: Servidores enviam multicast periódico com a sua descrição. Clientes guardam informação para responder às perguntas locais.

- ✓ Os serviços registados nos servidores de descoberta tendem a ser voláteis. Como lidar com esta volatilidade?
  - o Arquitectura com servidor:
    - Utilização de leases: servidores devem renovar o seu registo periodicamente.
  - o 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)



- ✓ 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.
  - o DIB: entry 1, …, entry N
  - o Entry: atribute 1, …, atribute N
  - o Atribute: type, value
  - o Value: distinguished value, value

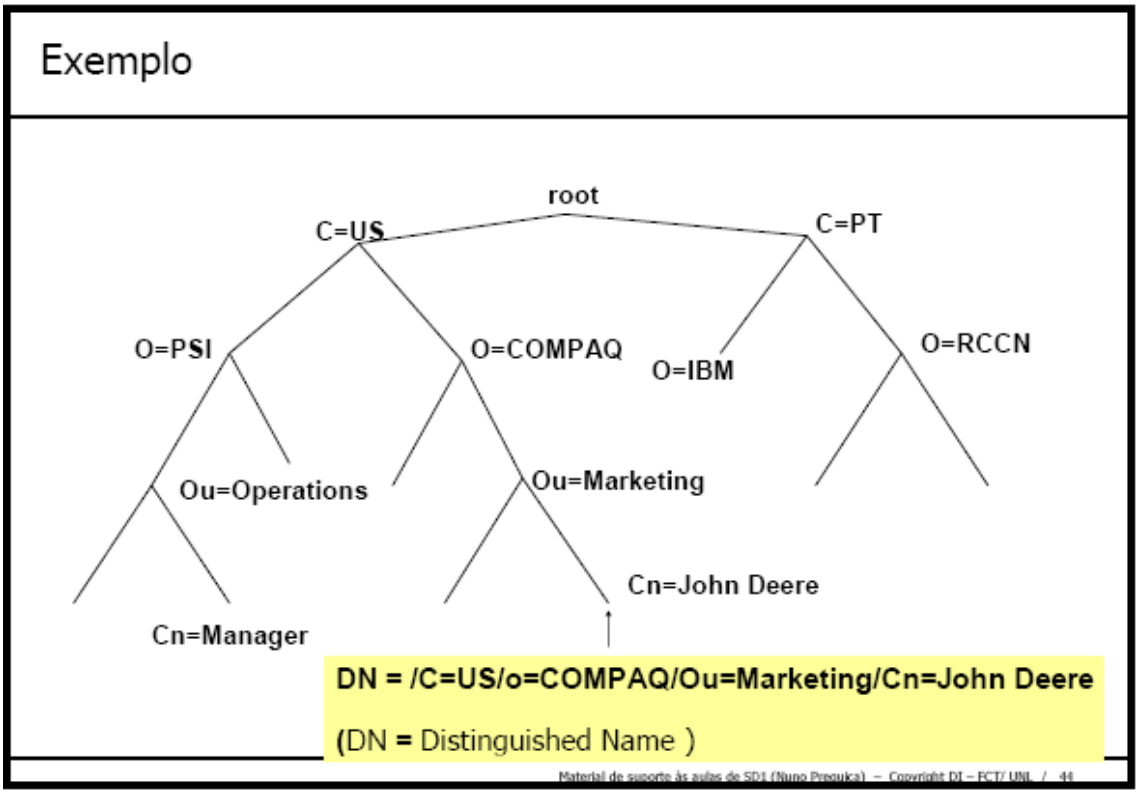
- ✓ Um dos atributos essenciais de uma entry é o atributo *objectClass* que é um atributo obrigatório que define o tipo da entry.

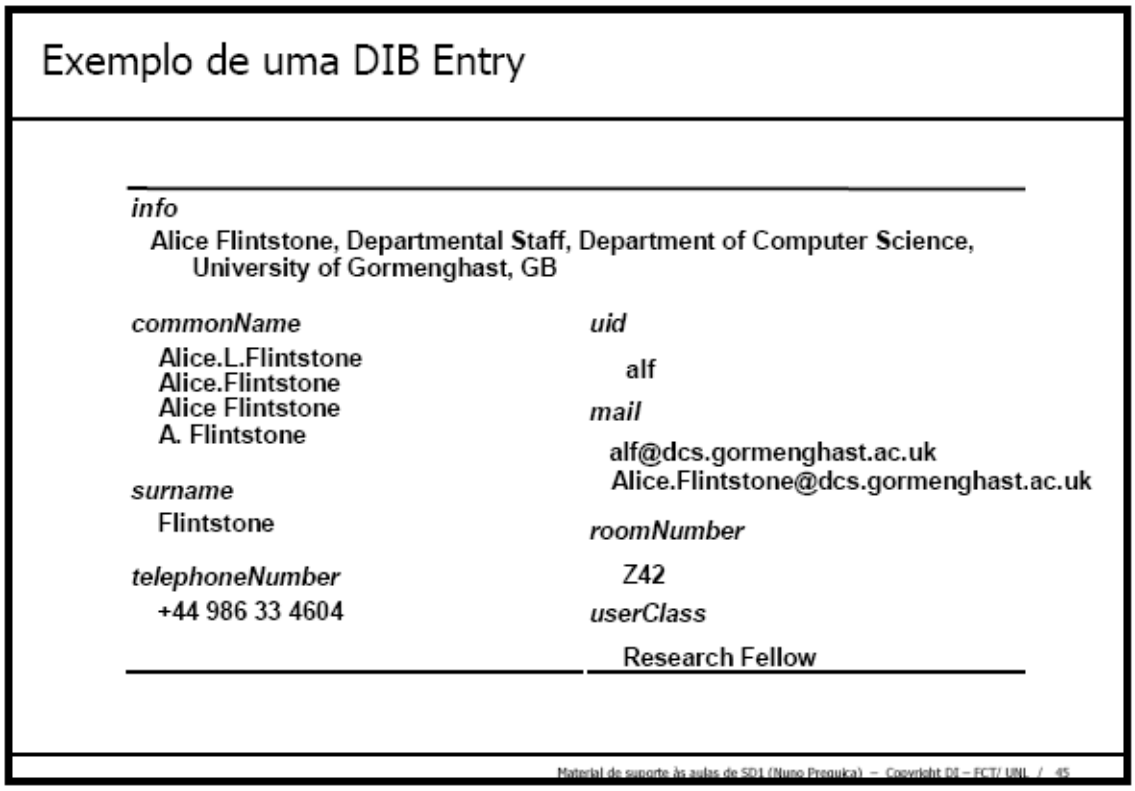
Tipos dos atributos

- ✓ 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.





Modelos de segurança

- ✓ Permite vários esquema de controlo de acessos
  - o Nenhuma
  - o Simples (password)
  - o Strong (Public Key Encryption) - baseada nos certificados X.509

- ✓ Suporta também ACLs para controlo dos acessos e modificações

Localização de entidades móveis

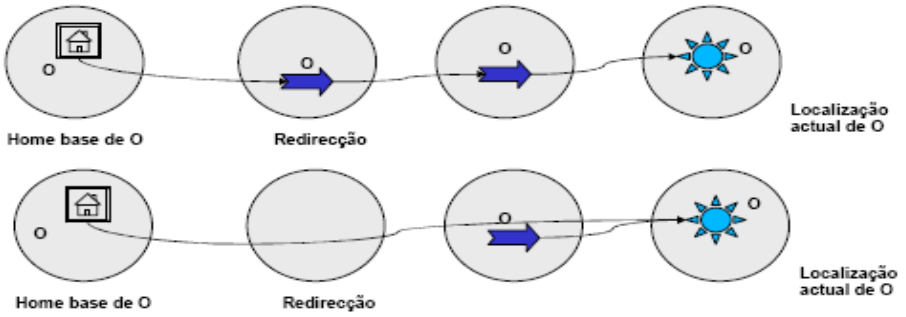
- ✓ Num sistema móvel, as entidades mudam frequentemente de localização. Neste caso, localizar estas entidades pode ser um problemas delicado.

- ✓ Nestes sistemas é necessário dispor de uma identificação global das entidades (um UID como por exemplo um número de telemóvel) e de mecanismos de localização eficazes das mesmas.

Localização de objectos por UIDs

- ✓ As técnicas mais comuns são as seguintes:
  - o **difusão** (onde está o objecto com um UID dado ?) que é uma técnica simples e universal mas tem problemas de escala;
  - o **cadeias de redirecção** (o objecto com este UID migrou para o site X) que é uma técnica que exige conhecer um ponto de partida (home base) para a localização do objecto;
  - o **serviço de localização**: uma espécie de base de dados hierárquica de localização de objectos.

Cadeias de redirecção



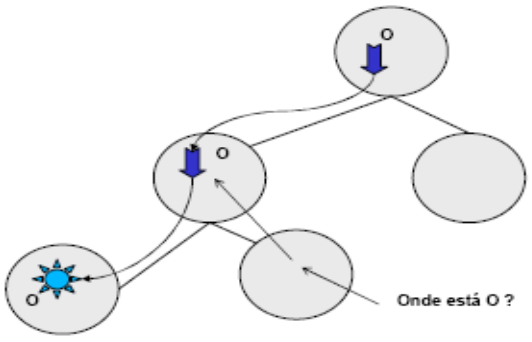
Quando O migra, é substituído por uma redirecção (por exemplo um *proxy*).

Pode-se optimizar obrigando O a avisar a *home base* de que migrou e ainda o seu *proxy* mais próximo.

Tornar fiáveis essas alterações é bastante complexo. Uma alternativa é trabalhar por técnicas de "soft state" mas, em qualquer hipótese, temos problemas delicados com referências falsas.

Material de suporte às aulas de SD1 (Nuno Preguiça) – Copyright DI – FCT/ UNL / 53

Serviço de localização



Para localizar O, pergunta-se ao servidor da nossa célula, seja ela qual for.

Se um objecto migra, é necessário actualizar a sua localização e todos os apontadores até à raiz.

O mesmo acontece quando o objecto é destruído.

A raiz conhece a localização de todos os objectos.

Existe uma hierarquia de servidores de localização associados a domínios. Todos os servidores que estão no caminho da célula actual, até à raiz, conhecem o domínio ou a localização do objecto. É possível generalizar este serviço ao caso em que há réplicas de O. Em qualquer hipótese existem problemas delicados de escala e de consistência.

Material de suporte às aulas de SD1 (Nuno Preguiça) – Copyright DI – FCT/ UNL / 54

VIII - Introducao aos sistemas distribuidos de gestao de ficheiros

Sistemas de ficheiros distribuidos

- ✓ Um sistema de gestao de ficheiros distribuidos fornece um servico de acesso a ficheiros semelhante ao de um sistema de ficheiros normal, mas estendido a um conjunto de maquinas ligadas em rede. Normalmente permite:
  - o Partilha de ficheiros por varios utilizadores
  - o Melhor qualidade de servico do que a disponivel localmente
  - o Maior dimensao do espaco disponivel
  - o Melhor tolerancia a falhas
  - o Desempenho semelhante ou superior

Questoes envolvidas na distribuicao

- ✓ Modelo de acesso: dois extremos: acesso completamente remoto ou acesso completamente local (usando cache).
- ✓ Designacao: como designar os ficheiros remotos?
  - o Externamente: ao nivel das aplicacoes
  - o Internamente: no sistema
- ✓ Partilha de ficheiros: problemas relacionados com o controlo da concorrencia
- ✓ Consistencia: quais as garantias que os utilizador tem de observar uma visao coerente do estado dos ficheiro quando ha varias replicas ou caching?
- ✓ Tolerancia a falhas, replicacao, …
- ✓ Escala, mobilidade, …

Sistema NFS: objectivos

- ✓ Transparencia da distribuicao
- ✓ Uma aplicacao acede a um ficheiro remoto de forma identica a um ficheiro local
- ✓ Compatibilidade tanto quanto possivel com a semantica no caso centralizado
- ✓ Suporte de heterogeneidade
- ✓ Implementacoes de clientes e servidores NFS disponiveis para quase todos os sistemas operativos (Unix, Windows, Mac, etc.)

O protocolo NFS (simplificado)

- ✓ **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.
- ✓ **rename(dirfh, name, todirfh, toname) -> status**
  - Changes the name of file name in directory dirfh to toname in directory to tod.irfh
- ✓ **link(newdirfh, newname, dirfh, name) -> status**
  - Creates an entry newname in the directory newdirfh which refers to file name in the directory dirfh.
- ✓ **mkdir(dirfh, name, attr) -> newfh, attr**
  - Creates a new directory name with attributes attr and returns the new file handle and attributes.
- ✓ **rmdir(dirfh, name) -> status**
  - Removes the empty directory name from the parent directory dirfh.
  - Fails if the directory is not empty.
- ✓ **readdir(dirfh, cookie, count) -> entries**
  - Returns up to count bytes of directory entries from the directory dirfh. Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a cookie. The cookie is used in subsequent readdir calls to start reading from the following entry. If the value of cookie is 0, reads from the first entry in the directory.
- ✓ **symlink(newdirfh, newname, string) -> status**
  - Creates an entry newname in the directory newdirfh of type symbolic link with the value string. The server does not interpret the string but makes a symbolic link file to hold it.
- ✓ **readlink(fh) -> string**
  - Returns the string that is associated with the symbolic link file identified by fh.
- ✓ **statfs(fh) -> fsstats**
  - Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file fh.

Geracao de file-handles

- ✓ Um file-handle e qualquer identificador que permita ao servidor identificar univocamente de que ficheiro / directorio se trata.
  - o Poder-se-iam usar diferentes metodos para gerar file-handles.
- ✓ Para evitar estar a memorizar numa tabela os file-handles, uma estrategia frequente nos servidores Unix consiste em usar um file-handle constituido por:
  - o [file system number, file number (i-node), i-node generation number], endereco IP do servidor, porta do servidor
- ✓ Esta aproximacao assegura a persistencia e unicidade da associacao entre um *file-handle* e um ficheiro (porque a informacao usada esta guardada em disco e e unica)
- ✓ Um *file-handle* associado com o endereco IP e porto do servidor designa um ficheiro remoto de forma univoca

Seguranca no NFS

- ✓ Ate a versao 2 do protocolo, os problemas de seguranca/controlo de acessos foram sempre tratados de forma algo simplista
- ✓ Pressupoe-se que no conjunto da rede os pares UID/GID estao uniformizados (para este efeito usava-se tradicionalmente o NIS para replicar os ficheiros /etc/passwd e /etc/group por exemplo)
  - o O UID "root" nao e aceite e e automaticamente traduzido para "nobody"
- ✓ Dado que o servidor nao mantem estado, como implementar controlo de acessos?
  - o O servidor testa sempre as proteccoes (identificacao do utilizador e enviada com cada pedido)
- ✓ Um ataque e facil desde que se conheca um file-handle e o UID/GID que lhe da acesso.
- ✓ Ha versoes de NFS baseadas em secure RPCs ou em Kerberos que nao tem estes problemas

Modelos de Caching : informacao replicada

- ✓ Quase todos os sistemas de ficheiros distribuidos mantem um cache que pode ter diferentes caracteristicas
- ✓ Modelo caching por blocos: Guarda blocos de ficheiros (geralmentes da dimensao do bloco dos discos e com dimensao identica a usada na transferencia entre o cliente e o servidor).
- ✓ Modelo Caching de Ficheiros "inteiros": Guarda ficheiros completos na cache – sempre que e necessario aceder a um ficheiro, este e transferido para a maquina do cliente.
- ✓ Modelo Servico Remoto ou sem Cache: Cada vez que e necessario aceder a um ficheiro, o cliente invoca o servidor.

Acessos concorrentes distribuidos

- ✓ Quando o sistema de gestao de ficheiros e distribuido o problema do acesso partilhado e do controlo da concorrencia e mais dificil de resolver.
- ✓ Os metodos usados para lidar com este aspecto podem agrupar-se em "metodos pessimistas" ou "metodos optimistas".
- ✓ Os metodos pessimistas tentam esconder que existem varias copias e procuram emular o melhor possivel a semantica do sistema centralizado – a distribuicao e mais transparente.

- ✓ Estes sistemas poderiam descrever-se como “sistemas com equivalencia a uma so copia” (“one copy serializability”).
- ✓ Os metodos optimistas tentam nao esconder totalmente a replicacao e o “caching” em troca de um maior desempenho, adaptacao a escala e flexibilidade.
- ✓ No limite sao os unicos aplicaveis em sistemas moveis quando se pretende admitir acesso em escrita com desconexao.

Controlo de concorrencia pessimista com servidores *stateless*

- ✓ A semantica tradicional centralizada consiste em uma leitura ver sempre o resultado da ultima escrita (serializacao dos acessos).
- ✓ Quando o sistema de gestao de ficheiros e no essencial “stateless”, a semantica dum esquema de gestao das caches depende do que se faz nas operacoes de leitura e escrita:
- ✓ Associada às escritas está a propagacao das modificacoes para o servidor (e sua visualizacao por outros clientes).
- ✓ Associada às leituras está a verificacao da coerencia da cache em relacao ao servidor.
- ✓ A introducao de um sistema de locks exige um ou mais servidores de locks que sao necessariamente statelful visto que as operacoes sobre locks sao necessariamente nao idempotentes.

Gestao da cache com servidores *stateless*

- ✓ Escrita “Write-through”. A cada escrita no cliente corresponde uma (ou mais) escritas no servidor. Favorece a fiabilidade e a semelhanca com a semantica tradicional tipo “sistema equivalente a uma so copia” a custa do sacrificio do desempenho.
- ✓ Escrita “Delayed-Write”. As escritas sao inicialmente efectuadas na cache local e, posteriormente, enviadas para o servidor. Estes esquemas diminuem o trafego de rede mas diminuem a fiabilidade e consistencia. Porque?
- ✓ Quando o servidor e stateless e ignora os clientes, nao e possivel implementar a semantica “equivalencia a uma so copia”. Porque?
- ✓ Leituras. Para garantir a frescura da informacao cached, os clientes de servidores stateless tem de testar a validade da mesma:
  - o teste antes de cada acesso - aumenta a coerencia mas diminui a performance
  - o teste periodico - semantica sem garantias totais

Gestao da cache com servidores stateful

- ✓ Aproximacao controlada pelo servidor: o servidor (stateful) anota cada cliente de um ficheiro e o modo de acesso e gere um sistema de notificacoes de alteracoes, assim como mecanismos baseados em tokens e locks para ajudar os clientes a melhorarem o seu desempenho.
- ✓ Quando existe um sistema de locks distribuidos, o cliente que possui um lock sobre um ficheiro pode usar esse conhecimento para melhorar a gestao da sua cache.
- ✓ Se o cliente possui um lock de leitura partilhada, o ficheiro nao pode ser escrito por outros e portanto enquanto o lock for valido nao sera modificado – os testes de validade do ficheiro cached sao inuteis.
- ✓ Se o cliente possui um lock de escrita exclusiva, o ficheiro nao pode ser escrito por outros e basta fazer escritas periodicas para minorar problemas em caso de crash.

Gestao de cache no sistema CIFS: Opportunistic locks

- ✓ Neste esquema, introduzido no sistema CIFS (sucessor do SMB), existem varios tipos de locks:
- ✓ Opportunistic locks (oplocks) exclusivos que permitem ao cliente ter acesso exclusivo ao ficheiro e fazer caching arbitrario 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 arbitrario do mesmo. Um oplock pode ser retirado ao cliente pelo servidor.
- ✓ Mandatory locks que permitem acessos exclusivos e caching e que nao podem ser retirados pelo servidor

Gestao dos oplocks

1. O primeiro cliente obtem:
   - 1.1 Oplock exclusivo caso pretenda escrever o ficheiro
   - 1.2 Oplock partilhado caso pretenda apenas ler o ficheiro
2. Quando surgem novos clientes de leitura:
   - 2.1 Existe um cliente com oplock exclusivo
     - a. Cliente perde lock e deve enviar as modifcacoes efctuadas
     - b. Ambos os clientes ficam sem oplocks (e portanto sem poder fazer cache)
   - 2.2 Existe um cliente com oplock partilhado
     - a. Ambos os clientes ficam com oplock partilhado
3. Quando surge um novo cliente de escrita
   - 3.1 Existe um cliente com oplock exclusivo
     - a. Cliente perde lock e deve enviar as modifcacoes efctuadas
     - b. Ambos os clientes ficam sem oplocks (e portanto sem poder fazer cache)
   - 3.2 Existe um (ou mais) clientes com oplock partilhado
     - a. Clientes perdem o oplock
     - b. Todos os clientes ficam sem oplocks (e portanto sem poder fazer cache)

- ✓ Quando um ficheiro e aberto para leitura/escrita tenta-se prolongar concorrencia, assumindo que o cliente e de leitura ate a primeira escrita – nesse momento, efectua-se o procedimento de adicao de um novo cliente.
- ✓ Se um oplock e quebrado, deixa de haver caching do ficheiro.
- ✓ Os mecanismos baseados em Oplocks (ou tokens) exigem a participacao de um servidor stateful que funciona como o gestor centralizado do controlo de acessos.
- ✓ Nestes esquemas e necessario detectar e tratar as falhas dos clientes, dos servidores ou das comunicacoes. Cada token tem uma validade limitada de 2 minutos por exemplo.
- ✓ Se um cliente nao revalida um token ou nao responde a uma mensagem de revogacao do token, o servidor assume que o cliente perdeu o token. Se o cliente nao obtem resposta a revalidacao do token, tambem conclui que o perdeu.
- ✓ Trata-se de um esquema de “leases” que permite, sem relogio centralizado, que o cliente e o servidor acabem por ficar de acordo quando existe uma anomalia. O risco maximo e corrido pelo cliente e esta limitado ao periodo de validade da “lease”.
- ✓ Revalidar uma “lease” e uma operacao idempotente que pode ser implementada por um protocolo com a semantica uma ou mais vezes.

Solução "Callback promise"

- ✓ Introduzido no AFS e usado tambem no Coda.
- ✓ Neste sistema os clientes fazem caching de ficheiros inteiros, sendo o ficheiro a unidade de transferencia entre o cliente e o servidor. Quando um cliente obtem um copia do ficheiro, o servidor promete informar o cliente de qualquer modificacao efectuada ao ficheiro – a esta promessa chama-se uma "callback promise".
- ✓ Desde que o cliente tenha uma "callback promise" valida, assume que a sua copia do ficheiro esta actualizada. Neste caso, um ficheiro e aberto no cliente sem nenhuma comunicacao com o servidor.
- ✓ Quando um programa no cliente fecha um ficheiro que acabou de modificar, o cliente AFS/Coda envia as modificacoes para o servidor. Nessa altura, o servidor notifica todos os clientes com "callback promises" validas.
- ✓ Um cliente, ao receber a notificacao, anula a sua "callback promise" e, se o ficheiro estiver aberto, obtem uma nova copia do ficheiro.
- ✓ Com este esquema todos as escritas feitas no cliente sao temporarias ate ao fecho do ficheiro. Desta forma, a unica semantica que pode ser implementada e uma semantica dita "semantica de sessao". Com esta semantica, uma leitura le o resultado das escritas feitas depois do ultimo close do ficheiro.
- ✓ Quando um cliente e inicializado, ele tem de revalidar as "calback promises" que tem.

CODA

Ver acetatos 8,9 e 10 do mesmo capítulo