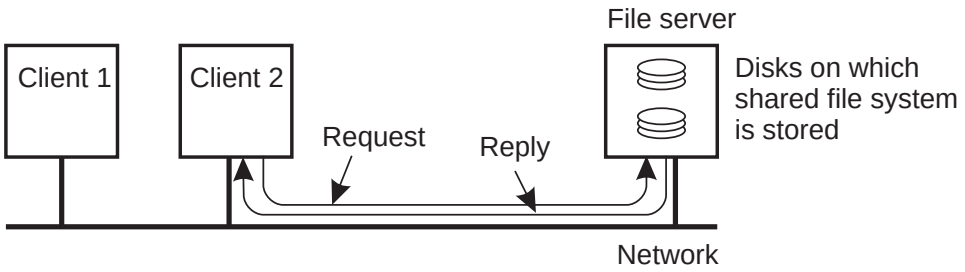
Clientes e Servidores

September 29, 2009

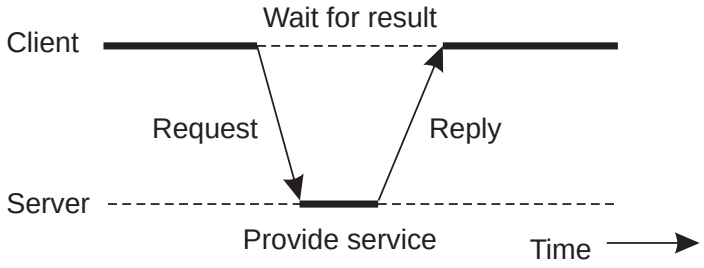
Sumário

Clientes e Servidores

- A maioria das aplicações distribuídas têm uma arquitectura do tipo *cliente-servidor*:



- Usaremos os termos *cliente* e *servidor* num sentido mais lato:



- Um servidor pode ser cliente (de outro serviço).

Localização do Servidor/Objectos

Problema: como é que um cliente sabe onde se encontra o servidor?

Solução: não há uma, mas várias respostas:

- *hard coded*, raramente;
- argumentos do programa: mais flexível, mas ...;
- ficheiro de configuração;
- usando *broadcast* ou *multicast*;
- serviço de localização/nomes (a abordar nas próximas aulas)
  - local, tipo *portmapper* ou *rmiregistry*;
  - *global*.

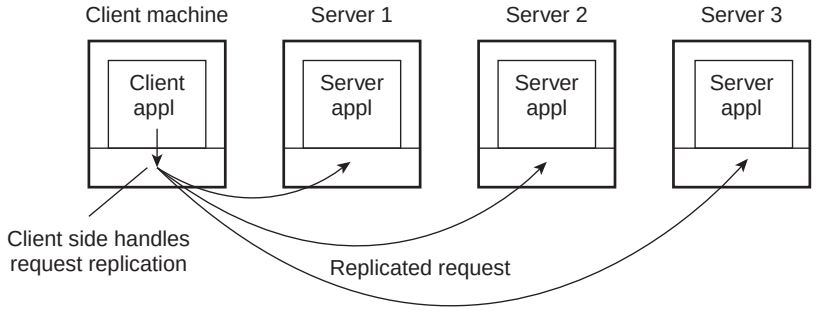
Transparência da Distribuição

Problema: Muitas facetas de transparência de distribuição podem ser conseguidas através de *stubs* do lado do cliente.

Acesso p. ex. usando RPC (próxima aula);

Localização p. ex. usando *multicast*;

Replicação p. ex. invocando operações em múltiplas réplicas;



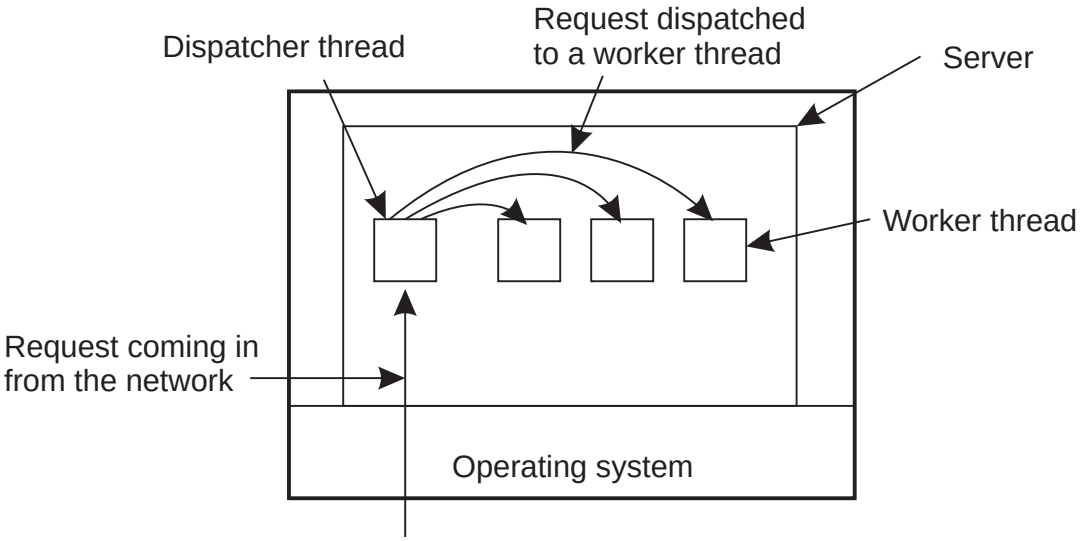
Avarias p. ex. mascarando avarias do servidor e da comunicação.

Concorrência

- Concorrência pode ser usada por razões de vária ordem:
  - Desempenho (+ nos servidores);
  - Usabilidade (+ nos clientes).
- Em qualquer dos casos, pretende-se esconder a latência de operações de E/S.
- Exemplo de cliente: *browser* da *Web*.
  - Uma página da *Web* pode consistir em vários *objectos*.
  - Um *browser* pode apresentar alguns objectos, enquanto outros ainda estão a ser transmitidos.
  - A ideia é realizar a comunicação e o processamento simultaneamente.
- Em clientes, a concorrência é conseguida normalmente por recurso a *threads* (por oposição a processos).

Concorrência em Servidores (1/2)

- Em servidores a sobreposição do processamento à comunicação permite o atendimento concorrente de pedidos.
- Por exemplo, no caso dum servidor de Web:



Concorrência em Servidores (2/2)

- Servidores podem ser:

  Concorrentes se são capazes de processar múltiplos pedidos simultaneamente;

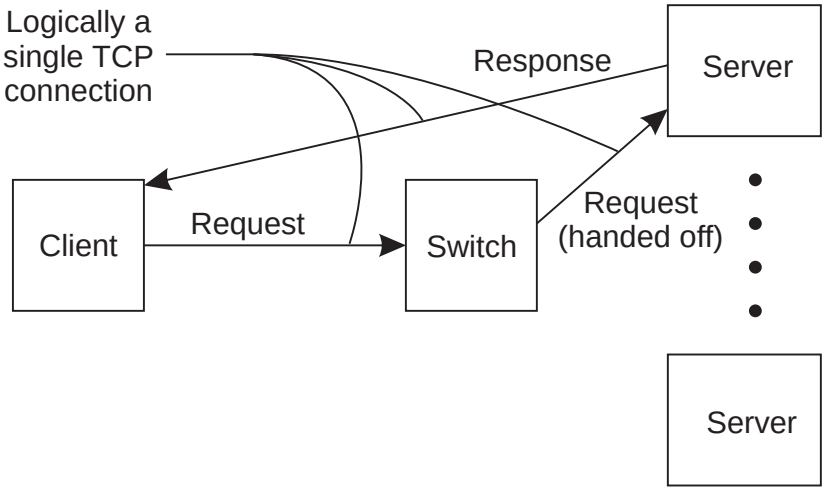
  Iterativos se processam os pedidos sequencialmente.
- Um servidor pode usar diferentes modelos:

Modelo	Paral.	Bloqueio em E/S	Progr.
n threads	sim	OK	*races*
1 *thread*	não	OK	fácil
Máquina de Estados	sim	Não	*event-driven*

- Para tirar partido de multiprocessadores só usando vários *threads* (ou processos).

Clusters de Servidores

- Quando o número de clientes é extremamente elevado pode recorrer-se a *server clusters/server farms*.
- Uma estratégia simples é fazer o encaminhamento dos pedidos ao nível de TCP



- Um problema chave é distribuir a carga pelos diferentes servidores.
  - *Round-robin* é talvez a aproximação mais simples.
  - Soluções ao nível da aplicação são também possíveis.

Servidores e Estado (1/4)

Problema a execução de certas tarefas por cada pedido pode impôr uma carga significativa ao servidor.

Solução o servidor pode manter alguma informação (*estado*) sobre interacções em curso (*sessões*):

- o tamanho;
- o tempo de processamento;

de cada mensagem são potencialmente menores.

- Por exemplo, num sistema de ficheiros distribuído, pode evitar-se abrir e fechar um ficheiro para executar uma operação remota de leitura/escrita.
  - Por exemplo, o servidor pode manter uma *cache* para cada cliente.

Servidores e Estado (2/4)

- A manutenção de estado por um servidor levanta problemas:
  - de consistência;
  - de gestão de recursos;

  na presença de avarias nos clientes e/ou servidores.
- Perda do estado quando o servidor avaria:
  - pedidos do cliente podem ser ignorados ou rejeitados pelo servidor após reinicialização: cliente terá que reiniciar.
  - incorrecta interpretação de pedidos enviados pelos clientes antes da avaria:
    - reutilização do porto duma conexão TCP.
- Manutenção de estado pelo servidor quando o cliente avaria:
  - pode conduzir à deplecção total dos recursos;
  - pode conduzir a uma incorrecta interpretação de pedidos posteriores por outros clientes.

Servidores e Estado (3/4)

- A não manutenção de estado no servidor não resolve os problemas resultantes de avarias:
  - a duplicação de mensagens pode conduzir à execução dum pedido múltiplas vezes:
    - os pedidos têm que ser *idempotentes*, se o protocolo de transporte usado não garantir a não duplicação de pacotes;
    - os pormenores são um bocado mais complicados (TCP).

Servidores e Estado (4/4)

- **Obs.-** A questão do estado dum servidor é uma questão relacionada com o protocolo:
  - Para que um servidor possa não manter estado, cada mensagem do protocolo tem que conter a informação necessária para o processamento do pedido correspondente.
  - Inversamente, para que seja possível manter estado, as mensagens deverão conter informação que permita relacioná-las entre elas.
- Por exemplo, a Netscape teve que acrescentar um campo ao *header* HTTP, para os *cookies*:
  - O protocolo HTTP é essencialmente *stateless*.
  - *Cookies* são um mecanismo que permite que um serviço *mantenha* estado sobre um cliente.
    - São os clientes que guardam os *cookies*.
    - Na *Web cookies* são usados normalmente em associação com estado do lado do servidor.

Identificação do Cliente

Em servidores que mantêm estado

- Usar o *ponto de acesso*, i.e. a extremidade do canal.
  - Por exemplo, o endereço IP e o porto usados pelo cliente.
  - Problema: pode não ser válido em mais do que uma sessão de transporte:
    - Se a conexão for interrompida e depois reestabelecida, o ponto de acesso pode ser diferente.
- Usar um *handle*, i.e. um “nome”, independente da sessão da camada de transporte. Exemplo:
  - *cookies* de HTTP.

Avarias

Problema: Dois importantes:

- componentes duma aplicação distribuída podem falhar, enquanto outros continuam a funcionar normalmente;
- na Internet não é possível distinguir entre problemas na rede ou problemas nos *hosts*.

Solução: depende da semântica da aplicação.

Crashes do Cliente

Problema: recursos reservados pelo servidor para o cliente continuam indisponíveis para outros fins:

- *sockets*, no caso de comunicação baseada em conexão;
- estado, no caso caso mais genérico de servidores com estado;
- recursos específicos da aplicação.

Solução *leases* (i.e. mais temporizadores):

- o servidor atribui recursos a um cliente apenas durante um intervalo de tempo finito: ao fim desse tempo, se o cliente não renovar o *lease* o recurso é-lhe removido.

Crashes do Servidor

Problema I: o servidor (pode) perder o estado:

- possibilidade de aceitar mensagens duplicadas (após *reboot*);

Problema II: como determinar se um pedido foi executado?

- Estão numa Caixa Multibanco. Introduzem o vosso PIN. Seleccionam levantar 50 Euros. De repente, a caixa recusa-se a dar-vos o dinheiro alegando problemas nas comunicações. (Ou terá sido no servidor?) Vocês não recebem o dinheiro. Mas será que ele foi debitado na vossa conta?

Adaptação ao Canal de Comunicação

Ordem a aplicação terá que reordenar as mensagens (uso dum número de sequência), se tal fôr importante.

Fiabilidade necessidade de usar *temporizadores* (*timers*) para recuperar da perda de mensagens (*duplicados*).

Controlo de fluxo: pode conduzir à perda de mensagens por falta de recursos.

Abstracção do canal: a aplicação poderá ter que identificar o limite das mensagens, ou fragmentá-las. Pode ter que reconstruir mensagens na recepção.

Fiabilidade do Canal

Problema: recuperar da perda dum pacote.

Solução: usar temporizadores (*timers*)

- os atrasos numa rede podem ter uma grande variância;
- o servidor pode estar sobrecarregado.

Mais Problemas:

- duplicação de mensagens;
- o servidor pode não ter recursos (*controlo de fluxo*).

Duplicação de Mensagens

Problema: um pedido pode ser executado mais do que uma vez.

Solução: 2 alternativas:

- usar *identificadores* nas mensagens – servidor com estado;
- definir operações *idempotentes*, i.e., que produzem o mesmo resultado se aplicadas 1 ou mais vezes.

Segurança (*Security*)

Problema: servidores executam com privilégios que a generalidade dos seus clientes não possui;

Solução: servidores têm que fazer:

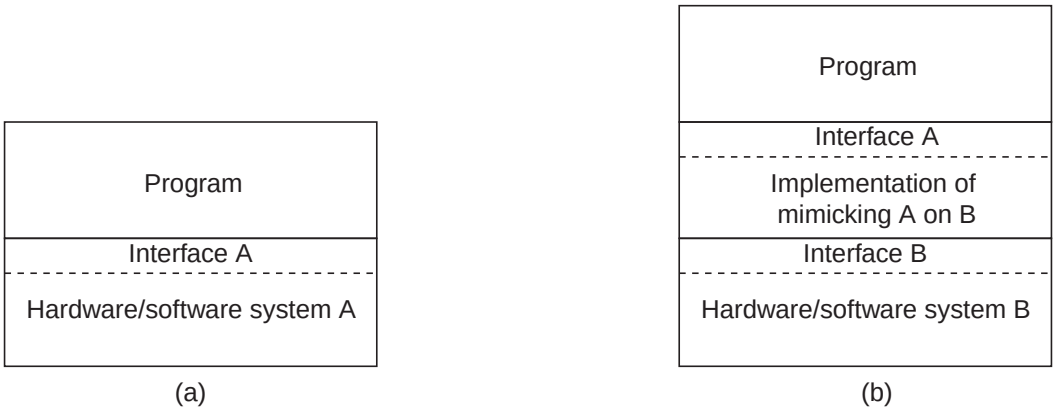
autenticação: garantir que o cliente é quem diz ser;

controlo de acesso: garantir que um cliente não acede a serviços ou informação para os quais não tem direitos;

- Outro requisito relacionado é a *confidencialidade* dos dados:
  - cifrar a informação transmitida através da rede.
- A *importação* código cria outro tipo de problemas.

Virtualização de Recursos (1/2)

Ideia

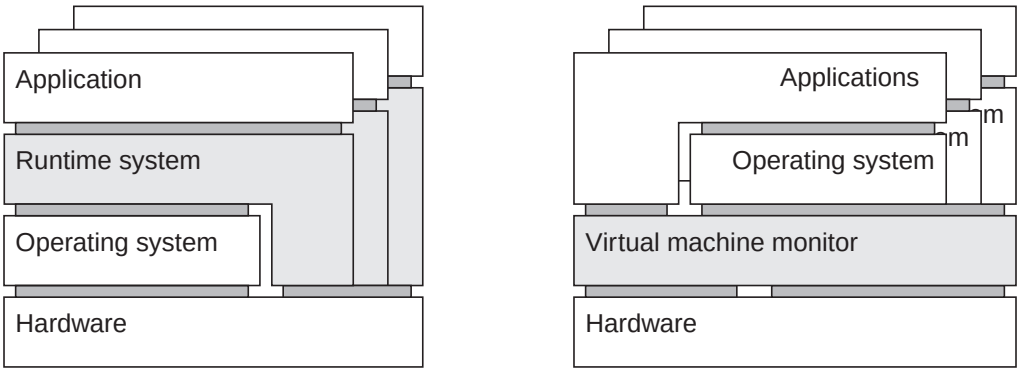


Essencialmente, a virtualização permite substituir uma interface de modo a imitar o comportamento dum outro sistema.

Objectivo Permitir que *legacy SW* desenvolvido para uma geração de *mainframes* executasse em plataformas posteriores (IBM).

Virtualização de Recursos (2/3)

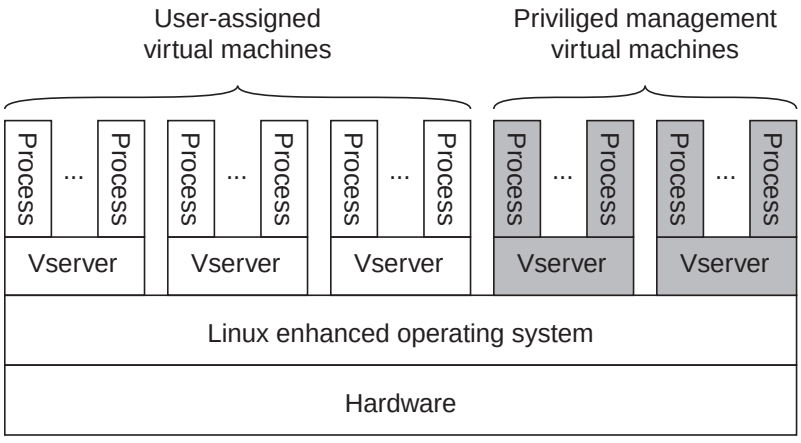
Implementações



Process Virtual Machine (a) Cada máquina virtual fornece um conjunto de instruções abstracto usado na execução de aplicações. Exemplo: *Java Virtual Machine*.

Virtual Machine Monitor (b) Cada máquina virtual fornece um conjunto de instruções abstracto ao nível do *hardware* e permite a execução concorrente e independente de múltiplos SOs. Exemplo: VMware, Xen, VirtualPC.

Virtualização de Recursos (3/3)



- O uso de máquinas virtuais em sistemas distribuídos tem essencialmente duas vantagens:
  1. Aumenta a segurança e a fiabilidade;
  2. Facilita a gestão.

Leitura Adicional

- Capítulo 3 de Tanenbaum e van Steen, *Distributed Systems, 2nd Ed.*
  - Subsecção 3.1.2 *Threads in Distributed Systems*, restante material da Secção 3.1 assume-se conhecido de Sistemas Operativos.
  - Subsecção 3.3.2 *Client-Side Software for Distribution Transparency*, restante material da Secção 3.3 não é essencial.
  - Secção 3.4 *Servers*, embora apenas algumas partes das subsecções 3.4.2 e 3.4.3
  - Secção 3.2 *Virtualization*