

Alguns aspectos do sistema distribuído devem ser transparentes para o utilizador. Apresente as diversas formas de transparência desejáveis num sistema distribuído.

preocupações em SD: transparência

• manter alguns aspectos da distribuição invisíveis para o programador ou utilizador, para que o sistema seja visto como um todo para que o utilizador ou programador possa focar a sua atenção na sua aplicação (cliente do sistema) sem depender de aspectos específicos da distribuição

• modularidade

acesso: permitir o acesso a recursos locais e remotos com operações idênticas

localização: permitir a utilização de recursos sem o conhecimento da sua localização exacta

concorrência: permitir a execução simultânea de vários processos com recursos partilhados sem que surjam interferências entre eles

replicação: utilização de múltiplas instâncias de recursos para aumentar a fiabilidade e performance, mas sem que o utilizador ou aplicações cliente tenham conhecimento das réplicas

falhas: permitir o tratamento de falhas, para que utilizadores e aplicações completem a sua tarefa, independentemente da ocorrência de um problema de hardware ou software

mobilidade: permitir a mobilidade de recursos e clientes dentro de um sistema sem afectar as operações de utilizadores e programas

performance: permitir o ajuste ou reconfiguração do sistema para aumentar o desempenho à medida que as solicitações (carga) variam, de modo transparente para o utilizador

escala: permitir a expansão das componentes do sistema sem alterar a estrutura do mesmo ou os algoritmos das aplicações

Descreva o processo designado por Marshalling e justifique a sua utilização.

Marshalling/Unmarshalling

• Marshalling:

— Linearização de uma coleção de itens de dados estruturados

— Tradução dos dados em formato externo

• Unmarshalling:

— Tradução do formato externo para o local

— Restauração dos itens de dados de acordo com sua estrutura

O processo de empacotar um ou mais itens de dados em uma mensagem {buffer}, previamente transmitindo tal buffer de mensagem por um canal de comunicação. O processo de empacotamento não só coleta valores juntos que provavelmente são armazenados em locais de memória não-consecutivos como também converte dados de tipos diferentes em uma representação padrão que de acordo com o recipiente da mensagem. O processo de codificação em um formato específico ideal para os dados serem transportados através da rede chama-se de *marshalling de parâmetros*. As informações que o *stub* possui são um identificador do objeto remoto, um número de operação que descreve o método a ser chamado e os parâmetros depois do processo de *marshalling*. Então essas informações são enviadas para o servidor onde há um objeto *skeleton* que tira o significado da informação contida no pacote e passa essas informações ao objeto que vai executar o método remoto; o *skeleton* decodifica os parâmetros passados, usando o processo de *marshalling*, chama o método real que se encontra no servidor, captura o valor de retorno ou exceção à chamada no servidor, codifica através do *marshalling* esse valor e envia um pacote tendo o valor após o *marshalling* de volta para o *stub* no cliente.

Descreva as vantagens fornecidas pela camada Middleware e indique algumas abstracções que disponibiliza ao programador de sistemas distribuídos.

- ❖ Transparência face à localização
 - ❖ o processo cliente não sabe o endereço do servidor
- ❖ Independência dos protocolos de comunicação
 - ❖ o Request-Reply pode ser implementado sobre TCP ou UDP
- ❖ Independência do Hardware
 - ❖ standards para Representação Externa de Dados escondem diferenças
- ❖ Independência dos SO
 - ❖ as abstrações de nível superior fornecidas pelo Middleware não dependem do SO
- ❖ Utilização de diferentes linguagens de programação
 - ❖ CORBA: cliente numa linguagem pode invocar métodos em servidores escritos noutra linguagem – graças à CORBA IDL

Existem três tipos de medidas de tolerância a falhas desejáveis para a primitiva doOperation no protocolo Request-Reply.

(a) Enumere e explique em que consistem essas medidas.

garantias de *doOperation* no protocolo RR:

- ❖ reenvio do pedido
 - ❖ reenviar a mensagem com o pedido para o servidor até a resposta chegar ou se detectar que o servidor está com problemas
- ❖ filtragem de duplicados
 - ❖ decidir se o duplicado deve ser processado para reenvio ou ignorado
- ❖ retransmissão de resultados
 - ❖ através de um histórico de resultados para evitar uma nova execução da operação

Indique quais são usadas pelas semânticas de invocação Maybe, At-least-once e At-most-once.

medidas			semântica de invocação
reenvio do pedido	filtragem de duplicados	executar de novo ou retransmitir resultado do histórico	
Não	Não aplicável	Não aplicável	Maybe
Sim	Não	executar de novo	At-least-once
Sim	Sim	retransmitir do histórico	At-most-once

Sun RPC
Java RMI
CORBA

Das semânticas de invocação referidas, indique qual a mais apropriada quando estão envolvidas operações não idempotentes.

Possíveis Falhas:

- ♦ Maybe: omissão, crash
- ♦ At-least-once: crash, arbitrárias (retransmissão do pedido, reexecução)
- ♦ At-most-once: falhas evitam-se com os mecanismos de tolerância a falhas

Descreva o papel de Proxy, Dispatcher e Skeleton na abstracção RMI.

Software RMI

♦ Proxy

- ♦ torna a invocação remota transparente para o cliente
- ♦ marshalling de argumentos, unmarshalling de resultado da invocação
- ♦ existe um proxy para cada objecto remoto que um processo refere
- ♦ implementa os métodos da interface remota do objecto, mas cada método faz marshall da referência do objecto, *methodId*, e argumentos, aguardando a resposta para o unmarshall

♦ Dispatcher

- ♦ um para cada classe de objecto remoto, no servidor
- ♦ recebe a mensagem e pelo *methodId* selecciona o método apropriado no Skeleton

♦ Skeleton

- ♦ um por cada classe que representa um objecto remoto, no servidor
- ♦ implementa os métodos na interface remota, mas efectuando unmarshall a argumentos no pedido, invocando o método no objecto remoto (localmente) e devolvendo o marshall do resultado e eventual exceção na reposta ao proxy

A distribuição dos objectos requer cuidados no Garbage Collection, especialmente do lado do servidor.

(a) Explique resumidamente o algoritmo de GC no Object Model Distribuído.

Cooperação com o GC local:

- ♦ cada servidor mantém uma lista com o conjunto de processos com referências para os seus objectos
- ♦ quando um cliente cria um Proxy para um objecto, é adicionado ao conjunto de processos com referências para aquele objecto
- ♦ quando o GC do cliente detecta que o Proxy do objecto já não necessário/referido, envia uma mensagem ao servidor (*removeRef(O)*) e elimina o proxy. O servidor remove o processo da lista.
- ♦ Quando a lista estiver vazia, o GC do servidor recupera o espaço do objecto, excepto se existirem referências locais.

(b) Identifique eventuais vulnerabilidades (ou medidas de tolerância a falhas) do algoritmo face a falhas no cliente.

Java Distributed GC

- ♦ tolera falhas no cliente
 - ♦ o servidor (com objectos remotos) atribui um intervalo de tempo (*lease*) ao cliente
 - ♦ a contagem é válida até o tempo expirar ou o cliente pedir *removeRef()*
- ♦ o cliente é responsável por renovar periodicamente o seu *lease*
 - ♦ e assim ser contabilizado na lista de processos com referências para os objectos remotos

Sincronização de Relógios:

Algoritmo de Berkeley

para sincronização interna de um grupo de computadores

- ♦ uma máquina é escolhida para coordenar – *master*
- ♦ o *master* pede periodicamente uma leitura aos restantes (*slaves*)
- ♦ master estima a hora em cada slave, pela observação do tempo de viagem das mensagens e pelo valor recebido (como em Cristian)
- ♦ faz a média de todos os valores (incluindo o seu tempo)
- ♦ em vez de enviar o tempo actualizado aos slaves (o que estaria sujeito ao tempo de envio variável), o master envia a cada um o valor exacto que este deve usar para ajustar o seu relógio
- ♦ se o master falhar, outro será escolhido para assumir a sua função

Network Time Protocol (NTP)

protocolo para distribuir informação horária sobre a Internet

Funcionalidades:

- ♦ permite a sincronização precisa de clientes, que se encontram espalhados pela Internet, com UTC
- ♦ serviço fiável que resiste a perdas de conectividade
- ♦ escalável em termos de nº de clientes e nº de servidores
- ♦ protecção contra interferência (accidental ou maliciosa) na informação horária

Serviço suportado por uma rede de servidores na Internet:

- ♦ primários: ligados directamente a uma fonte de UTC
- ♦ secundários: sincronizados com a informação dos primários

Algoritmo de Cristian

- ♦ probabilístico: sincroniza se o tempo para a troca de mensagens (um par) cliente-servidor é suficientemente pequeno quando comparado com a precisão desejada
- ♦ servidor de tempo UTC S
- ♦ processo p envia pedido m_p e recebe um tempo t em m_t
- ♦ p regista o tempo de viagem de m_p e m_t , T
- ♦ estimativa do tempo em p: $t + T/2$
 - ♦ assume o mesmo tempo para as duas mensagens (!)
 - ♦ razoável: excepto se enviadas por redes diferentes ou se ocorre um afunilamento repentino aquando da 2ª mensagem
- ♦ Opcão para melhorar a precisão:
 - ♦ informação horária prestada por um grupo de servidores sincronizados
 - ♦ cliente faz multicast do pedido para o grupo de servidores e usa a primeira resposta obtida

1. Alguns aspectos do sistema distribuído devem ser transparentes para o utilizador. Apresente as diversas formas de transparência desejáveis num sistema distribuído.

Uma das preocupações em Sistema Distribuído é a transparência que mantém alguns aspectos da distribuição invisíveis para o utilizador. As diversas formas de transparência desejáveis num Sistema Distribuído são o acesso que permite o acesso a recursos locais e remotos com operações idênticas, a localização que permite a utilização de recursos sem o conhecimento da sua localização exacta, a concorrência que permite a execução simultânea de vários processos com recursos partilhados sem que haja interferências entre eles, a replicação que utiliza múltiplas instâncias de recursos para aumentar a fiabilidade e a performance, mas sem que o utilizador ou as aplicações cliente tenham conhecimento das réplicas, as falhas que permite o tratamento de falhas, para que o utilizador e as aplicações completem a sua tarefa, independentemente da ocorrência de um problema de hardware ou software, a mobilidade que permite a mobilidade de recursos e clientes dentro de um sistema sem afectar as operações de utilizadores e programas, a performance que permite a reconfiguração do sistema para aumentar o desempenho à medida que as cargas variam e a escala que permite a expansão das componentes do sistema sem alterar a estrutura do mesmo.

2. Distinga Modelo Cliente-Servidor de Modelo Peer Processes.

No modelo Cliente-Servidor, os clientes invocam um servidor mas o servidor também pode assumir o papel de cliente para ligar-se a outro servidor enquanto no modelo Peer Processes é baseado em peer processes (processos que cooperam e comunicam de forma simétrica para realizar uma tarefa) em que os processos desempenham papéis idênticos, sem uma separação prévia entre clientes e servidores. Pontualmente, alguns destes processos podem comportar-se como clientes ou como servidores.

3. Descreva o comportamento das operações send e receive (na transmissão de uma mensagem), relativamente ao bloqueio, em cenários de comunicação síncrona e assíncrona.

Na comunicação síncrona os processos emissor e receptor sincronizam-se a cada mensagem, onde o send e o receive são operações bloqueantes. O emissor fica parado no send até que o receive seja efectuado, e ao efectuar um receive, o receptor fica bloqueado até a mensagem chegar enquanto que na comunicação assíncrona não há sincronização, ou seja, o send não é bloqueante visto que o emissor prossegue assim que a mensagem passa ao buffer local de saída e o receive pode ser bloqueante ou não, visto que o processo prossegue com um buffer que será preenchido em background, havendo uma notificação quando isso acontecer.

4. Explique a diferença entre sistemas síncronos e assíncronos.

Nos sistemas síncronos existem limites para o tempo de execução de cada passo de um processo, para o tempo até à recepção de uma mensagem enviada e para o clock drift rate em cada máquina enquanto nos sistemas assíncronos não há limite definido ou garantias para a velocidade de execução de um processo, para o tempo de transmissão de uma mensagem, esta pode ter atraso (delay) nem para o clock drift rate, visto que a taxa deste é arbitrária.

5. Descreva o processo designado por Marshalling e justifique a sua utilização.

O processo de Marshalling consiste na tradução das estruturas e tipos primitivos para uma representação externa de dados (formato usado para a representação de estruturas e tipos primitivos) adequada para a sua transmissão.

O processo de Marshalling é utilizado para que uma estrutura de dados possa ser usada em RPC ou RMI, ou seja, para que possa ser representado de modo flattened e os tipos primitivos num formato acordado.

7. Indique uma diferença entre Marshalling em CORBA e o usado em Java RMI.

No Marshalling em CORBA assume-se que o emissor e receptor conhecem os tipos de cada elemento da mensagem, por isso o tipo não é passado (apenas o valor) enquanto que em Java

RMI, a aplicação que recebe a mensagem pode não conhecer o tipo de dados. Então, a representação serializada inclui informação sobre a classe do objecto

8. Descreva as vantagens fornecidas pela camada Middleware e indique algumas abstracções que disponibiliza ao programador de sistemas distribuídos.

As vantagens fornecidas pela camada Middleware são:

- Transparência face à localização;
- Independência dos protocolos de comunicação;
- Independência do Hardware;
- Independência dos Sistemas Operativos;
- Utilização de diferentes linguagens de programação.

Algumas abstracções que estão disponíveis para o programador são a abstracção RPC e a abstracção RMI. A RPC descreve os procedimentos disponíveis e respectivos argumentos e não se podem passar pointers como argumentos e a RMI tem métodos de um objecto disponíveis para Invocação Remota e podem passar-se referências para objectos remotos.

9. Existem três tipos de medidas de tolerância a falhas desejáveis para a primitiva doOperation, no protocolo Request-Reply.

a) Enumere e explique em que consistem essas medidas.

As medidas de tolerância da primitiva doOperation no protocolo Request-Reply são o reenvio do pedido que reenvia a mensagem com o pedido para o servidor até a resposta chegar ou se detectar que o servidor está com problemas, a filtragem de duplicados que decide se o duplicado deve ser processado para reenvio ou ignorado e a retransmissão de resultados que através de um histórico de resultados evita uma nova execução da operação.

b) Indique quais são usadas pelas semânticas de invocação Maybe, At-least-once e At-most-once.

A Maybe não reenvia o pedido, a filtragem de duplicados e a retransmissão de resultados não são aplicáveis.

O At-least-once reenvia o pedido, não filtra os duplicados e executa de novo a retransmissão de resultados.

O At-most-once reenvia o pedido, filtra os duplicados e retransmite do histórico os resultados.

c) Das semânticas de invocação referidas, indique qual a mais apropriada quando estão envolvidas operações não idempotentes.

Quando estão envolvidas operações idempotentes, a semântica de invocação mais apropriada é o Maybe.

10. Imagine um sistema baseado em diferentes linguagens de programação e plataformas.

a) Indique um mecanismo para invocação remota de métodos adequado a este cenário.

Um mecanismo para invocação remota adequado a este cenário é o CORBA. Que permite que objectos distribuídos e implementados em diferentes linguagens de programação possam comunicar.

b) Ainda neste cenário, seria necessária uma Interface Definition Language? Porquê?

Neste cenário é necessário uma IDL pois requer uma IDL que forneça a notação para as interfaces que poderão ser usadas pelas diferentes aplicações.

11. Em Java RMI, que operações devem ser obrigatoriamente efectuadas pela aplicação Servidor, para que um cliente possa invocar métodos remotamente sobre o objecto remoto que presta um serviço?

As operações que devem ser obrigatoriamente efectuadas pela aplicação Servidor é ter as classes dispatcher e skeleton, ter as classes com implementação para todos os objectos remotos que podem ser a Servant ou Impl, tem-se de criar pelo menos um objecto remoto e inicializá-lo,

registar o objecto no binder e para evitar demoras, cada invocação remota é tratada numa nova thread.

12. Descreva o papel de Proxy, Dispatcher e Skeleton na abstracção RMI.

O papel do Proxy torna a invocação remota transparente para o cliente, faz o marshalling de argumentos e o unmarshalling do resultado da invocação, é único para cada objecto remoto que um processo referencia e implementa os métodos da interface remota do objecto, mas cada método faz marshall da referência do objecto, methodid, e argumentos, aguardando a resposta para o unmarshall.

O Dispatcher é único para cada classe de objecto remoto, no servidor. Recebe a mensagem e pelo methodId selecciona o método apropriado no Skeleton. O Skeleton é um por cada classe que representa um objecto remoto, no servidor e implementa os métodos na interface remota, mas efectuando unmarshall a argumentos no pedido, invocando o método no objecto remoto (localmente) e devolvendo o marshall do resultado e eventual excepção na resposta ao proxy.

13. Distinga os modos de sincronização interna e externa de relogios num sistema.

Relogios sincronizados de modo interno não estão necessariamente sincronizados de modo externo, ou seja, se cada nó de um sistema está sincronizado de modo externo (com a mesma fonte) com limite D, então esse sistema está internamente sincronizado com um limite 2D.

14. Descreva sucintamente o algoritmo de sincronização de Cristian.

O algoritmo de sincronização de Cristian sincroniza-se o tempo para a troca de mensagens cliente-servidor é suficientemente pequeno quando comparado com a precisão desejada e o servidor de tempo é UTC S.

O processo p envia pedido mr e recebe um tempo t em mt, p regista o tempo de viagem de mr e mt onde $T = T_{mr} + T_{mt}$. Daqui estima-se o tempo em p com $t + T/2$.

15. Descreva sucintamente o algoritmo de sincronização de Berkeley.

O algoritmo de Berkeley é para sincronização interna de um grupo de computadores, onde uma máquina é escolhida para coordenar (master). O master pede periodicamente uma leitura aos restantes (slaves) e estima a hora em cada slave, pela observação do tempo de viagem das mensagens e pelo valor recebido (como em Cristian) e faz a média de todos os valores (incluindo o seu próprio tempo). O master em vez de enviar o tempo actualizado aos slaves (o que estaria sujeito ao tempo de envio variável), o master envia a cada um o valor exacto que deve usar para ajustar o seu relógio. Se o master falhar, outro será escolhido para assumir a sua função.

16. A distribuição dos objectos requer cuidados no Garbage Collection, especialmente do lado do servidor.

a) Explique resumidamente o algoritmo de GC no Object Model Distribuído.

A cada servidor mantém uma lista com o conjunto de processos com referências para os seus objectos e quando um cliente cria um Proxy para um objecto, é adicionado ao conjunto de processos com referências para aquele objecto. Quando o Garbage Collection do cliente detecta que o Proxy do objecto já não é necessário, este envia uma mensagem ao servidor e elimina o proxy, onde o servidor remove o processo da lista e quando a lista estiver vazia, o Garbage Collection do servidor recupera o espaço do objecto, excepto se existirem referências locais.

b) Identifique eventuais vulnerabilidades do algoritmo face a falhas no cliente.

Eventuais vulnerabilidades do algoritmo são que o servidor (com objectos remotos) atribui um intervalo de tempo ao cliente e a contagem é válida até que o tempo expire ou o cliente remova a referência do objecto.

17. O protocolo NTP permite três modos para sincronização de relógios em servidores.

a) Descreva esses métodos de sincronização.

Os métodos de sincronização são o multicast que usa-se em redes locais de alta velocidade, onde um ou mais servidores enviam periodicamente o tempo num broadcast e os servidores noutras máquinas acertam o relógio assumindo um pequeno delay. O multicast é de baixa precisão. Outro método é o procedure-call onde um servidor aceita pedidos de outros computadores, aos quais responde com a informação horária que tem. Este é utilizado quando se pretende maior precisão que no modo multicast, ou simplesmente não é possível multicast. O terceiro é o modo simétrico que serve para sincronizações entre servidores que fornecem a informação em redes locais e em níveis mais altos da NTP subnet, onde se pretende a máxima precisão. Aqui um par de servidores trabalha de modo simétrico, troca mensagens com informação horária e o tempo das mensagens também é considerado.

b) Qual o método onde se consegue a precisão máxima?

O método onde se consegue a precisão máxima é no modo simétrico.

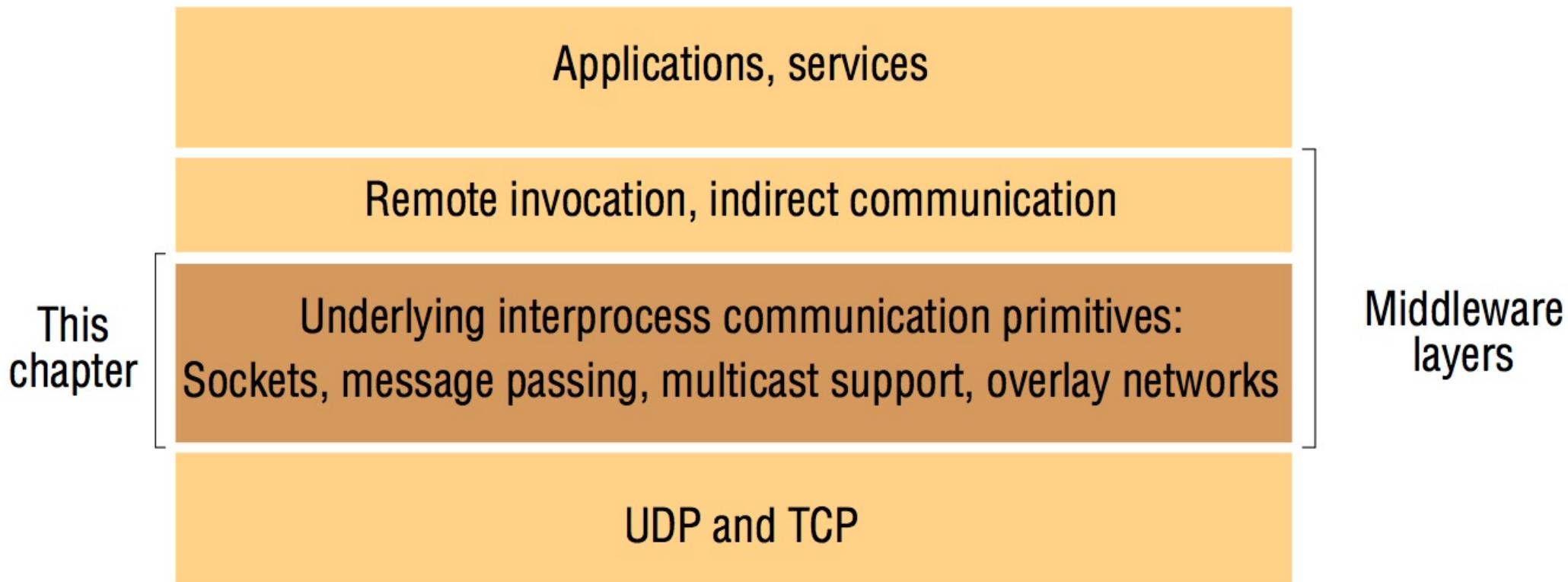
Sistemas Operativos II

Comunicação em Sistemas Distribuídos
camada inferior de *middleware*

Introdução

- ◆ Camada “inferior” do *Middleware*, relacionada com detalhes de
 - ◆ comunicação
 - ◆ representação externa de dados
 - ◆ *marshalling*
- ◆ Características de protocolos de comunicação entre processos num SD

Middleware: camada inferior



Protocolos de Internet

- ♦ recordar...
 - ♦ UDP
 - ♦ abstração para comunicação entre processos
 - ♦ permite o envio de mensagens isoladas, através de pacotes chamados datagramas
 - ♦ TCP
 - ♦ abstração para comunicação entre processos
 - ♦ fornece uma ligação (*stream*, canal) bidirecional entre dois processos
- ♦ Comunicação: **Operações** envolvidas uma mensagem:
 - ♦ **send**
 - ♦ **receive**

Protocolos de Internet

- ◆ Existe uma **fila-de-espera (buffer)** associada ao processo destinatário
 - ◆ envio: a mensagem é adicionada à fila remota
 - ◆ recepção: a mensagem é retirada da fila local
- ◆ Comunicação **síncrona**: os processos emissor e receptor sincronizam-se a cada mensagem
 - ◆ *send* e *receive* são operações **bloqueantes**
 - ◆ o emissor fica parado no *send* até que o *receive* seja efetuado
 - ◆ ao efetuar um *receive*, o receptor fica bloqueado até a mensagem chegar
- ◆ Comunicação **assíncrona**: não há sincronização
 - ◆ *send* **não é bloqueante**
 - ◆ o emissor prossegue assim que mensagem passa ao *buffer* local de saída
 - ◆ *receive* pode ser **bloqueante** ou **não bloqueante***
 - ◆ * - o processo prossegue com um *buffer* que será preenchido em *background*, havendo uma notificação quando isso acontecer

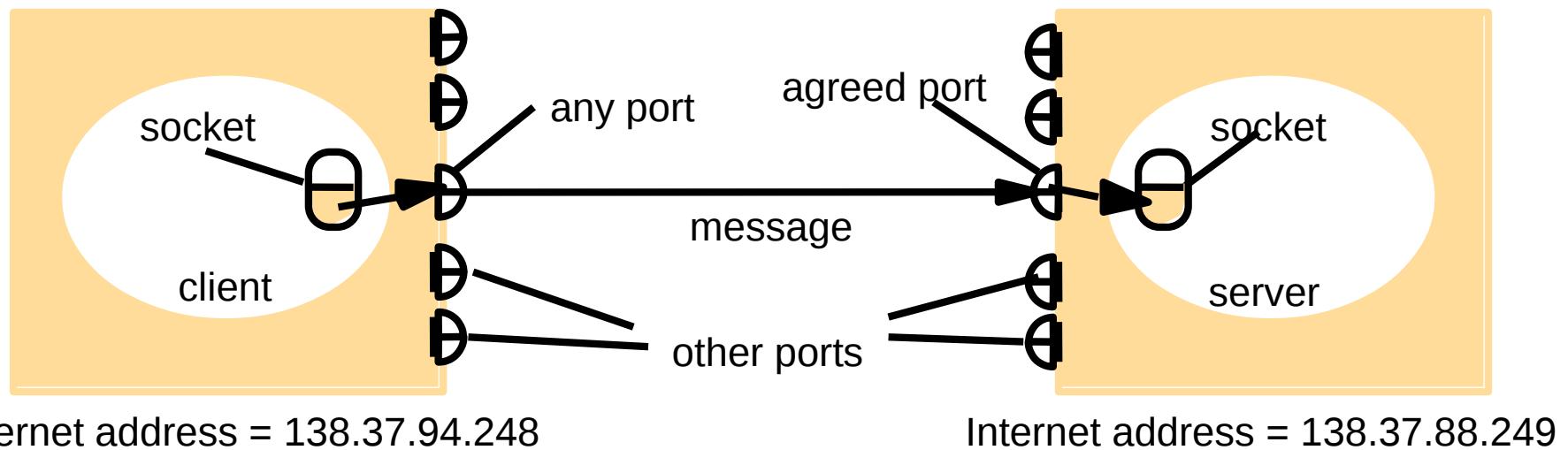
Protocolos de Internet

- ❖ Destino da mensagem: (endereço, porto)
- ❖ Porto:
 - ❖ inteiro, identifica o destinatário da mensagem numa máquina
 - ❖ associado a um só destinatário mas usado por vários emissores
 - ❖ um processo pode receber mensagens de vários portos
 - ❖ um processo não partilha portos com outros processos na mesma máquina
 - ❖ uma máquina tem 2^{16} números para portos disponíveis
- ❖ Preocupações com o envio de mensagens:
 - ❖ **fiabilidade**: garantia de entrega das mensagens, ainda que alguns pacotes sejam perdidos
 - ❖ **ordenação correta**: garantia de que as mensagens são entregues pela ordem com que foram enviadas (e não há trocas)

Protocolos de Internet

- ◆ O endereço do destinatário ou de um serviço:
 - ◆ endereço IP definido
 - ◆ obtido de um servidor de nomes (*binder*)
 - ◆ alguma transparência de localização
- ◆ UDP e TCP usam Sockets
(socket: abstração para o acesso ao canal de comunicação)
 - ◆ Comunicação entre processos
 - ◆ transmitir mensagens entre o *socket* de um processo e o *socket* de outro processo
 - ◆ Para um processo receber mensagens:
 - ◆ o *socket* tem de estar *bound* a um *porto* local e um *endereço* da máquina onde é executado

Sockets e Portos



UDP

- ◆ não há *acknowledgement* (confirmação) ou reenvios
- ◆ em caso de falha a mensagem não é recebida
- ◆ tamanho da mensagem:
 - ◆ IP: até 2^{16} bytes (headers + mensagem) (muitos ambientes limitam 8K)
 - ◆ para mensagens maiores, as aplicações devem dividir a mensagem em vários pedaços (*chunks*)
- ◆ Para enviar ou receber
 - ◆ é necessário um socket associado (*bound*) a um porto local e endereço IP
- ◆ *send* não bloqueante; *receive*: bloqueante (com possível timeout)
- ◆ Modo *receive from any*
 - ◆ a origem da mensagem é desconhecida
 - ◆ o endereço e porto do emissor é detectado na operação *receive*

UDP

- ◆ Modelo de Falhas
 - ◆ **falhas de omissão:** algumas mensagens podem eventualmente perder-se (não chegam, ou têm erros de *checksum* ou falta de espaço no *buffer* de chegada)
 - ◆ **ordenação:** não há garantia de entrega pela ordem de envio
- ◆ Caberá às aplicações assegurar algumas garantias de fiabilidade, por exemplo através de confirmações (ack.)
- ◆ Utilização de UDP
 - ◆ por vezes o risco destas falhas pontuais pode ser tolerável/aceitável
 - ◆ vantagem: menos *overheads* relacionados
 - ◆ estado da informação na origem e destino
 - ◆ latência
 - ◆ exemplo de utilização: DNS

Cliente UDP

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

Servidor UDP em ciclo a atender clientes

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

TCP

- tamanho da mensagem: à responsabilidade da aplicação, tamanho arbitrário. O TCP envia (de modo transparente) os dados em blocos, desde o buffer de saída
- usa *acknowledgement* e *checksums*
 - garante a entrega desses blocos reenviando aqueles cuja confirmação não chega antes de um *timeout*
- controlo do fluxo de dados: se o emissor é mais rápido, então é bloqueado até que o receptor tenha consumido alguns dados
- elimina duplicados e garante a ordem correta na recepção (através de números de sequência em cada pacote IP)
- com conexão
- bloqueante para leitura; bloqueante no envio apenas se o controlo de fluxo atuar

TCP

-
- ◆ Modelo de Falhas
 - ◆ não é *reliable/fiável*, na medida em que não garante a entrega em todos os cenários (quando há dificuldades na rede, a ligação pode perder-se)
 - ◆ um processo não distingue entre falha na rede e falha no processo do outro lado da ligação
 - ◆ um processo não sabe se uma mensagem (recente) já foi ou não recebida (pode estar ainda a ser reenviada)
 - ◆ Utilizações de TCP
 - ◆ HTTP, FTP, Telnet, SMTP

Cliente TCP: connect, send, receive

```

import java.net.*;
import java.io.*;

public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =
                new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);           // UTF is a string encoding see Sn 4.3
            String data = in.readUTF();
            System.out.println("Received: "+ data);
        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        finally {if(s!=null)try{s.close()};catch(IOException e)
        {System.out.println("close:"+e.getMessage());}}
    }
}

```

Portuguese Portuguese for Java Developers: System Concepts and Design Edn. 3
 © Addison-Wesley Publishers 2000

Servidor TCP: em ciclo, aceita ligações

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

// this figure continues on the next slide
```

Servidor TCP: concorrente, serviço echo

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

Repr. Externa de Dados e *Marshalling*

- ◆ Processos: estruturas de dados
- ◆ Mensagens na Comunicação: sequências de bytes
 - ◆ é necessário **converter** no envio e recepção
 - ◆ inteiros: *big-endian* (byte mais significativo primeiro, como *network byte order*), *little-endian*
- ◆ para uma estrutura de dados poder usar-se em RPC ou RMI
 - ◆ serializável, passível de ser representado de modo *flattened* e os tipos primitivos num formato acordado
- ◆ Representação Externa de Dados
 - ◆ formato usado para representação de estruturas e tipos primitivos
- ◆ *Marshalling*
 - ◆ tradução das estruturas e tipos primitivos para uma RED adequada para a transmissão
 - ◆ *unmarshalling*: processo inverso para reconstruir os dados à chegada

Repr. Externa de Dados e Marshalling

- ◆ Alternativas para RED:
 - ◆ CORBA CDR (*Common Object Request Broker Architecture CDR*)
 - ◆ Serialização Java – usada em RMI
 - ◆ <https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html>
 - ◆ Sun XDR
 - ◆ XML
 - ◆ JavaScript Object Notation (JSON)
 - ◆ <http://www.json.org/>
 - ◆ Protocol Buffers (Google)
 - ◆ <https://developers.google.com/protocol-buffers/>

CORBA CDR (*Common Data Representation*)

- ◆ tipos primitivos (short, long, float... char, boolean, bit) #15 tipos
 - ◆ bytes transmitidos de acordo com o emissor (*big-endian / little-endian*), que é também especificada na mensagem
- ◆ tipos compostos:

<i>Type</i>	<i>Representation</i>
<i>sequence</i>	length (unsigned long) followed by elements in order
<i>string</i>	length (unsigned long) followed by characters in order (can also can have wide characters)
<i>array</i>	array elements in order (no length specified because it is fixed)
<i>struct</i>	in the order of declaration of the components
<i>enumerated</i>	unsigned long (the values are specified by the order declared)
<i>union</i>	type tag followed by the selected member

Mensagem CORBA CDR

- não é possível passar uma estrutura com ponteiros
- exemplo com uma Struct:

<i>index in sequence of bytes</i>		<i>4 bytes</i>	<i>notes on representation</i>
0–3		5	<i>length of string</i>
4–7		"Smit"	'Smith'
8–11		"h "	<i>padding</i>
12–15		6	
16–19		"Lond"	<i>length of string</i>
20–23		"on "	'London'
24–27		1934	<i>unsigned long</i>

Representa uma **Struct Person** com os valores: {'Smith', 'London', 1934}

Marshalling CORBA

- ◆ Operações de Marshalling
 - ◆ geradas automaticamente a partir da especificação dos tipos de dados a transmitir na mensagem (argumentos, retorno)
- ◆ Tipos descritos com CORBA IDL (interface definition language)
- ◆ Assume-se que o emissor e receptor **conhecem os tipos** de cada elemento da mensagem, por isso o tipo não é passado (apenas o valor)

Java RMI

- ◆ objetos e tipos primitivos podem ser transmitidos
 - ◆ estes dados são transmitidos como argumento ou resultado de invocação remota de métodos
- ◆ formalmente, podem ser transmitidos (argumentos, retorno):
 - ◆ tipos primitivos
 - ◆ instâncias de classes que implementem a interface `java.io.Serializable`
- ◆ a aplicação que recebe a mensagem pode não conhecer o tipo dos dados
 - ◆ a representação serializada inclui informação sobre a classe do objeto (nome, versão/hash)

Serialização em Java

- Handle: referência para um objeto incluído na representação serializada

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

h0 e h1 são *handles*; O valor das Strings é precedido pelo comprimento

Cada objeto é escrito uma só vez, depois pode ser referido pelo seu Handle

Java Reflection

- ◆ Java Reflection
 - ◆ capacidade de descobrir propriedades de uma classe, como o nome e tipo de variáveis de instância ou métodos.
- ◆ Permite criar, obter e descobrir classes até então desconhecidas a partir do nome
- ◆ **java.lang.reflect**

Representação da Referência Remota de um Objeto

- ◆ Identificador para um objeto remoto válido no SD
- ◆ Cada objeto remoto tem uma única Ref. Remota
 - ◆ passada na mensagem do cliente que pede a invocação remota
- ◆ É importante garantir que a referência é única universalmente
 - ◆ junção de diversos elementos (data de criação...)

<i>32 bits</i>	<i>32 bits</i>	<i>32 bits</i>	<i>32 bits</i>	
Internet address	port number	time	object number	interface of remote object

Serialização em XML

```
<person id="123456789">
    <name>Smith</name>
    <place>London</place>
    <year>1984</year>
    <!-- a comment -->
</person >
```

Serialização em XML: esquema XML com tipologia da estrutura

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

Serialização em JSON

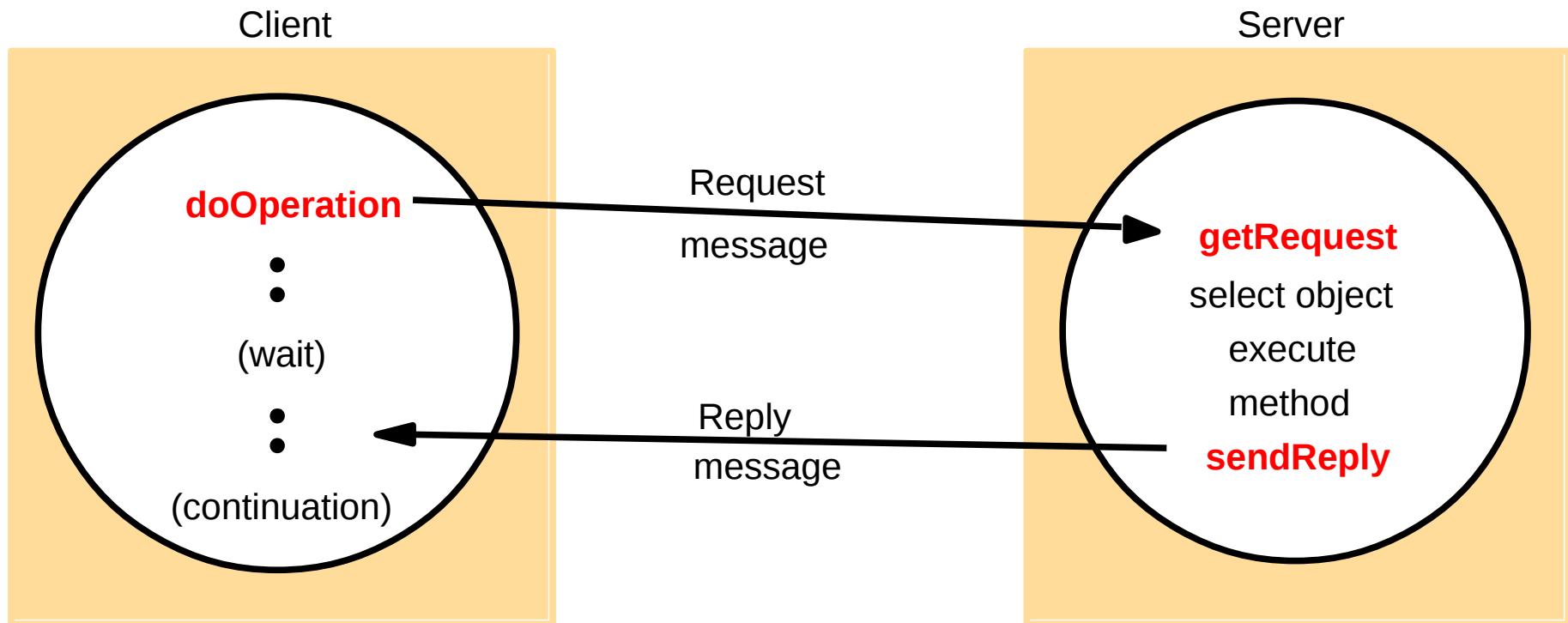
```
{"id": 123456789,  
 "name": "Smith",  
 "place": "London",  
 "year": 1984 }
```

Comunicação Cliente-Servidor

- ◆ usualmente baseado em comunicação Request-Reply síncrona
 - ◆ pois a resposta do servidor comprova a recepção da mensagem
 - ◆ mas também pode ser assíncrona (se o cliente recolhe as respostas mais tarde)
- ◆ UDP ou TCP
- ◆ UDP evita *overhead*:
 - ◆ *acknowledgement* – aqui são redundantes (já temos o reply)
 - ◆ estabelecimento da conexão (mais 2 pares de mensagens)
 - ◆ controlo de fluxo é desnecessário na maioria dos casos (são passados argumentos e resultados de tamanho reduzido)

Protocolo *Request-Reply*

- ◆ assenta em 3 primitivas:



Operações no Protocolo *Request-Reply*

- ◆ Primitivas usadas em *Request-Reply*:

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

envia a mensagem com um pedido ao objeto remoto e devolve a resposta.

Argumentos: o objeto remoto, o método a invocar e respetivos argumentos.

public byte[] getRequest ()

através de um porto no servidor, recebe o pedido do cliente

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

envia a mensagem de resposta para o cliente, através do seu par (endereço,porto)

Estrutura da Mensagem - *Request-Reply*

messageType
requestId
objectReference
methodId
arguments

int (0=Request, 1= Reply)

int gerado em doOperation()

RemoteObjectRef

int or Method

array of bytes

Identificador da mensagem composto por:

- **requestId** (torna o ID único para o emissor)
- **identificador do processo emissor** (e.g. endereço e porto) (torna o ID único no sistema)

Modelo de Falhas para *Request-Reply*

- ◆ se implementado sobre UDP: falhas de comunicação
 - ◆ falhas de omissão (algumas mensagens podem perder-se)
 - ◆ não há garantia de entrega pela mesma ordem do envio
- ◆ Falhas relacionadas com os processos (paragem, crash)
- ◆ Como tornar o sistema com UDP + fiável (a nível da aplicação):
 - ◆ utilizar timeout na operação doOperation, para reenvio de pedido
 - ◆ descartar requests duplicados (servidor verifica o identificador)
 - ◆ tratar mensagens perdidas – caso o servidor já tenha enviado o *reply*, então o *request* duplicado é processado para enviar de novo a resposta (se a op. for **idempotente**, caso contrário deve usar-se uma tabela, designada **Histórico** – ver adiante)
 - ◆ uso de Histórico: tabela com os últimos resultados que permite o reenvio de respostas sem repetir processamento)
 - ◆ normalmente precisa apenas da última resposta para cada cliente

RPC

- ◆ protocolos usados em RPC:
 - ◆ request (R)
 - ◆ request-reply (RR)
 - ◆ request-reply-acknowledge reply (RRA)
- ◆ diferentes comportamentos perante falhas de comunicação
- ◆ a sua utilização depende do nº de mensagens necessárias

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

Request-Reply sobre TCP

- ◆ facilita a transmissão de argumentos e resultado de tamanho arbitrário
- ◆ mais fiável
- ◆ evita a necessidade de filtragem de duplicados e reenvios a nível do protocolo *request-reply*

- ◆ TCP facilita a implementação do protocolo RR

- ◆ Exemplo de protocolo RR:
 - ◆ HTTP

HTTP: request-reply

- ♦ Request

<i>method</i>	<i>URL or pathname</i>	<i>HTTP version</i>	<i>headers</i>	<i>message body</i>
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

- ♦ Reply

<i>HTTP version</i>	<i>status code</i>	<i>reason</i>	<i>headers</i>	<i>message body</i>
HTTP/1.1	200	OK		resource data

HTTP 1.1: ligações persistentes

- perduram durante várias séries de mensagens request-reply, para evitar o *overhead* do estabelecimento de uma nova ligação

Sockets para Datagramas e Streams

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress)
•
•
sendto(s, "message", ServerAddress)
```

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress)
•
•
amount = recvfrom(s, buffer, from)
```

UDP

Requesting a connection

```
s = socket(AF_INET, SOCK_STREAM, 0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM, 0)
•
bind(s, ServerAddress);
listen(s, 5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

TCP
stream

Comunicação em Grupo

- ◆ *multicast operation*: operação que envia uma mensagem de um processo para cada um dos membros de um grupo de processos
- ◆ usualmente, a constituição do grupo é transparente para o emissor e não há garantias de entrega ou ordenação
- ◆ **Multicast** fornece uma útil infraestrutura para (construir) SD que incluem:
 - ◆ deteção de servidores em *spontaneous networking*
 - ◆ replicação de dados para melhor desempenho
 - ◆ tolerância a falhas baseada na replicação dos serviços
 - ◆ propagação de notificações de eventos

IP Multicast

- ◆ Multicast sobre Internet Protocol (IP)
 - ◆ pacotes IP são destinados a endereços (máquinas – sem portos (estes dizem respeito à camada de transporte))
- ◆ **multicast group:** definido por um endereço IPv4 classe D
 - ◆ Em particular na gama 224.0.0.0 a 239.255.255.255
 - ◆ Endereços geridos pela *Internet Assigned Numbers Authority*
- ◆ é possível enviar datagramas para um grupo sem ser membro
- ◆ a nível da programação, o IP Multicast pode usar-se apenas com UDP
 - ◆ os dados circulam como datagramas
- ◆ ao nível de IP, um computador pertence ao grupo multicast se um ou mais processos tem sockets associados ao grupo

IP Multicast

- ◆ Time To Live (TTL)
 - ◆ limita a distância de propagação de um datagrama *multicast*
 - ◆ indica o nº de *multicast routers* que o datagrama pode atravessar
- ◆ Endereços *Multicast*:
 - ◆ permanentes
 - ◆ existem mesmo quando não há membros no grupo
 - ◆ temporários
 - ◆ cessam quando não há membros

IP Multicast

- ♦ Modelo de Falhas no *multicast* de datagramas
 - ♦ falhas de omissão (inerentes ao UDP)
 - ♦ alguns membros podem não receber algumas mensagens
 - ♦ *unreliable multicast*: não garante a entrega de uma mensagem para cada membro do grupo
 - ♦ ordenação:
 - ♦ duas mensagens enviadas por processos diferentes podem não chegar pela ordem de envio
 - ♦ os pacotes IP de uma mensagem podem não chegar pela ordem de envio

Java API para IP Multicast: *join*, troca de datagramas com o grupo

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
        // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s=null;
        try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut =
                new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);
        }
    }
}
```

Java API para IP Multicast: *join*, troca de datagramas com o grupo

```
// get messages from others in group
    byte[] buffer = new byte[1000];
    for(int i=0; i< 3; i++) {
        DatagramPacket messageIn =
            new DatagramPacket(buffer, buffer.length);
        s.receive(messageIn);
        System.out.println("Received:" + new String(messageIn.getData()));
    }
    s.leaveGroup(group);
}catch (SocketException e){System.out.println("Socket: " + e.getMessage());
}catch (IOException e){System.out.println("IO: " + e.getMessage());}
}finally {if(s != null) s.close();}
}
```

Comunicação entre Processos - UNIX

- ◆ as primitivas para IPC (comunicação entre processos) estão disponíveis em *system calls* em cima de UDP ou TCP
- ◆ destino da mensagem: socket address (internet address, port)
- ◆ um processo pode criar um socket para comunicar com outro processo, invocando a *system call* **socket**
- ◆ Comunicação via:
 - ◆ datagramas - UDP
 - ◆ *streams* - TCP

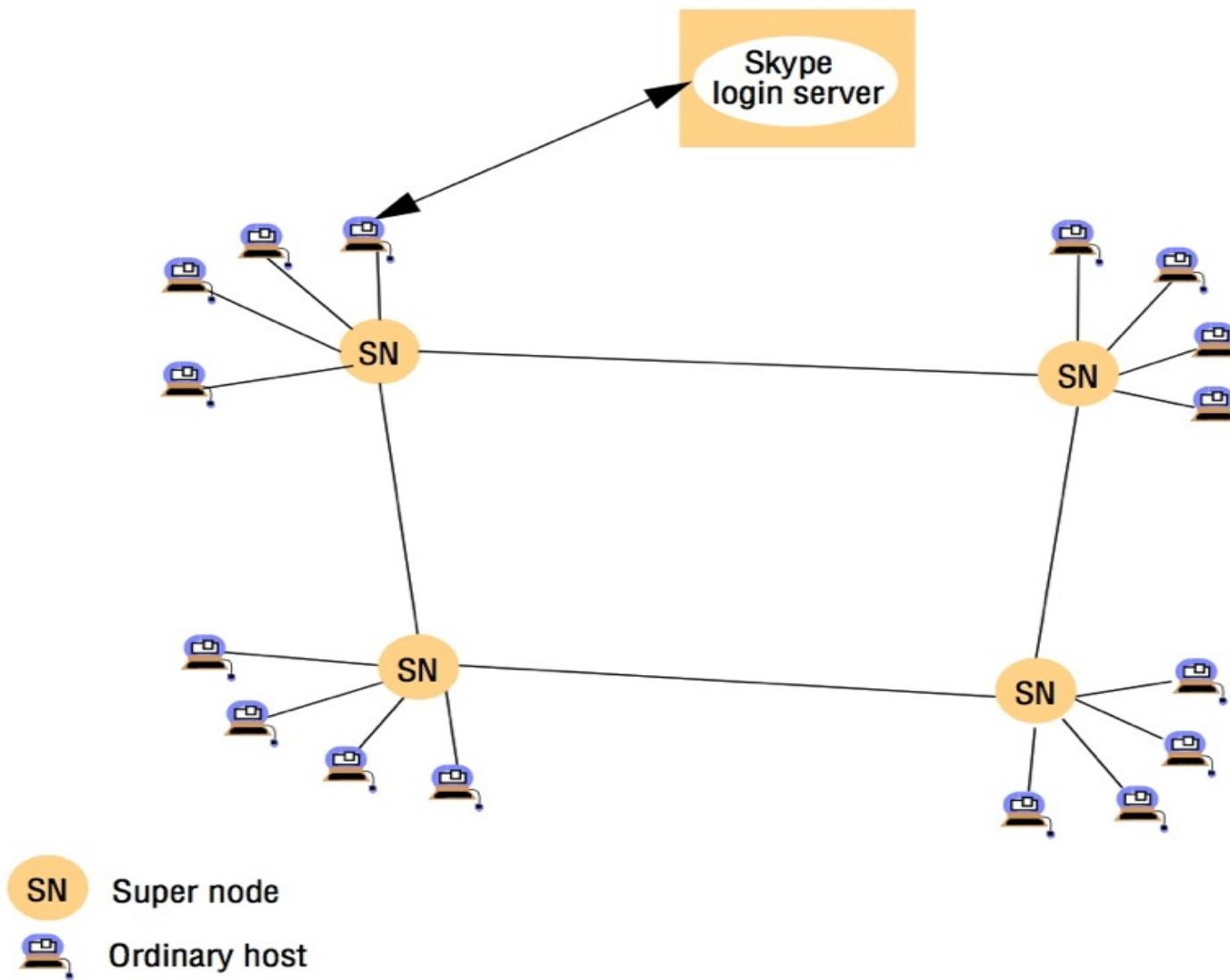
Redes Virtuais

- ◆ *Network Overlay / Virtual Network*
 - ◆ Rede virtual formada por um conjunto de nós e ligações lógicas/virtuais entre esses nós
 - ◆ Assenta numa rede convencional (como uma rede IP)
 - ◆ Esconde aspetos particulares dos segmentos em que assenta... fornece uma plataforma uniforme
 - ◆ Oferece serviços adicionais
 - ◆ Garantias adicionais de qualidade do serviço
 - ◆ Forma de encaminhamento (routing) própria
 - ◆ Segurança
 - ◆ Outros, vocacionados para uma aplicação ou serviço específico
 - ◆ Possível Desvantagem
 - ◆ Eventual perda de desempenho por existir mais uma camada

Caso de estudo: Skype

- ◆ Skype: aplicação Peer-to-Peer
 - ◆ de VoIP, IM, interfaces para serviço telefónico normal
 - ◆ Network overlay system com perto de 400 milhões de utilizadores
- ◆ Arquitetura: *Peer-to-peer...* com
 - ◆ Hosts cliente
 - ◆ Super nodes: hosts com capacidade suficiente
 - ◆ largura de banda, acessibilidade e/ou tempo de ligação ativa
 - ◆ endereço público de acessibilidade global
 - ◆ capacidade de processamento
- ◆ Funcionalidades importantes
 - ◆ Pesquisa de utilizadores
 - ◆ Envolve a consulta de 8 super nodes, em média
 - ◆ Chamadas de voz (TCP para negociar; UDP ou TCP para streaming) 47

Skype Network Overlay



MPI

- ◆ *Message Passing Interface (MPI)*

- ◆ Mecanismo para comunicação entre processos baseado nas primitivas send/receive, usado em **programação distribuída**, para computação de alto desempenho

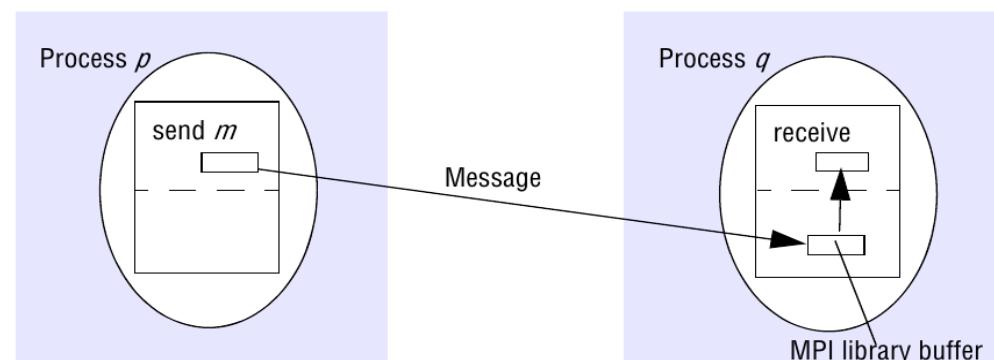
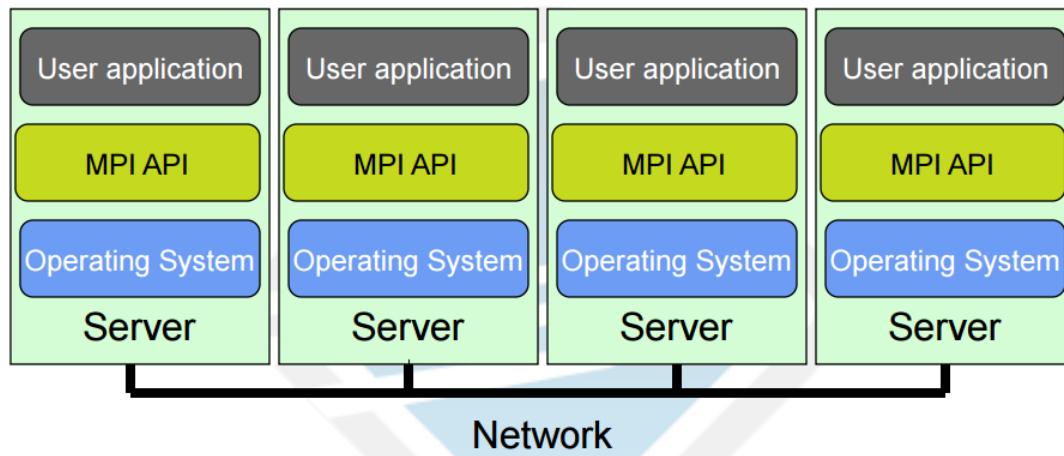
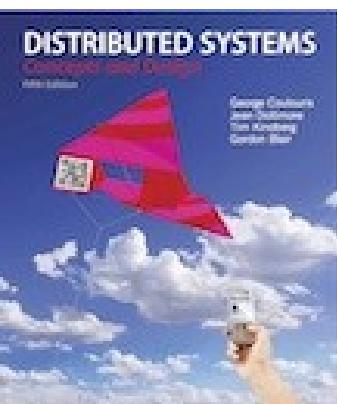


Imagen:

[https://www.open-mpi.org/video/general/what-is-\[open\]-mpi-1up.pdf](https://www.open-mpi.org/video/general/what-is-[open]-mpi-1up.pdf)

MPI

<i>Send operations</i>	<i>Blocking</i>	<i>Non-blocking</i>
<i>Generic</i>	<i>MPI_Send</i> : the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	<i>MPI_Isend</i> : the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via <i>MPI_Wait</i> or <i>MPI_Test</i> .
<i>Synchronous</i>	<i>MPI_Ssend</i> : the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	<i>MPI_Issend</i> : as with <i>MPI_Isend</i> , but with <i>MPI_Wait</i> and <i>MPI_Test</i> indicating whether the message has been delivered at the receive end.
<i>Buffered</i>	<i>MPI_Bsend</i> : the sender explicitly allocates an MPI buffer library (using a separate <i>MPI_Buffer_attach</i> call) and the call returns when the data is successfully copied into this buffer.	<i>MPI_Ibsend</i> : as with <i>MPI_Isend</i> but with <i>MPI_Wait</i> and <i>MPI_Test</i> indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
<i>Ready</i>	<i>MPI_Rsend</i> : the call returns when the sender's application buffer can be reused (as with <i>MPI_Send</i>), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	<i>MPI_Irsend</i> : the effect is as with <i>MPI_Isend</i> , but as with <i>MPI_Rsend</i> , the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),



Sistemas Operativos II

Modelos: Físico, de Arquitetura
e de Análise Fundamental

Abstrações

- Abstrações são familiares ao programador
 - Ajudam a simplificar um problema
 - Vários níveis
 - Os níveis superiores usam interfaces para as operações de níveis inferiores
- A distribuição tem implicações
 - Ausência de memória partilhada
 - Ausência de relógio global...
 - Concorrência
 - Falhas
 - Dificuldades de coordenação
- Modelo: especificação dos aspectos relevantes, um conjunto de pressupostos sobre o contexto e funcionamento de um sistema

Modelo Físico

- Modelo Físico
 - Representação da camada de hardware onde assenta todo o sistema distribuído
 - Identifica os diferentes computadores e outros dispositivos pertencentes ao sistema e o modo como estes estão interligados
 - Sem detalhes de tecnologia
 - Modelo físico mínimo
 - Conjunto (extensível) de computadores (os nós), interligados por rede
- Gerações de Modelos
 - Iniciais
 - redes locais, PCs e impressoras...
 - Escala global / Era da Web
 - desde 1990's; redes de redes; PCs e servidores (fixos)
 - Contemporâneo
 - nós podem ser notebooks ou smartphones; mobilidade

Dificuldade e ameaças nos SD

- variados modos de utilização
 - distintas necessidades
 - capacidade de processamento
 - volume de informação
- grande variedade de ambientes
 - heterogeneidade de hardware, sistemas operativos, redes e respectivos protocolos...
- problemas internos
 - sincronização de relógios, falhas de hardware ou software, incoerência decorrente da concorrência
- problemas externos
 - ataques à integridade, confidencialidade, *denial of service*

Gerações de Sistemas Distribuídos

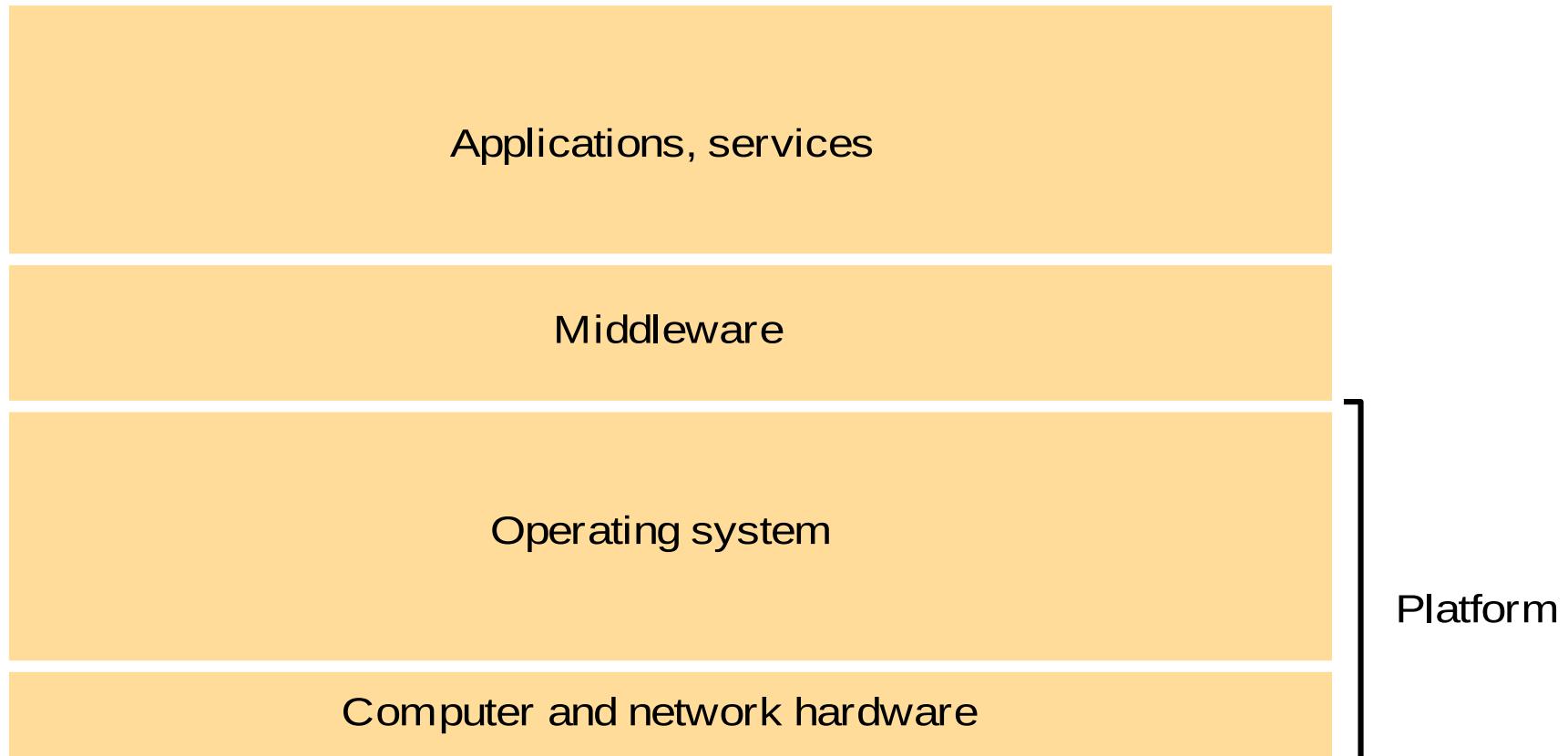
<i>Distributed systems:</i>	<i>Early</i>	<i>Internet-scale</i>	<i>Contemporary</i>
<i>Scale</i>	Small	Large	Ultra-large
<i>Heterogeneity</i>	Limited (typically relatively homogenous configurations)	Significant in terms of platforms, languages and middleware	Added dimensions introduced including radically different styles of architecture
<i>Openness</i>	Not a priority	Significant priority with range of standards introduced	Major research challenge with existing standards not yet able to embrace complex systems
<i>Quality of service</i>	In its infancy	Significant priority with range of services introduced	Major research challenge with existing services not yet able to embrace complex systems

Outros Modelos

- Modelos de Arquitetura
 - modelo *client-server*
 - *variantes*
 - modelo *peer process*
 - modelos de camadas *two-tier* e *three-tier*
- Modelos de Análise Fundamental
 - modelo de interação
 - modelo de falha
 - modelo de segurança

Camadas nos SD

cada camada fornece um conjunto de serviços à camada superior, escondendo os detalhes do nível abaixo



Categorías de Middleware

<i>Major categories:</i>	<i>Subcategory</i>	<i>Example systems</i>
<i>Distributed objects (Chapters 5, 8)</i>	Standard	RM-ODP
	Platform	CORBA
	Platform	Java RMI
<i>Distributed components (Chapter 8)</i>	Lightweight components	Fractal
	Lightweight components	OpenCOM
	Application servers	SUN EJB
	Application servers	CORBA Component Model
	Application servers	JBoss
<i>Publish-subscribe systems (Chapter 6)</i>	-	CORBA Event Service
	-	Scribe
	-	JMS
<i>Message queues (Chapter 6)</i>	-	Websphere MQ
	-	JMS
<i>Web services (Chapter 9)</i>	Web services	Apache Axis
	Grid services	The Globus Toolkit
<i>Peer-to-peer (Chapter 10)</i>	Routing overlays	Pastry
	Routing overlays	Tapestry
	Application-specific	Squirrel
	Application-specific	OceanStore
	Application-specific	Ivy
	Application-specific	Gnutella

Entidades em Comunicação, num SD

- Perspetiva orientada ao sistema
 - Processos
 - Threads
 - Eventualmente sensores e outros dispositivos; outros nós.
- Perspetiva orientada ao problema, a perspetiva da programação
 - Objetos
 - Componentes (CORBA, EJB)
 - Mecanismos parecidos aos objetos; oferecem uma abstração para implementar sistemas; baseados numa API; com algumas vantagens face ao manuseamento direto/convencional de objetos
 - Web Services
 - Outro paradigma para implementação de sistemas distribuídos, que recorre a protocolos e normas de internet, onde uma aplicação é identificada pelo URI

Paradigmas de Comunicação, entre entidades no SD

- **IPC**
 - Comunicação com API de baixo nível
 - Exemplos: comunicação baseada em sockets, com API do Soperativo
- **Invocação Remota**
 - Execução remota de processo ou método
 - Protocolos Request-Reply
 - RPC
 - RMI
- **Comunicação Indireta**
 - Comunicação em grupo (um para vários; *multicast*)
 - *Publish-subscribe* (um para muitos)
 - *Message Queues* (um para muitos, mas ponto a ponto; fila entre produtor e consumidor)
 - *Distributed shared memory* (DSM): abstração para partilha de dados entre processos que não partilham a mesma memória física

Agentes de comunicação e paradigmas de comunicação

<i>Communicating entities (what is communicating)</i>		<i>Communication paradigms (how they communicate)</i>		
<i>System-oriented entities</i>	<i>Problem-oriented entities</i>	<i>Interprocess communication</i>	<i>Remote invocation</i>	<i>Indirect communication</i>
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

Arquitetura do Sistema

- O sucesso com que se satisfazem as necessidades atuais e futuras depende da arquitetura do sistema
- Preocupações:
 - fiabilidade
 - permitir a configuração do sistema (adaptabilidade)
 - rentabilidade

Arquitetura do Sistema

- Modelo de Arquitetura (*Architectural Model*) de um SD:
 - Descreve o sistema em termos das tarefas computacionais e comunicação desempenhadas por cada elemento
 - **Elemento:** computador; ou conjunto de computadores interligados
- Definir a disposição dos componentes individuais de um sistema numa rede de computadores e o modo como interagem entre si
 1. simplifica e abstrai o papel de cada componente individual
 2. considera a disposição dos componentes pela rede
 - identificar distribuição de dados e locais onde se requer grande quantidade de trabalho/processamento) (*workload/carga*)
 3. relações entre os componentes
 - do ponto de vista do papel funcional que desempenham
 - comunicações que estabelecem

Arquitetura do Sistema

- Simplificação inicial do papel de cada componente:
 - classificar os processos como:
 - server
 - client
 - peer processes: processos que cooperam e comunicam de forma simétrica para realizar uma tarefa (usualmente de interesse comum)
 - esta classificação facilita a percepção:
 - responsabilidades dos componentes
 - locais de *workload* (carga)
 - ajustes para alcançar fiabilidade e eficiência

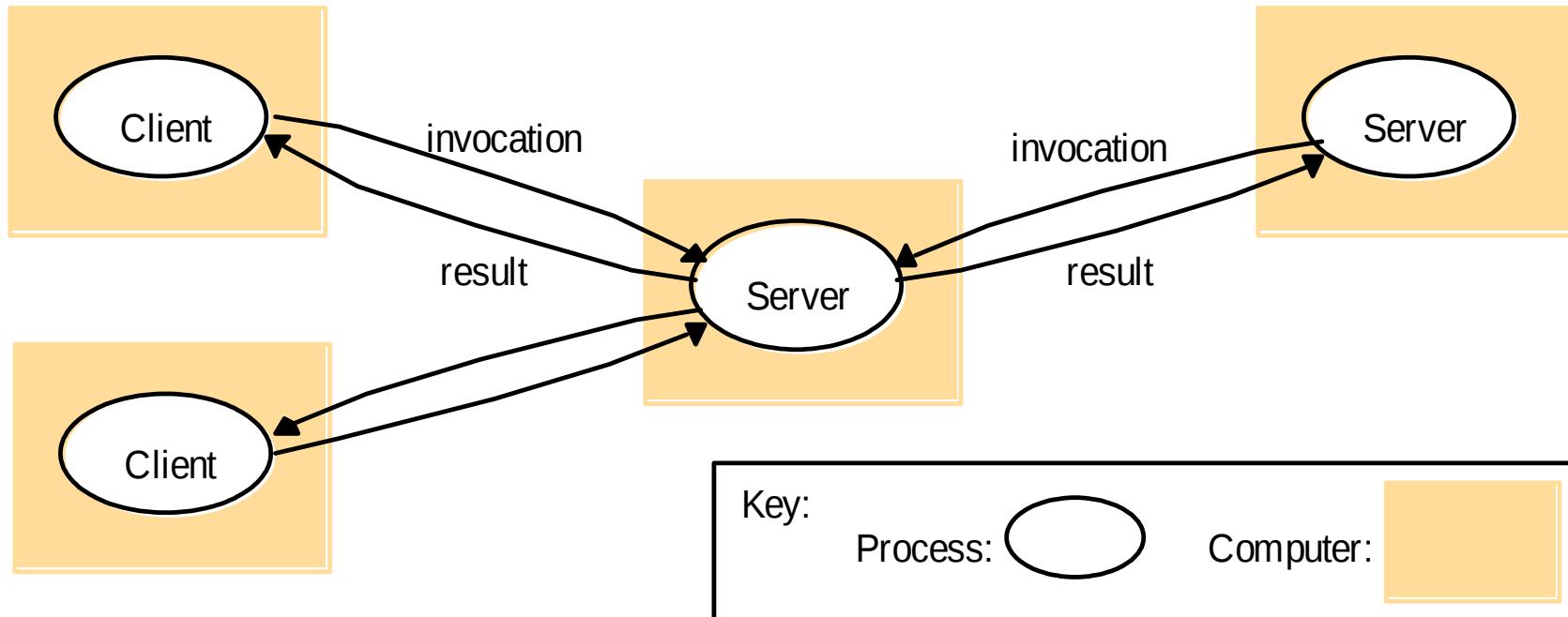
Modelos na Arquitetura de SD

- cliente-servidor
- variantes do modelo cliente-servidor
 - divisão ou replicação dos dados em sistemas de múltiplos servidores
 - caching de dados por clientes ou *proxy servers*
 - código móvel e agentes móveis
- *peer processes*
- *two-tier* e *three-tier*

Modelo Cliente-Servidor

Cientes invocam um servidor

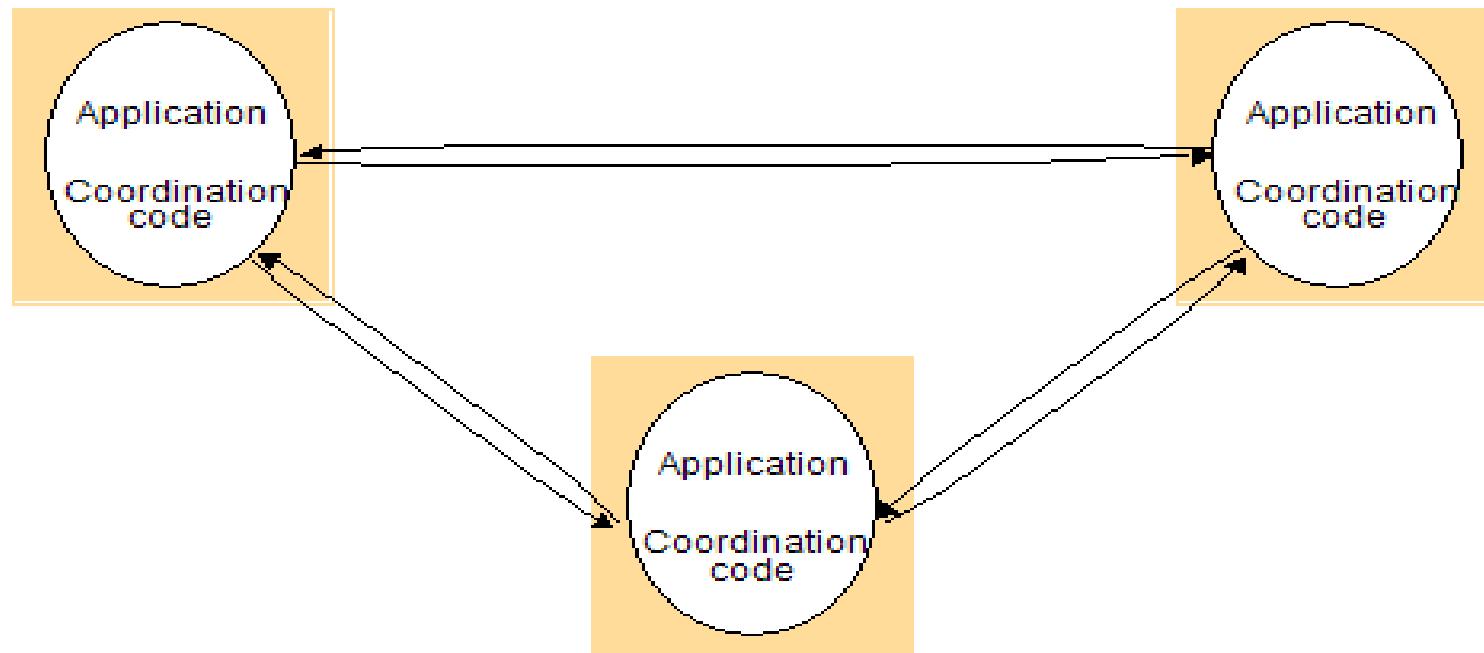
-o servidor pode também assumir o papel de cliente para ligar-se a outro servidor



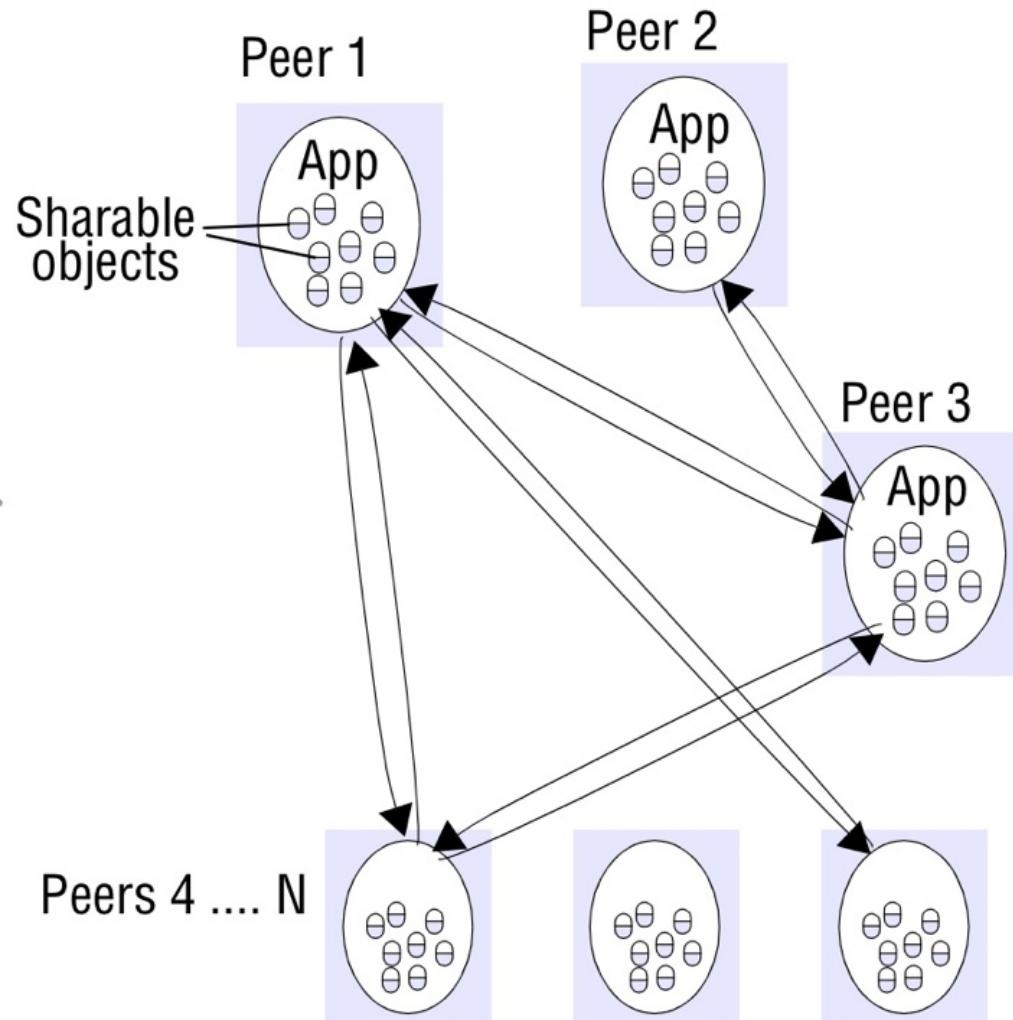
Modelo Peer Processes

Aplicação baseada em *peer processes*

- arquitetura em que os processos (nós) desempenham papéis idênticos, sem uma separação prévia entre clientes e servidores
- pontualmente, os processos poder-se-ão comportar como clientes ou como servidores
- a interação varia em função das necessidades



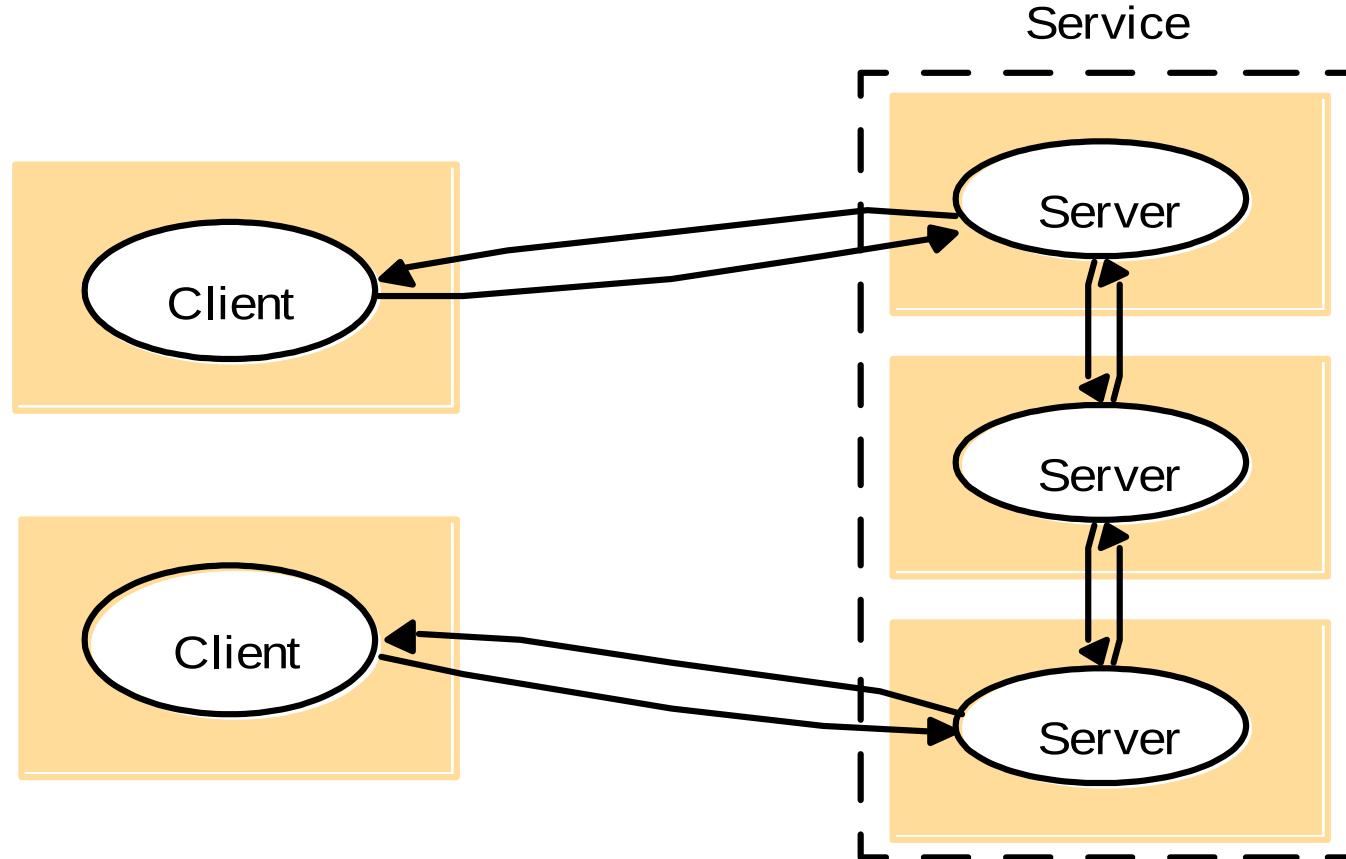
Modelo Peer Processes



Variante do Modelo Cliente-Servidor: múltiplos servidores

Serviço prestado por múltiplos servidores

- os servidores podem correr em diferentes computadores
- cada servidor pode lidar com uma partição dos dados ou uma réplica dos dados

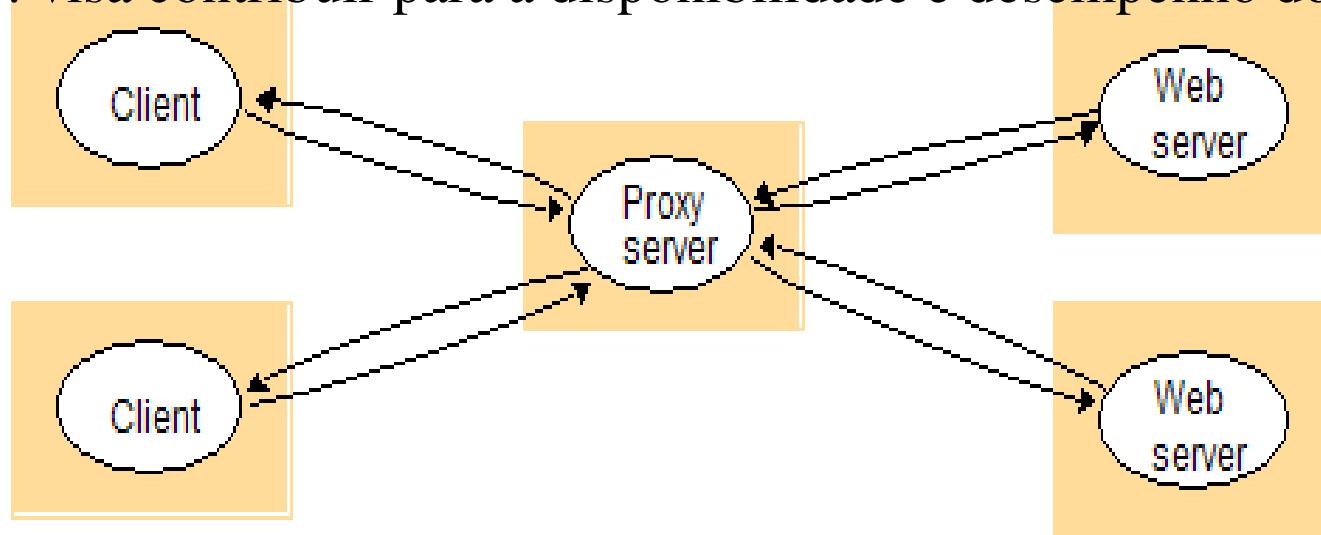


Variante do Modelo Cliente-Servidor: proxy e cache

Cache: uma réplica de dados acedidos recentemente que é mantida

- na aplicação cliente
- num servidor (Proxy Server), quando a réplica dos dados é disponibilizada a vários clientes

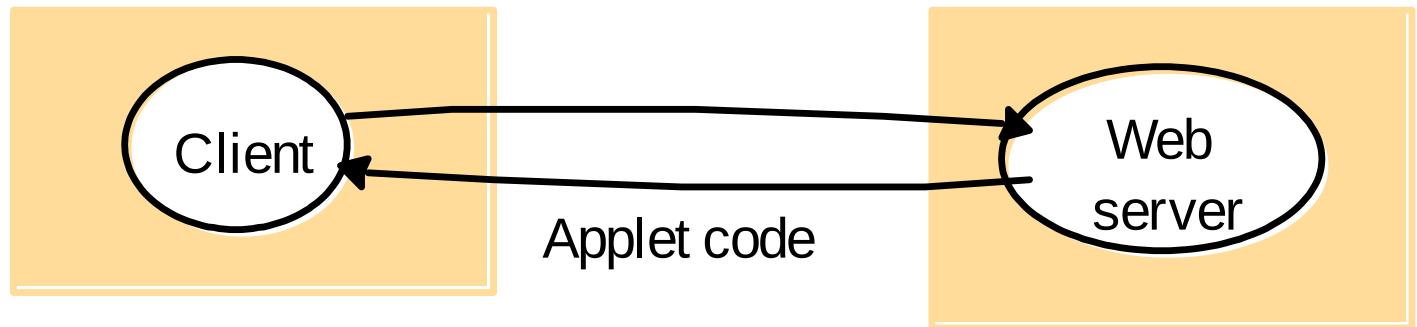
Cache: visa contribuir para a disponibilidade e desempenho do sistema



Variante do Modelo Cliente-Servidor: Código Móvel

- *Web applets*

- a) client request results in the downloading of applet code



- b) client interacts with the applet



Variante do Modelo Cliente-Servidor: Código Móvel

- Agentes móveis
 - programas em execução
 - incluem o código do programa e os dados
 - viajam na rede para realizar uma tarefa (para um utilizador ou um processo)
- *Network Computer*
 - SO e restante software obtido pela rede, a partir de um servidor remoto
 - execução local

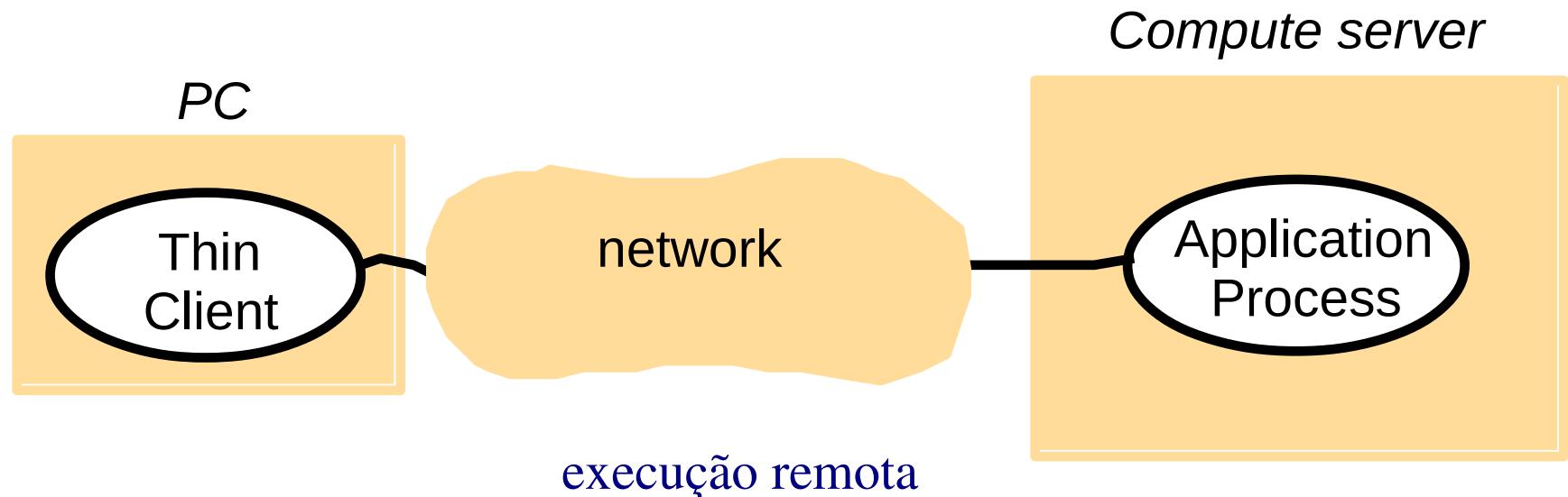
Variante do Modelo Cliente-Servidor: Código Móvel

- *Thin clients e compute servers*

- interface gráfica local (*thin client*) que controla a execução de aplicações num computador remoto (*compute server*)

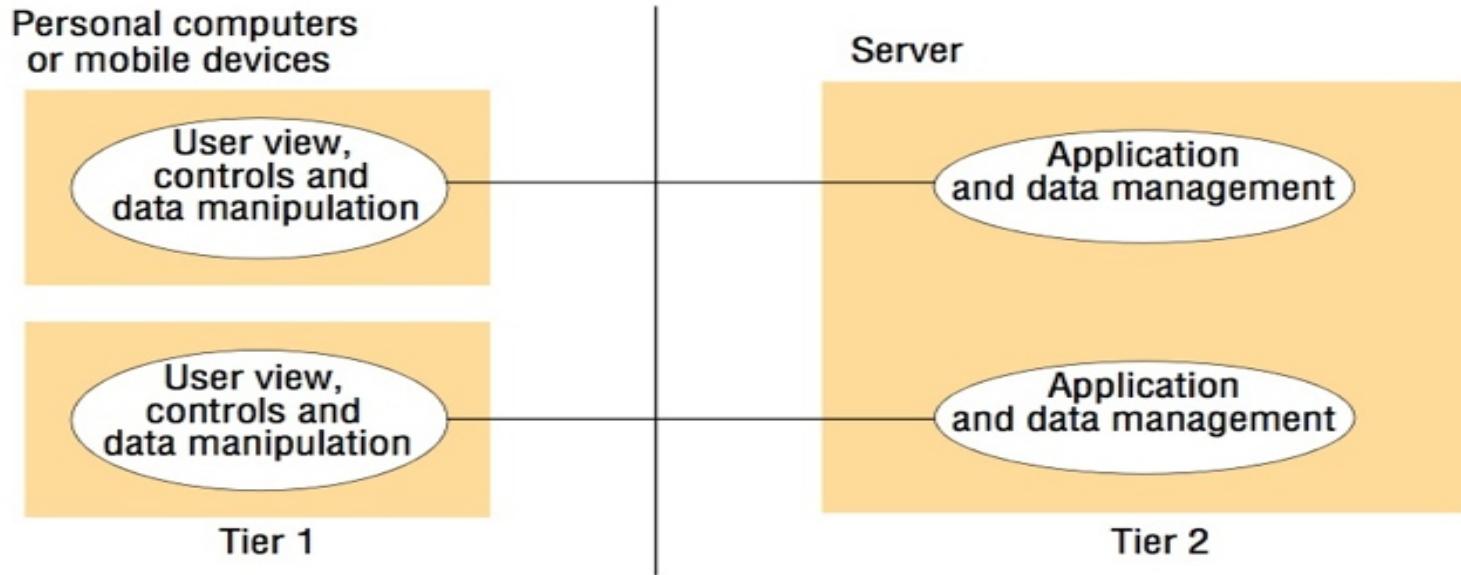
- Gestores gráficos

- X11 (clientes invocam operações no servidor via RPC)
- VNC: virtual network computer

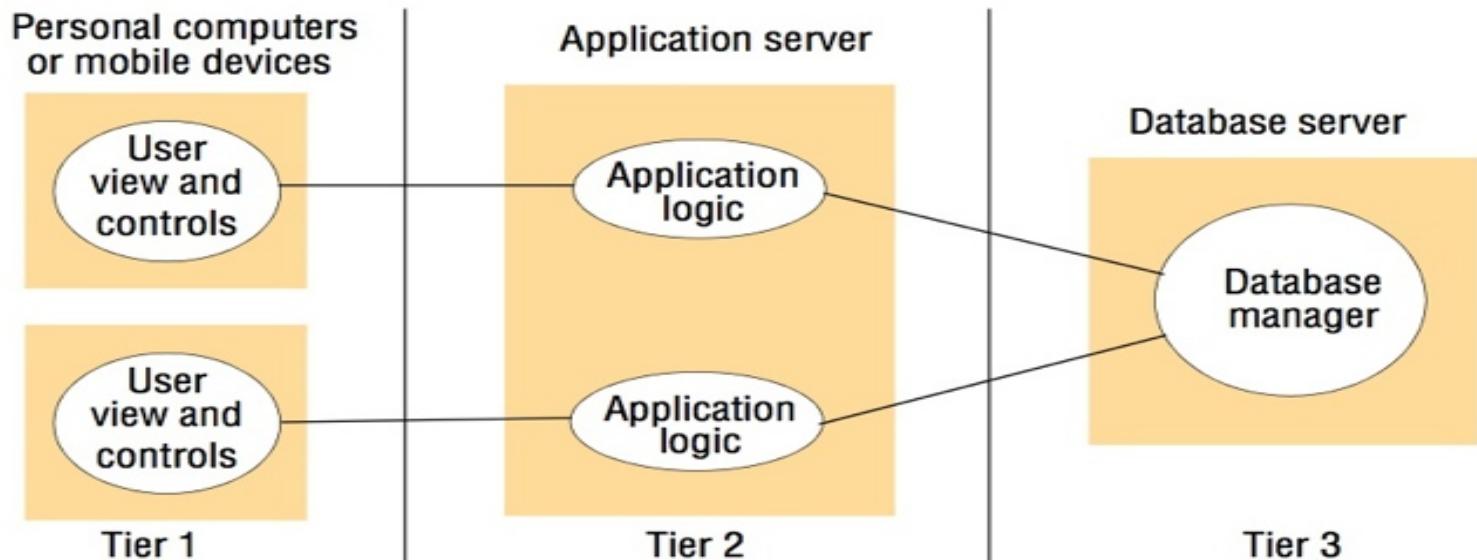


Aquiteturas Two-tier e Three-tier

a)



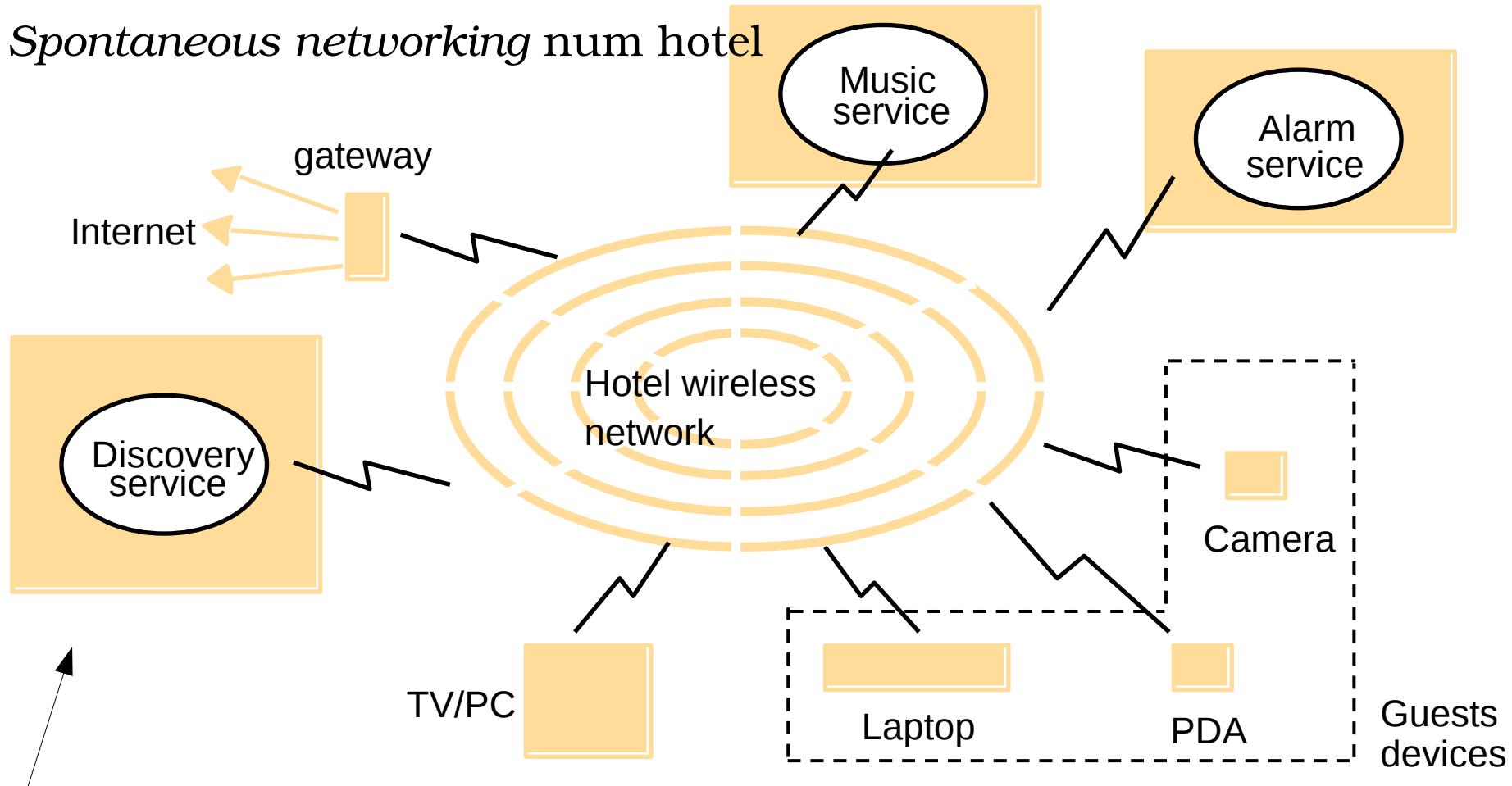
b)



Mobilidade e Redes de Computadores

Dispositivos móveis e *Spontaneous networking*

Spontaneous networking num hotel



Spontaneous Networking

- *Spontaneous Networking*
 - A forma de interação automática entre dispositivos móveis e outros dispositivos inseridos numa rede ou ambiente distribuído
- Aspetos chave
 - Facilidade de associação a uma rede local, de preferência sem fios
 - Conectividade limitada
 - Dificuldades inerentes à mobilidade
 - Capacidade reduzida dos equipamentos terminais
 - Aspectos de segurança e privacidade
 - Facilidade de integração com **serviços** locais
 - *Discovery Services*
 - *Registration Service + Lookup Service*

Interfaces e Objetos

- As funções disponíveis para interação com um processo são especificadas em **Interfaces**
- cliente-servidor (simples)
 - interface fixa
- modelos Orientados por Objetos (CORBA, RMI)
 - objetos podem variar: novas funcionalidades
 - interface dinâmica (tipicamente)

Requisitos no Desenho de Arquiteturas Distribuídas

- Desempenho
 - responsiveness: a acessibilidade às respostas
 - throughput: taxa de trabalho computacional por unidade de tempo
 - balanceamento (para resolver situações de carga)
- Qualidade do Serviço (QoS)
 - Depende de aspectos não funcionais, como a fiabilidade e o desempenho
- Cache e Replicação
 - cache: server proxy e também do lado do cliente
- Fiabilidade
 - correção
 - segurança
 - tolerância a falhas
 - *o serviço deve continuar a funcionar corretamente, mesmo na presença de falhas de hardware, software ou redes. Consegue-se com Redundância)*

Modelos de Análise Fundamental (*Fundamental Models*)

- Propósito
 - realçar aspetos de desenho, dificuldades e ameaças a considerar no desenvolvimento de SD, de modo que estes possam desempenhar a sua tarefa de modo correto, fiável, eficiente e seguro
- Descrição formal de aspectos
 - comuns a todos os modelos de arquitetura
 - que influenciam a sua fiabilidade
 - a nível de processos, rede e recursos

Modelos de Análise Fundamental

- Modelos de análise a diferentes níveis:
 - **Interação:** análise de aspectos relacionados com o desempenho e a dificuldade de estabelecer limites temporais num SD, associados à comunicação e coordenação entre processos
 - **Falhas:** especificação exata das falhas que poderão surgir em processos, dispositivos ou canais de comunicação/rede
 - **Segurança:** análise das possíveis ameaças a processos e canais de comunicação/rede
 - ataques internos
 - ataques externos

Modelo de Interação

- **Algoritmo**
 - sequência de passos a executar para completar uma tarefa
- **Algoritmo Distribuído**
 - sequência de passos a executar **por cada um dos processos** do sistema, incluindo a **transmissão de mensagens** (de dados e de coordenação) entre os mesmos, para completar uma tarefa

Modelo de Interação

- Fatores que influenciam a interação entre processos em SD:
 1. Desempenho dos canais de comunicação
 - latência
 - largura de banda
 - *Jitter*
 - variação no tempo necessário para o envio/entrega de fragmentos de dados de uma mesma mensagem.
Exemplo: *Streaming multimedia, > jitter > distorção < qualidade*
 2. Relógios e Eventos Temporais
 - *clock drift rate*
 - taxa de desvio do tempo face a uma referência correta

Modelo de Interação: Variantes

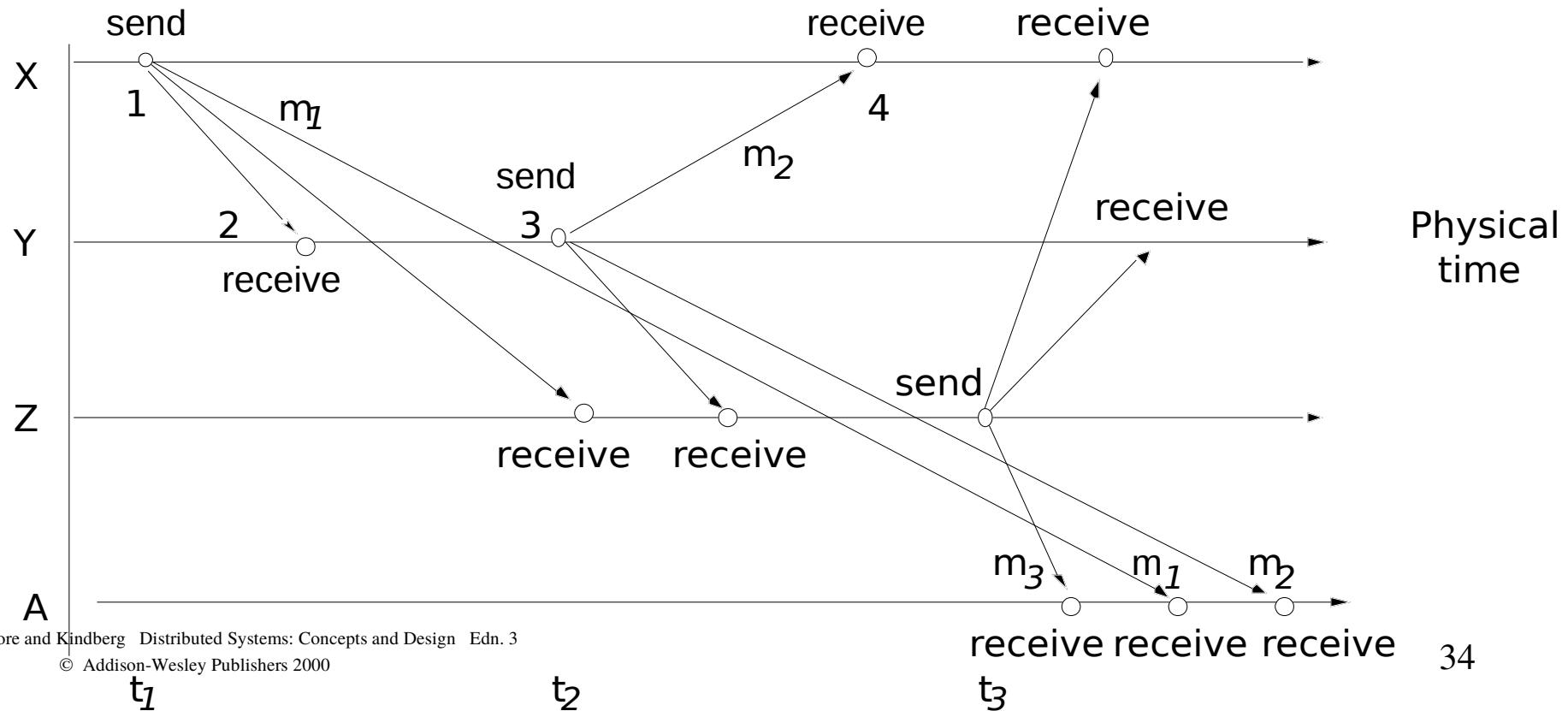
- **SD Síncronos:** existem limites para:
 - tempo de execução de cada passo de um processo
 - tempo até à recepção de uma mensagem enviada
 - *clock drift rate* (conhecido) em cada máquina
- **SD Assíncronos:** não há limite definido ou garantias para:
 - velocidade de execução de um processo
 - tempo de transmissão de uma mensagem, pode ter atraso (*delay*)
 - *clock drift rate*: a taxa é arbitrária
- exemplo de SD assíncrono: **Internet**

Modelo de Interação: Ordenação Cronológica de Eventos

Princípio do **Tempo Lógico**

regras para ordenação (ordenação causal ou outra)

E-mail: Y envia m2 depois de receber m1; X envia m1 antes de Y receber m1
... portanto o que concluir em A, sobre a ordenação de m1 e m2?



AJAX

- Outra variante da interação clássica da Web, com a arquitetura Cliente-Servidor
 - JavaScript no browser cliente
 - Aplicação servidor tem o estado da sessão na aplicação
 - Permite maior interatividade
 - A aplicação no *client-side*, no browser, pode fazer pedidos ao servidor
 - A resposta recebida pode ser processada e só depois se decide se atualiza o conteúdo web em exibição, eventualmente apenas de **modo parcial**, sem necessidade de refazer toda a página
 - Depois de fazer o pedido AJAX, o browser pode responder a outros inputs locais (*)... Quando a resposta do servidor chega, o AJAX processa os dados recebidos, podendo atualizar uma zona específica do conteúdo Web atualmente exibido
 - *- comportamento assíncrono

Protocolo com pedidos síncronos e assíncronos: AJAX e XMLHttpRequest

```
new Ajax.Request('scores.php?  
                  game=Arsenal:Liverpool',  
                  {onSuccess: updateScore});
```

```
function updateScore(request) {
```

```
....
```

(request contains the state of the Ajax request including the returned result.

The result is parsed to obtain some text giving the score, which is used to update the relevant portion of the current page.)

```
....
```

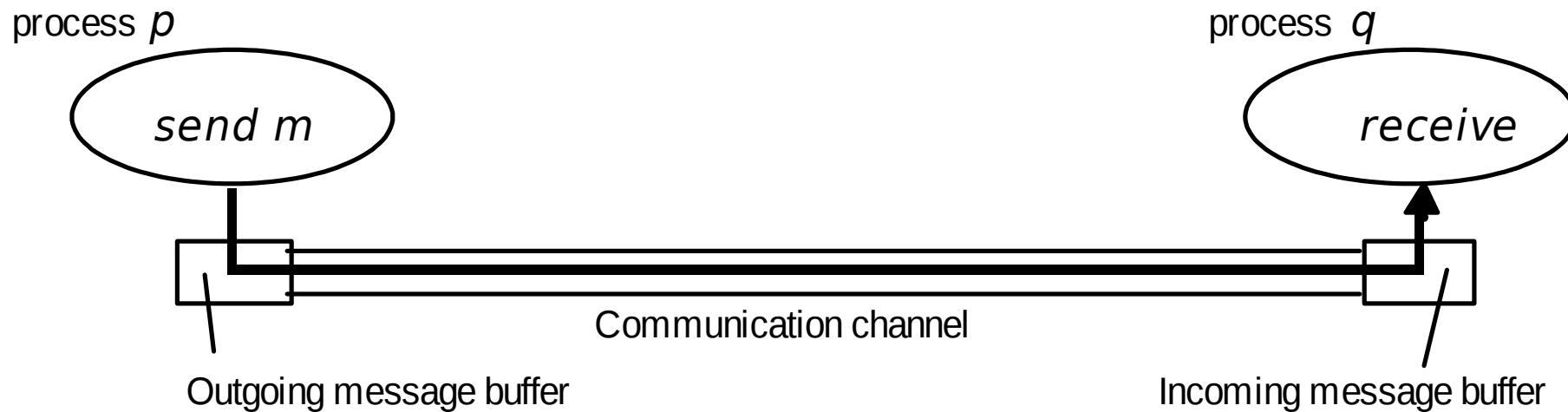
```
}
```

Modelo de Falhas

- Enumerar as **formas em que o sistema pode falhar**, facilitando a compreensão dos efeitos de cada falha

Modelo de Falhas

Processos e Canais de Comunicação



Modelo de Falhas

- Falhas de Omissão (*omission failures*)
 - quando o processo ou o canal falham no desempenho da tarefa que lhes cabe
 - ***process omission failures***
 - por crash efetivo ou por lentidão na resposta
 - ***communication omission failures***
 - *dropping messages* (canal falha o transporte entre os buffers)
 - *send-omission*
 - *receive-omission*
- Falhas Arbitrárias (bizantinas)
 - pior cenário
 - falhas diversas
- Falhas Temporais

Modelo de Falhas

Falhas por Omissão e Arbitrârias

<i>Tipo de Falha</i>	<i>Afecta</i>	<i>Descrição</i>
Fail-stop	Processo	Process halts and remains halted. Other processes <u>may detect this state</u> (Sistemas <u>Síncronos</u> onde há garantias de entrega).
Crash	Processo	Process halts and remains halted. Other processes <u>may not be able to detect this state</u> .
Omissão	Canal	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Processo	A process completes a <i>send</i> , but the message is not put in its outgoing message buffer.
Receive-omission	Processo	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrária (Bizantina)	Processo ou canal	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Modelo de Falhas

Falhas Temporais

sistemas síncronos: estes erros levam à não entrega de respostas (que teriam de chegar no intervalo determinado)

<i>Tipo de Falha</i>	<i>Afecta</i>	<i>Descrição</i>
Relógio	Processo	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Processo	Process exceeds the bounds on the interval between two steps.
Performance	Canal	A message's transmission takes longer than the stated bound.

Num sistema assíncrono estes fenómenos acarretam lentidão mas não correspondem necessariamente a Falhas Temporais, pois não há imposições temporais rígidas.

Modelo de Segurança

- Identifica possíveis ameaças num sistema distribuído (aberto)
 - ameaças a processos (clientes, servidores)
 - identidade do interlocutor remoto
 - legitimidade daquele para aceder ao recurso do processo
 - ameaças a canais de comunicação
 - introdução de mensagens forjadas
 - adulteração do conteúdo de mensagens em trânsito
- Visa:
 - garantir segurança de objetos, processos e dos canais de comunicação

Modelo de Segurança

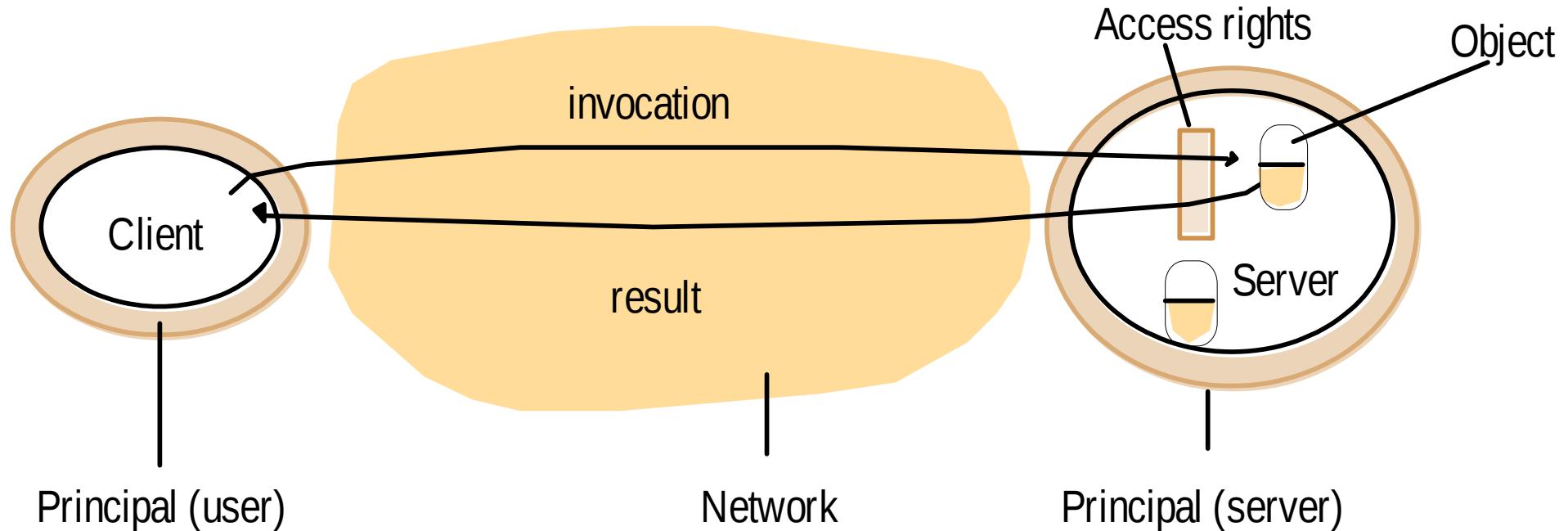
- proteger os objetos
 - *principal*: uma entidade envolvida, utilizador ou processo
 - direitos de acesso
 - especificar Quem pode fazer o Quê sobre Que Objetos

Modelo de Segurança

Interveniente (*principal*)

Objetos

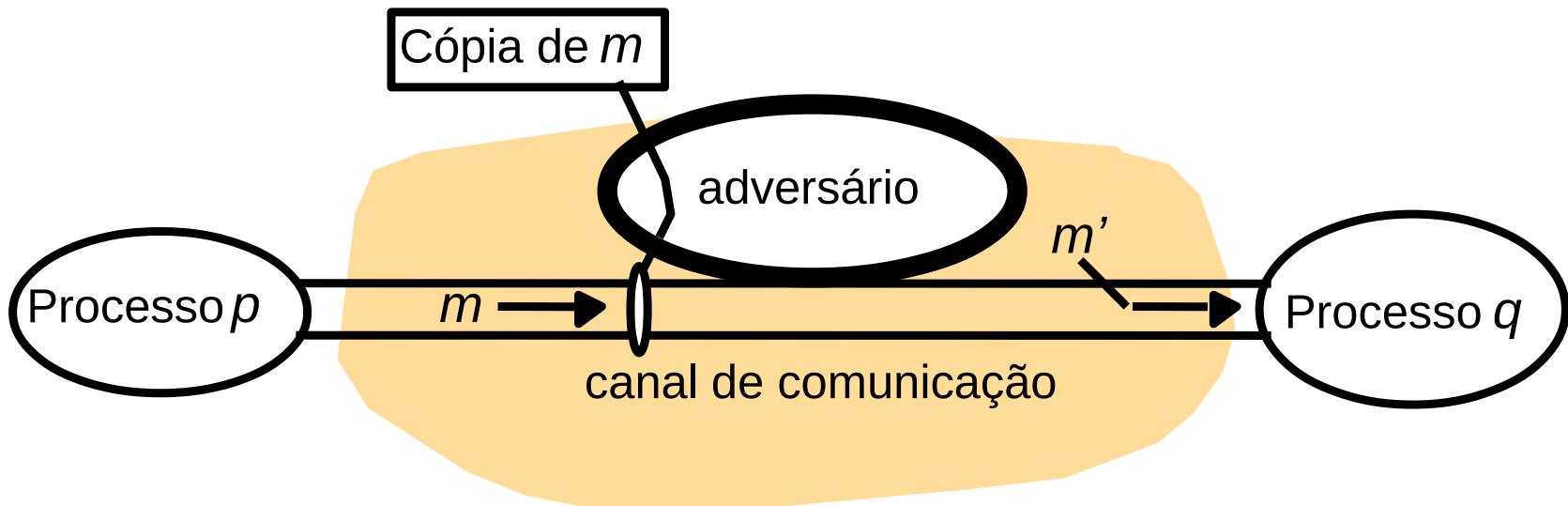
Privilégios de Acesso



Modelo de Segurança

Proteger os canais de comunicação contra adversários

- canais podem ser alvo de ataques externos por parte de utilizadores hostis (adversários/opONENTES/atacantes)

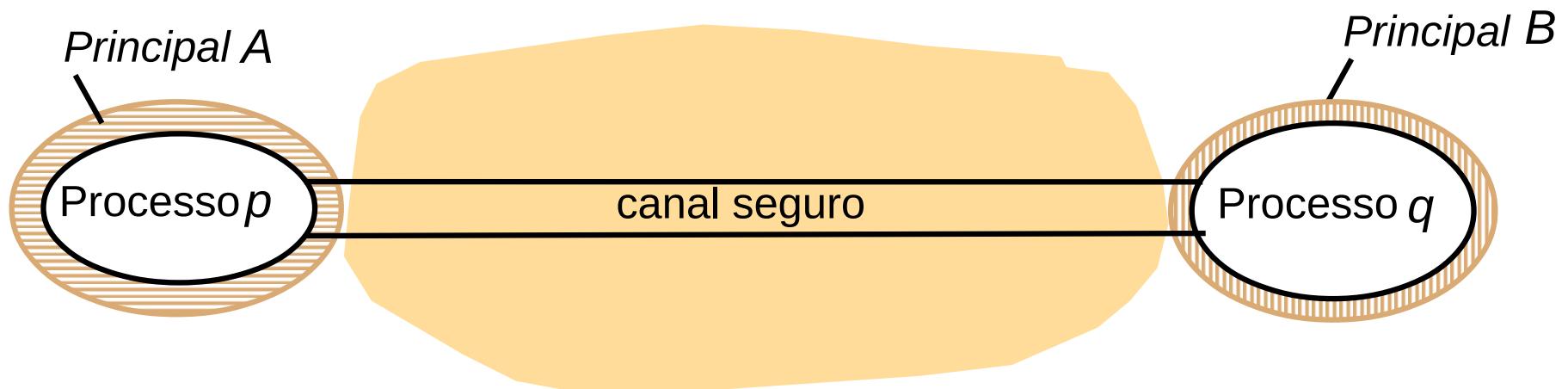


Modelo de Segurança

- Superar ameaças à segurança
 - criptografia
 - autenticação
 - canais seguros

Modelo de Segurança

Utilização de Canais Seguros*



* como exemplo, ver Stunnel

Modelo de Segurança

- Outros ataques possíveis
 - *denial of service*
 - *mobile code*
- Utilização de Modelos de Segurança
 - Repercussões na eficiência
 - Custos diretos e indiretos...
 - É preciso um compromisso entre os ganhos e o impacto que esses mecanismos acarretam...

Sistemas Operativos II

Características de um Sistema Distribuído

Conceitos relacionados com Paralelismo

- **Multiprocessor systems**

- Memória Partilhada
- **Bus**-based interconnection network
- E.g. SMPs (symmetric multiprocessors) with two or more CPUs

- **Multicomputer systems**

- Memória Não Partilhada
- Tipicamente homogéneos em hardware e software
- Massively Parallel Processors (MPP)
 - Tightly coupled high-speed network
- PC/Workstation clusters
 - High-speed networks/switches based connection.
 - A extensibilidade dos clusters dá lugar a alguma heterogeneidade
 - Introdução de PCs comuns, de características diversas...

Conceitos relacionados com Paralelismo

- Computação Paralela
 - Uso simultâneo de múltiplos recursos computacionais para resolver um problema
- Programação Paralela
 - Concepção de um programa para executar computação paralela
 - OpenMP, MPI, GPU programming...
- Sistemas Distribuídos
 - Escalam mais que sistemas convencionais (de Memória Partilhada)
 - Menos dependentes de hardware especializado
 - A programação destes sistemas é **complexa**
 - A performance pode degradar-se seriamente se não houver cuidados e se a arquitetura não for tida em consideração

Características de um Sistema Distribuído

- Uma definição:

Um **sistema distribuído** é aquele em que os seus componentes estão distribuídos por uma rede de computadores, comunicam e coordenam as suas ações através de troca de mensagens, e funciona para o utilizador como se de um sistema elementar se tratasse.

- Motivação

- porquê construir ou usar um SD?*

- a necessidade de partilhar recursos

- Recurso: hardware, periféricos ou software

Características de um Sistema Distribuído

- Os componentes de um SD encontram-se dispersos (por salas, edifícios, cidades ou países), como tal:
 - pode haver concorrência
 - não existe um relógio global
 - podem surgir falhas isoladas
 - Em qualquer aspecto do funcionamento de um dos componentes do sistema
 - Detetável??

Sistema Distribuído: exemplos em vários domínios

<i>Finance and commerce</i>	eCommerce e.g. Amazon and eBay, PayPal, online banking and trading
<i>The information society</i>	Web information, search engines , Wikipedia; Social networking: Facebook , MySpace...
<i>Creative industries and entertainment</i>	online gaming , music and film in the home, user-generated content, e.g. YouTube, Flickr
<i>Healthcare</i>	health informatics, on online patient records, monitoring patients
<i>Education</i>	e-learning, virtual learning environments; distance learning
<i>Transport and logistics</i>	GPS in route finding systems, map services : Google Maps, Google Earth
<i>Science</i>	The Grid as an enabling technology for collaboration between scientists
<i>Environmental management</i>	sensor technology to monitor earthquakes, floods or tsunamis

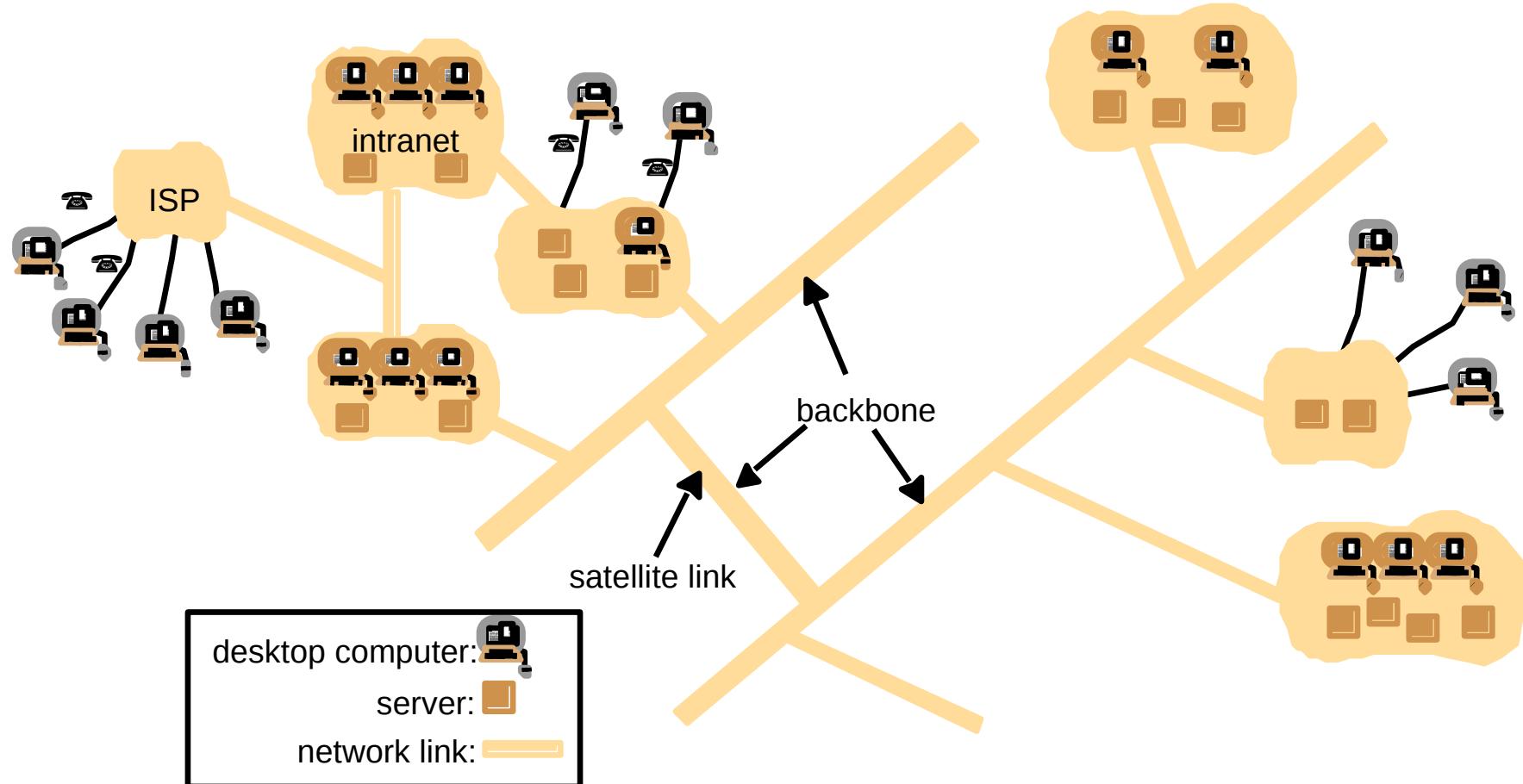
Exemplo de Sistema Distribuído

exemplo de SD: Internet

- Milhões de Aparelhos Terminais* interligados
 - * PCs, servidores, laptops, tablets, smartphones
- Equipamentos de rede
 - Canais de comunicação
 - Routers
- Normas da Internet
 - RFC: *Request for comments*
 - desde 1969
 - IETF: *Internet Engineering Task Force*
 - 1994: Português escreve o RFC 1713: *Tools for DNS debugging*

Exemplo de Sistema Distribuído

exemplo de SD: Internet



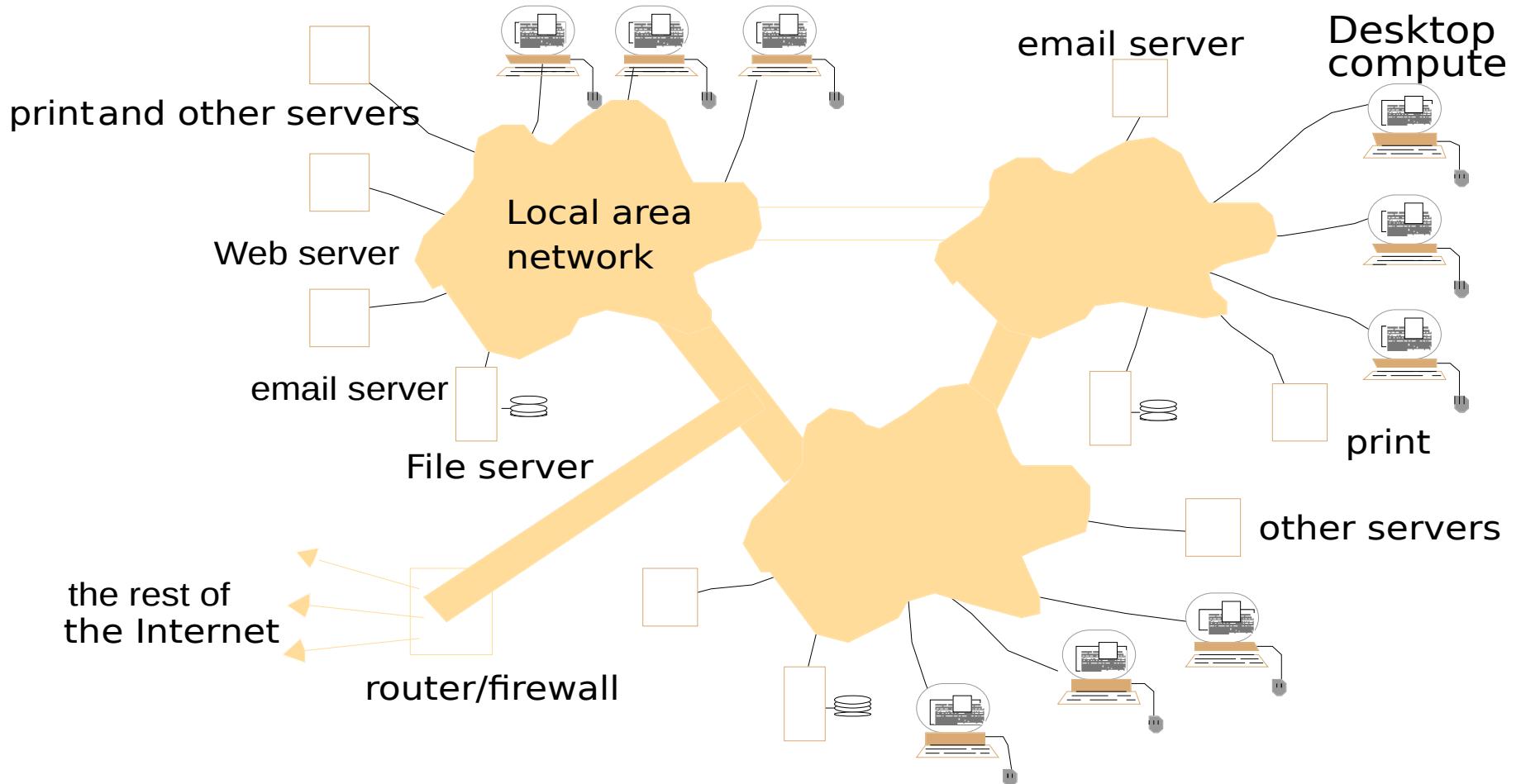
Exemplo de Sistema Distribuído

exemplo de SD: **Intranet**

- porção da internet com administração própria, com limites que permitem a aplicação de políticas locais de segurança
 - Firewall
 - Controlar o tráfego de e para o exterior

Exemplo de Sistema Distribuído

exemplo de SD: Intranet



Exemplo de Sistema Distribuído

exemplo de SD: **Web**

ou *World Wide Web*, ou WWW, ou W3

- 1991/1992: Tim Berners-Lee

- na altura no CERN, Suíça

- Um dos serviços da Internet

- Para publicar e aceder a recursos

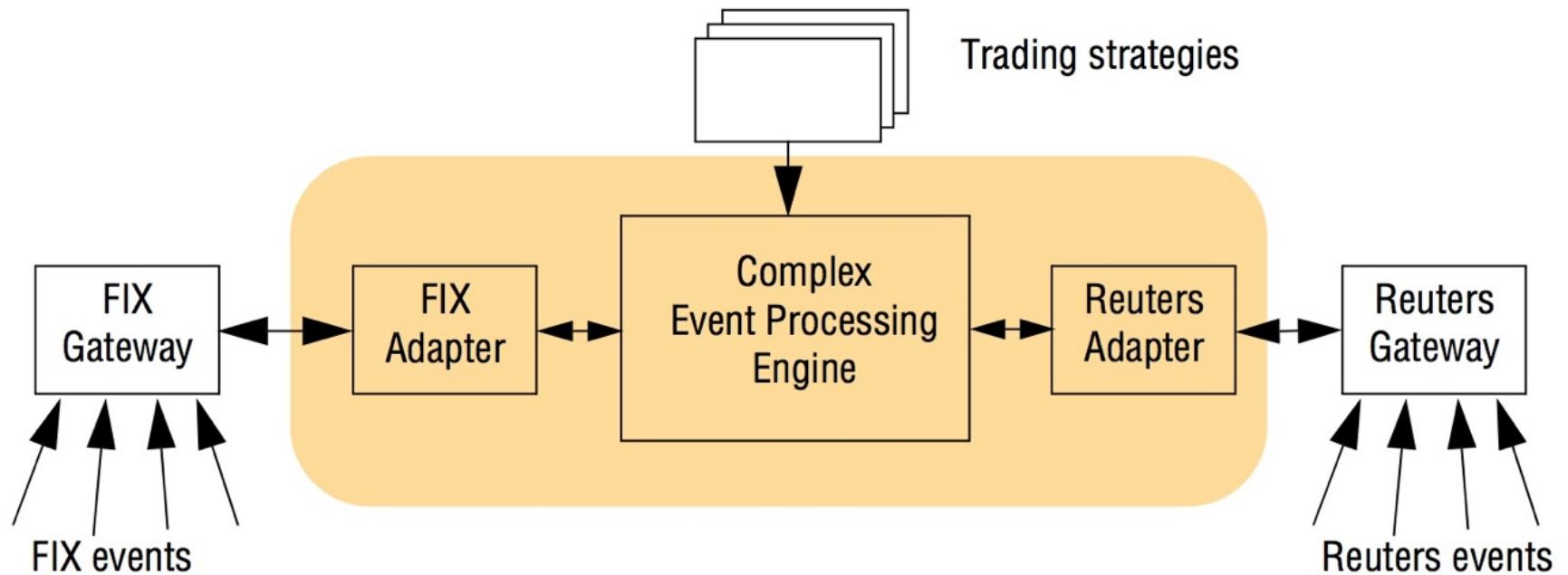
- Documentos multimédia com referências cruzadas (hipertexto)



1º Web Server

Exemplo de Sistema Distribuído

Trading – sistema de transações financeiras



Características de um Sistema Distribuído

Alguns conceitos:

- serviço

- uma funcionalidade a disponibilizar, que normalmente envolve o uso de alguns recursos

- servidor

- um programa em execução num computador
 - aceita pedidos de outros computadores para prestar um serviço

- cliente

- programa que faz pedidos ao servidor

Características de um Sistema Distribuído

Alguns conceitos:

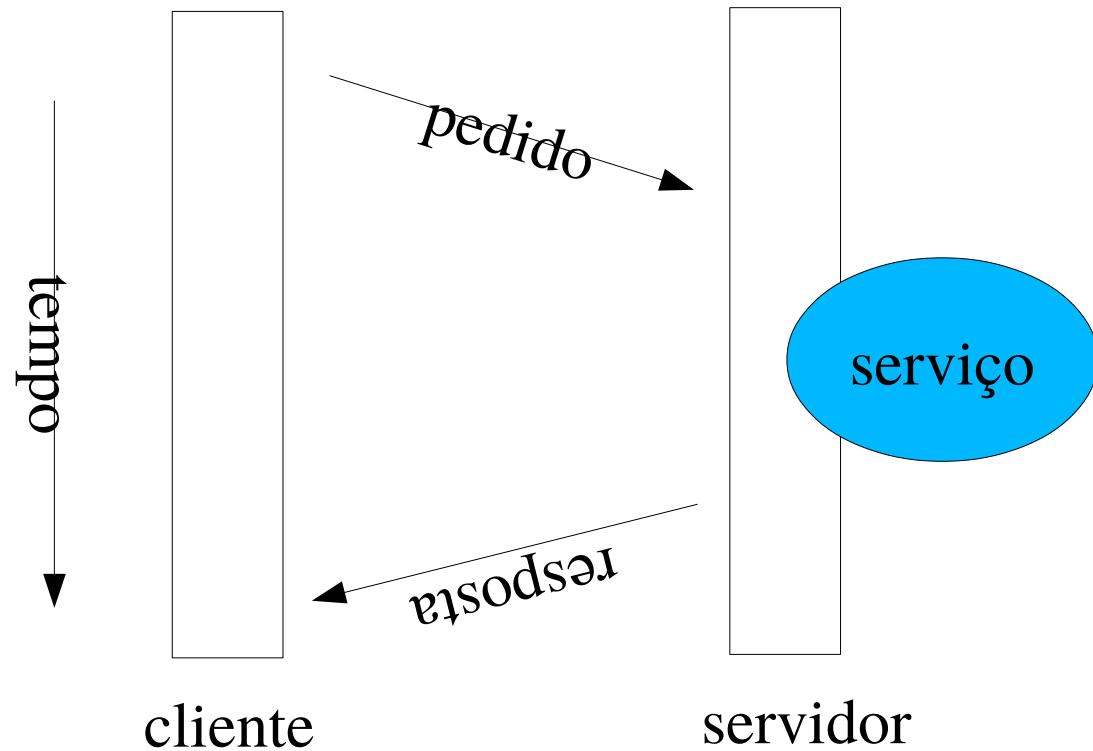
- **processo**

uma instância de um programa em execução. Compreende um ambiente de execução e uma ou mais *threads*.

- ***thread***

uma abstração do SO para uma tarefa ou atividade

Características de um Sistema Distribuído



Características de um Sistema Distribuído

Sistema Aberto:

- um sistema que pode ser estendido em termos de componentes ou funcionalidades sem perturbação da operações existentes

A Web é um sistema aberto

Facilidade de publicação de documentos

Características de um Sistema Distribuído

<i>Date</i>	<i>Computers</i>	<i>Web servers</i>	<i>Percentage</i>
1993, July	1,776,000	130	0.008
1995, July	6,642,000	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
2003, July	~200,000,000	42,298,371	21
2005, July	353,284,187	67,571,581	19

Características de um Sistema Distribuído

Internet

comunicação baseada em normas ou standards (IP, TCP, UDP)

A Web baseia-se fundamentalmente em 3 normas:

- HTML
- URL
- HTTP

esquema : localização



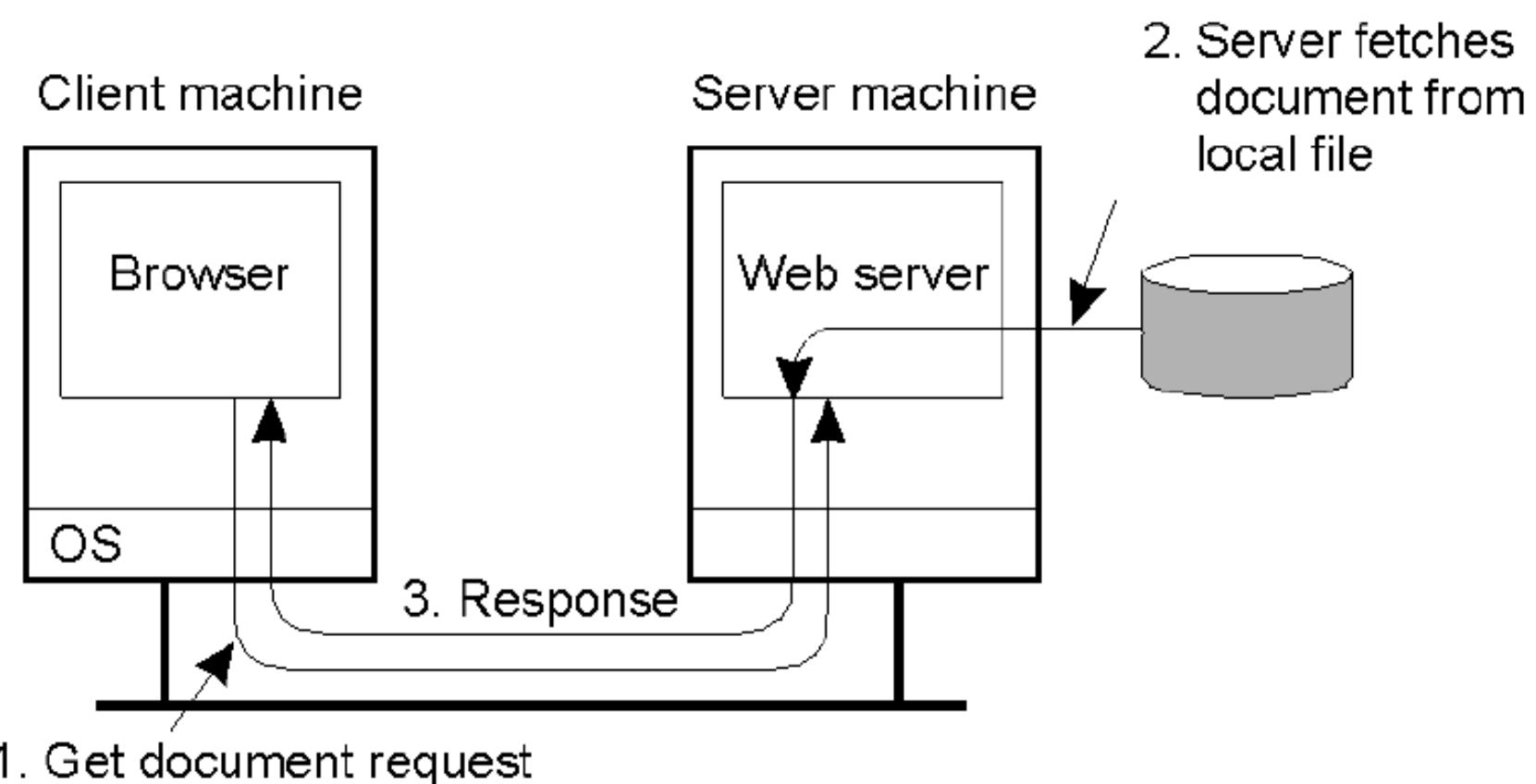
```
http:// host[:port][/:path][?arguments]  
ftp://[user[:password@]]host[:port][/:path]  
file://host/path  
mailto:aluno@uevora.pt
```

Outros:

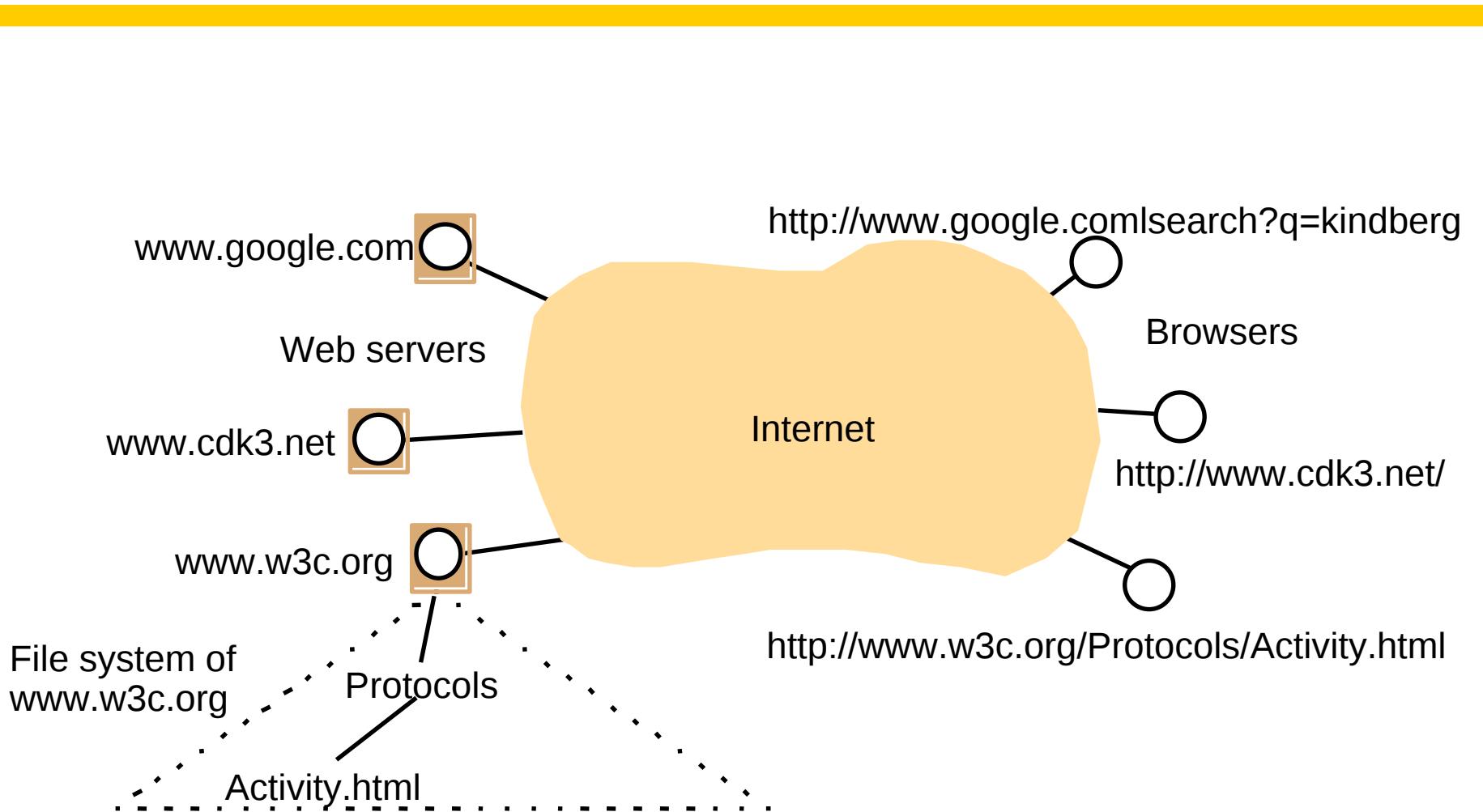
- server side: CGI, Servlets
- client side: javascript, applet
- XML, XSL, JSON
- Semantic Web**: RDF, OWL

Características de um Sistema Distribuído

Web: fluxo de dados

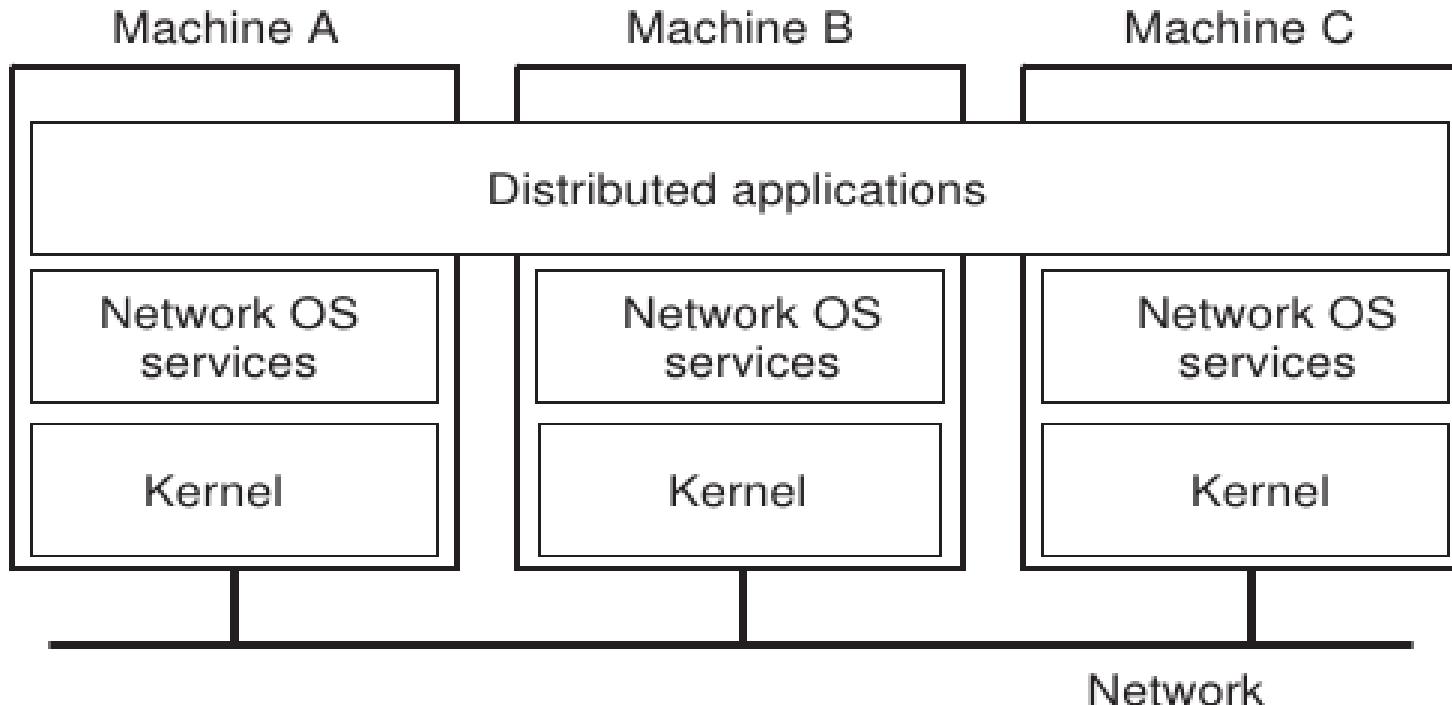


Sistema Distribuído: normas sobre os nomes



Aplicações Distribuídas: 1

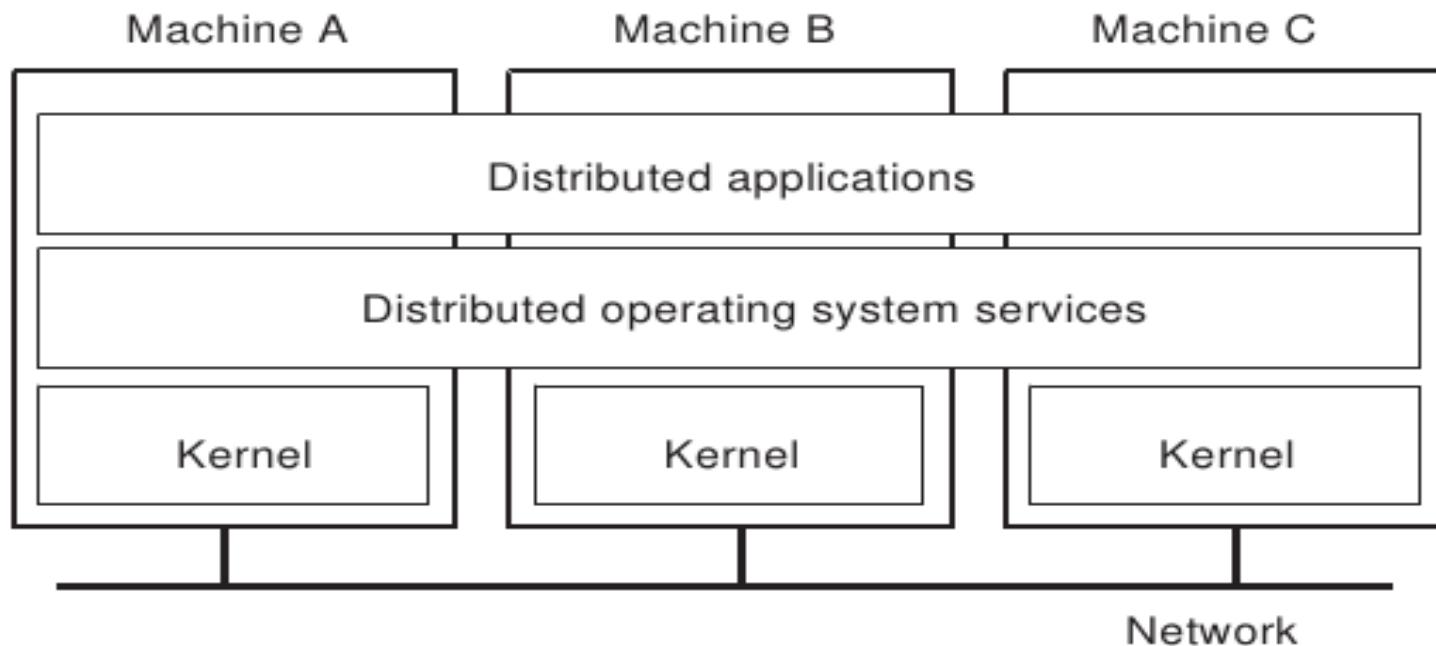
- Diretamente sobre os serviços de rede do Sistema Operativo
 - distribuição **explicitamente controlada** pelo utilizador...



Aplicações Distribuídas: 2

- Sobre um Sistema Operativo Distribuído*

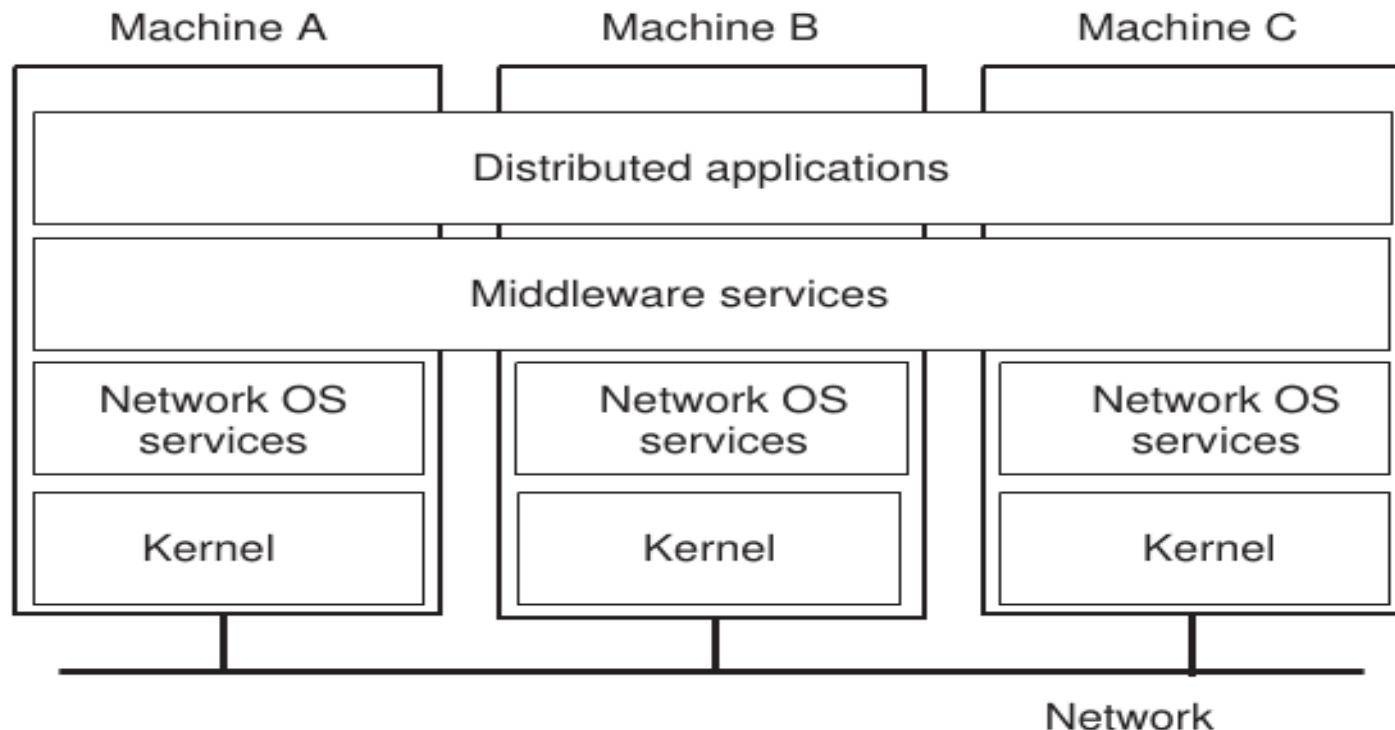
- * exemplo: *Amoeba*
- hardware homogéneo, transparência



Aplicações Distribuídas: 3

- Aplicações Distribuídas sobre o *Middleware*

- Maior abstração
- Permite utilização de diferentes máquinas, de fácil acesso
- Esconde heterogeneidade



Características de um Sistema Distribuído

Questões a considerar ao desenvolver aplicações e serviços em SD:

- heterogeneidade
- abertura
- segurança
- escalabilidade
- resolução de falhas ou problemas
- concorrência
- transparência

Características de um Sistema Distribuído

preocupações em SD: **heterogeneidade**

- o SD pode conter elementos de natureza diferente:
 - sistemas operativos
 - redes de computadores
 - hardware
 - linguagens de programação
 - algoritmos de diferentes programadores, com protocolos próprios
- *Middleware*

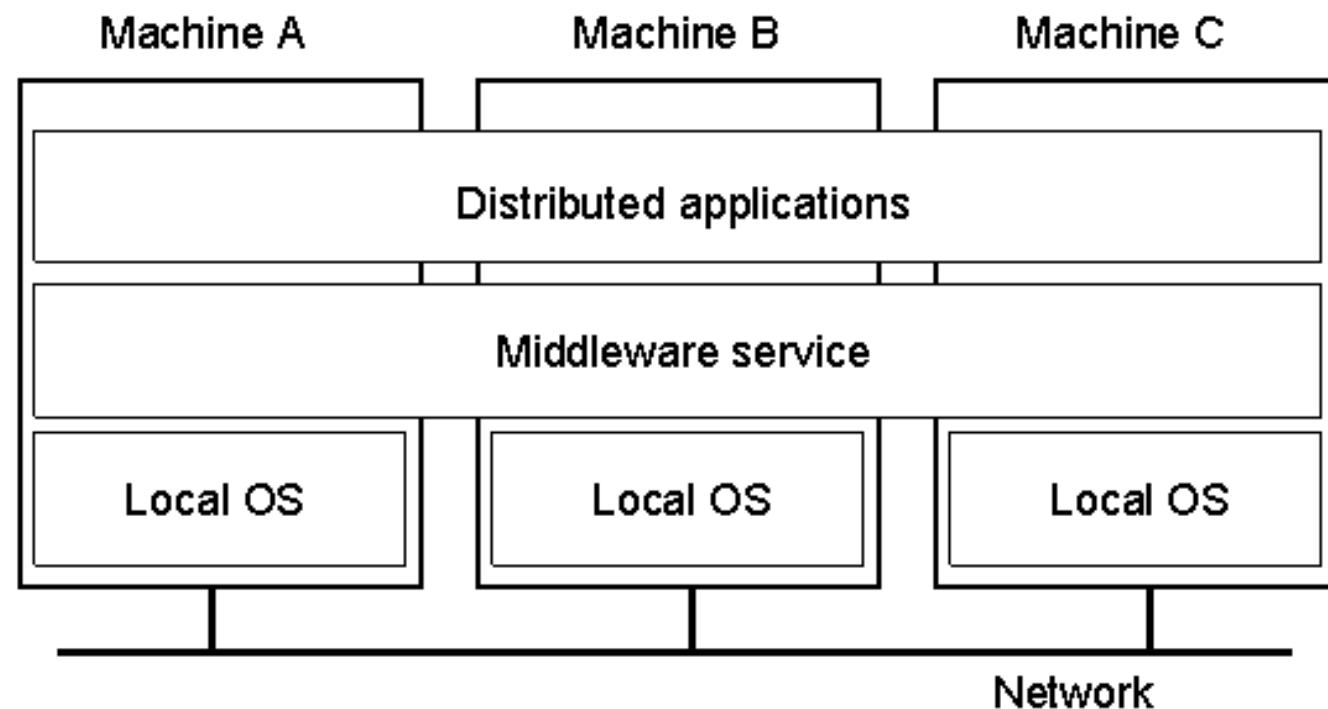
uma camada de software que fornece uma abstração, esconde a heterogeneidade dos vários componentes (hardware, rede, SO, LP) e oferece um modelo computacional uniforme para programadores e aplicações

- ex: CORBA, Java RMI

Características de um Sistema Distribuído

preocupações em SD: **heterogeneidade**

... Ainda o **Middleware**



Características de um Sistema Distribuído

preocupações em SD: **abertura**

- a abertura num sistema caracteriza-se pela possibilidade de o estender, em termos de componentes ou funcionalidades, sem perturbação dos serviços existentes

- Necessidade: documentar as interfaces de software para disponibilizar aos *developers*

protocolos de comunicação da Internet: publicados em *request for comments* (RFCs)

Características de um Sistema Distribuído

preocupações em SD: **segurança**

Aspectos:

- › confidencialidade
- › integridade
- › autenticação
- › não repúdio

Características de um Sistema Distribuído

preocupações em SD: **escalabilidade**

Um sistema é **escalável** se permanece funcional quando há um aumento significativo no nº de recursos e no nº de utilizadores.

Aspectos relevantes do ponto de vista da escalabilidade:

- custo dos recursos (físicos ou lógicos)
- perda de performance
- prevenir o esgotamento de recursos (ex: ipV4)
- evitar afunilamentos ou *bottleneck*

Características de um Sistema Distribuído

preocupações em SD: **resolução de falhas ou problemas**

- As falhas em SD são parciais (de uma ou várias componentes)
- detetar falhas
- tolerância a falhas, esconder falhas, se possível (associado à replicação)
- recuperação de um estado de erro (*rollback*)
- redundância

Características de um Sistema Distribuído

preocupações em SD: **concorrência**

- execução de várias tarefas em simultâneo
- pode originar conflitos

Características de um Sistema Distribuído

preocupações em SD: **transparência**

- manter alguns aspectos da distribuição invisíveis para o programador ou utilizador, para que o sistema seja visto como um todo
- Porquê?
 - para que o utilizador ou programador possa focar a sua atenção na sua aplicação (cliente do sistema) sem depender de aspectos específicos da distribuição
 - modularidade

Características de um Sistema Distribuído: transparência

acesso: permitir o acesso a recursos locais e remotos com operações idênticas

localização: permitir a utilização de recursos sem o conhecimento da sua localização exata

concorrência: permitir a execução simultânea de vários processos com recursos partilhados sem que surjam interferências entre eles

replicação: utilização de múltiplas instâncias de recursos para aumentar a fiabilidade e performance, mas sem que o utilizador ou aplicações cliente tenham conhecimento das réplicas, da sua quantidade ou substituição

Características de um Sistema Distribuído: transparência

falhas: permitir o tratamento de falhas, para que utilizadores e aplicações completem a sua tarefa, independentemente da ocorrência de um problema de hardware ou software

mobilidade: permitir a mobilidade de recursos e clientes dentro de um sistema sem afetar as operações de utilizadores e programas

performance: permitir o ajuste ou reconfiguração do sistema para aumentar o desempenho à medida que as solicitações (carga) variam, de modo transparente para o utilizador

escala: permitir a expansão das componentes do sistema sem alterar a estrutura do mesmo ou os algoritmos das aplicações

Leitura complementar

<http://book.mixu.net/distsys/intro.html>

<https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/>

Cloud Computing

The concept

- ➔ **Cloud computing**
 - is the use of computing resources (hardware and software) that are delivered as a service over a network (typically the Internet).
- NIST **Cloud Computing** definition
 - “Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”
 - <http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf>

Cloud computing Essential Characteristics

On-demand self-service. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

Broad network access. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).

Resource pooling. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand.

Rapid elasticity. Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

Measured service. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

Cloud computing

- Uses Internet technologies to offer scalable and elastic services.
- The term “**elastic computing**” refers to the ability of *dynamically acquiring computing resources* and supporting a variable workload.
 - The resources used for these services can be metered and the *users can be charged only for the resources they used*.
 - **Utility computing** service provisioning model
- The maintenance and security are ensured by service providers.
- The service providers can operate more efficiently due to specialization and centralization.

Cloud computing (cont'd)

- Lower costs for the cloud service provider are passed to the cloud users.
- Data is stored:
 - closer to the site where it is used.
 - in a device and in a location-independent manner.
- The data storage strategy can increase reliability, as well as security, and can lower communication costs.

Types of clouds

- Public Cloud - the infrastructure is made available to the general public or a large industry group and is owned by the organization selling cloud services.
- Private Cloud – the infrastructure is operated solely for an organization.
- Community Cloud - the infrastructure is shared by several organizations and supports a community that has shared concerns.
- Hybrid Cloud - composition of two or more clouds (public, private, or community) as unique entities but bound by standardized technology that enables data and application portability.

The “good” about cloud computing

- Resources, such as CPU cycles, storage, network bandwidth, are **shared**.
- When multiple applications share a system, their peak demands for resources are not synchronized thus, *multiplexing leads to a higher resource utilization*.
- Resources can be **aggregated** to support data-intensive applications.
- Data sharing facilitates collaborative activities. Many applications require multiple types of analysis of shared data sets and multiple decisions carried out by groups scattered around the globe.

More “good” about cloud computing

- Eliminates the initial investment costs for a private computing infrastructure and the maintenance and operation costs.
- Cost reduction: concentration of resources creates the opportunity to **pay as you go** for computing.
- Elasticity: the ability to accommodate workloads with very large peak-to-average ratios.
- User convenience: **virtualization** allows users to operate in familiar environments rather than in idiosyncratic ones.

Why cloud computing could be successful when other paradigms have failed?

- It is in a better position to exploit recent advances in software, networking, storage, and processor technologies promoted by the same companies who provide cloud services.
- It is focused on enterprise computing; its adoption by industrial organizations, financial institutions, government, and so on could have a huge impact on the economy.
- A cloud consists of a homogeneous set of hardware and software resources.
- The resources are in a single administrative domain (AD). Security, resource management, fault-tolerance, and quality of service are less challenging than in a heterogeneous environment with resources in multiple ADs.

Computing Paradigm Concepts / Distinctions

Centralized Computing

All computer resources are centralized in one physical system.

Parallel Computing

All processors are either tightly coupled with central shared memory or loosely coupled with distributed memory

Distributed Computing

Field of CS/CE that studies distributed systems. A distributed system consists of multiple autonomous computers, each with its own private memory, communicating over a network.

Cloud Computing

An Internet cloud of resources that may be either centralized or decentralized.

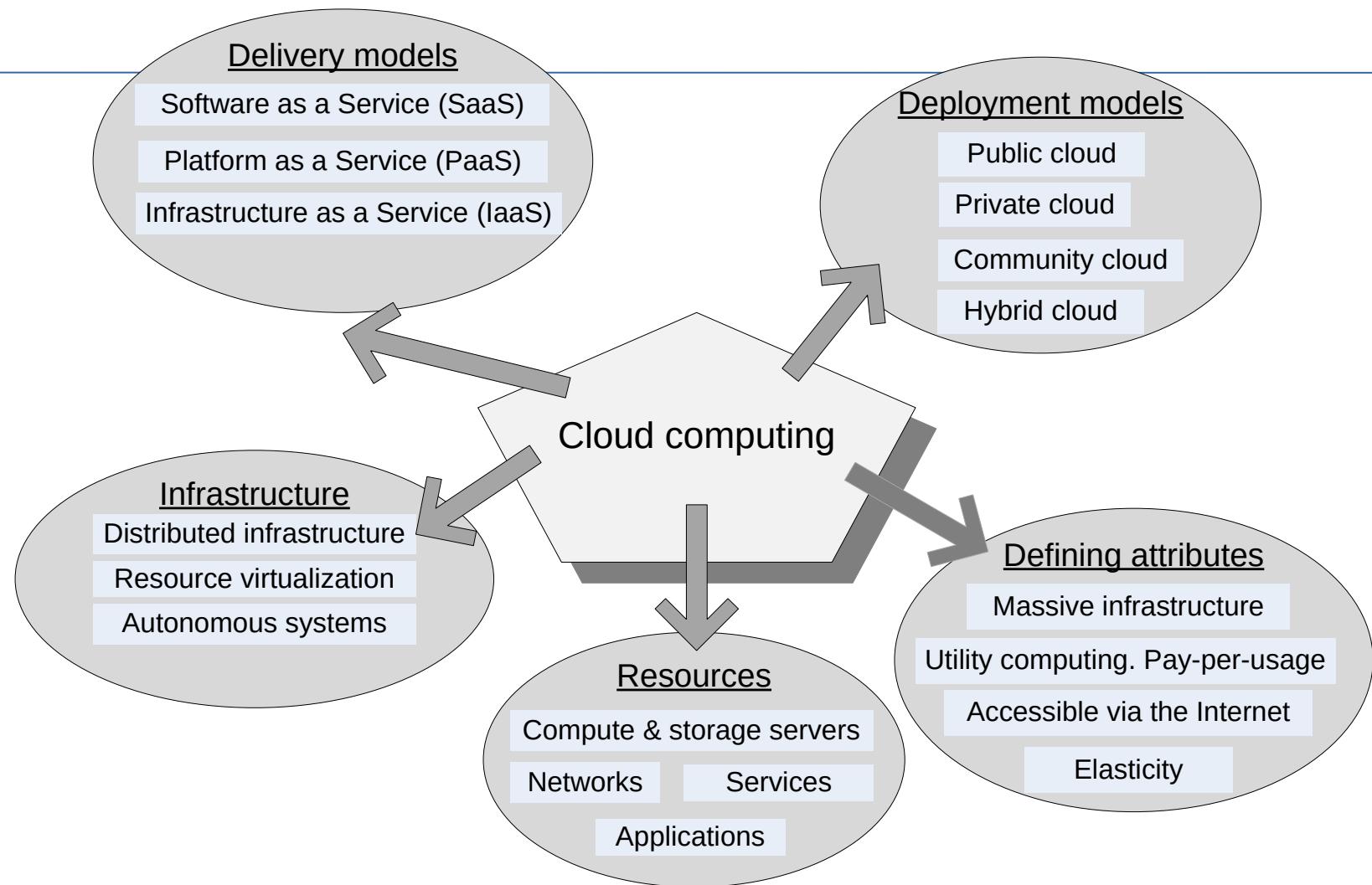
The cloud applies to parallel or distributed computing **or both**. Clouds may be built from **physical or virtualized resources**.

Challenges for cloud computing

- Availability of service; what happens when the service provider cannot deliver?
- Diversity of services, data organization, user interfaces available at different service providers limit user mobility; once a customer is hooked to one provider it is hard to move to another.
Standardization efforts at NIST!
- Data confidentiality and auditability, a serious problem.
- Data transfer bottleneck; many applications are data-intensive.

More challenges

- Performance unpredictability, one of the consequences of resource sharing.
 - How to use resource virtualization and performance isolation for QoS guarantees?
 - How to support elasticity, the ability to scale up and down quickly?
- Resource management; are self-organization and self-management the solution?
- Security and confidentiality; major concern.
- Addressing these challenges provides good research opportunities!!



Software-as-a-Service (SaaS)

- Applications are supplied by the service provider.
- The user does not manage or control the underlying cloud infrastructure or individual application capabilities.
- Services offered include:
 - Enterprise services such as: workflow management, group-ware and collaborative, supply chain, communications, digital signature, customer relationship management (CRM), desktop software, financial management, geo-spatial, and search.
 - Web 2.0 applications such as: metadata management, social networking, blogs, wiki services, and portal services.
- Not suitable for real-time applications or for those where data is not allowed to be hosted externally.
- Examples: Gmail, Google search engine.

Platform-as-a-Service (PaaS)

- Allows a cloud user to deploy consumer-created or acquired applications using programming languages and tools supported by the service provider.
- The user:
 - Has control over the deployed applications and, possibly, application hosting environment configurations.
 - Does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage.
- Not particularly useful when:
 - The application must be portable.
 - Proprietary programming languages are used.
 - The hardware and software must be customized to improve the performance of the application.

Infrastructure-as-a-Service (IaaS)

- The user is able to deploy and run arbitrary software, which can include operating systems and applications.
- The user does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of some networking components, e.g., host firewalls.
- Services offered by this delivery model include: server hosting, Web servers, storage, computing hardware, operating systems, virtual instances, load balancing, Internet access, and bandwidth provisioning.

Cloud activities

- Service management and provisioning including:
 - Virtualization.
 - Service provisioning.
 - Call center.
 - Operations management.
 - Systems management.
 - QoS management.
 - Billing and accounting, asset management.
 - SLA management.
 - Technical support and backups.

Cloud activities (cont'd)

- Security management including:
 - ID and authentication.
 - Certification and accreditation.
 - Intrusion prevention.
 - Intrusion detection.
 - Virus protection.
 - Cryptography.
 - Physical security, incident response.
 - Access control, audit and trails, and firewalls.

Cloud activities (cont'd)

- Customer services such as:
 - Customer assistance and on-line help.
 - Subscriptions.
 - Business intelligence.
 - Reporting.
 - Customer preferences.
 - Personalization.
- Integration services including:
 - Data management.
 - Development.

Ethical issues

- Paradigm shift with implications on computing ethics:
 - The control is relinquished to third party services.
 - The data is stored on multiple sites administered by several organizations.
 - Multiple services interoperate across the network.
- Implications
 - Unauthorized access.
 - Data corruption.
 - Infrastructure failure, and service unavailability.

De-perimeterisation

- Systems can span the boundaries of multiple organizations and cross the security borders.
- The complex structure of cloud services can make it difficult to determine who is responsible in case something undesirable happens.
- Identity fraud and theft are made possible by the unauthorized access to personal data in circulation and by new forms of dissemination through social networks and they could also pose a danger to cloud computing.

Privacy issues

- Cloud service providers have already collected petabytes of sensitive personal information stored in data centers around the world.
 - The acceptance of cloud computing therefore will be determined by privacy issues addressed by these companies and the countries where the data centers are located.
- Privacy is affected by cultural differences; some cultures favor privacy, others emphasize community.
 - This leads to an ambivalent attitude towards privacy in the Internet which is a global system.

Cloud vulnerabilities

- Clouds are affected by malicious attacks and failures of the infrastructure, e.g., power failures.

- Such events can affect the Internet domain name servers and prevent access to a cloud or can directly affect the clouds:
 - in 2004 an attack at Akamai caused a domain name outage and a major blackout that affected Google, Yahoo, and other sites.
 - in 2009, Google was the target of a denial of service attack which took down Google News and Gmail;
 - in 2012 lightning caused a prolonged down time at Amazon.

Credits, references and reading material

- *Cloud Computing: Theory and Practice*
Dan C. Marinescu
Chapters 1, 2
- *Cloud Computing: Principles and Paradigms*
R. Buyya, J. Broberg, and A. Goscinski
Wiley Press, 2011
- *Distributed and Cloud Computing*
K. Hwang, G. Fox and J. Dongarra
Morgan Kaufmann, 2012



Sistemas Operativos II

Web Services

Web Services

- camada de software para facilitar a interação entre cliente e servidor, tornando-a mais rica e mais estruturada.
- incluem uma API que permite aceder a serviços remotos através da rede
- independentemente da linguagem das aplicações cliente e servidor, os pedidos e respostas são *usualmente* codificados numa **Representação Externa de Dados (RED)** em **XML** e transmitidos sobre HTTP
 - Outra possível RED: **JSON**
- são identificados por um URI (URL ou URN)

Interface e formato das mensagens

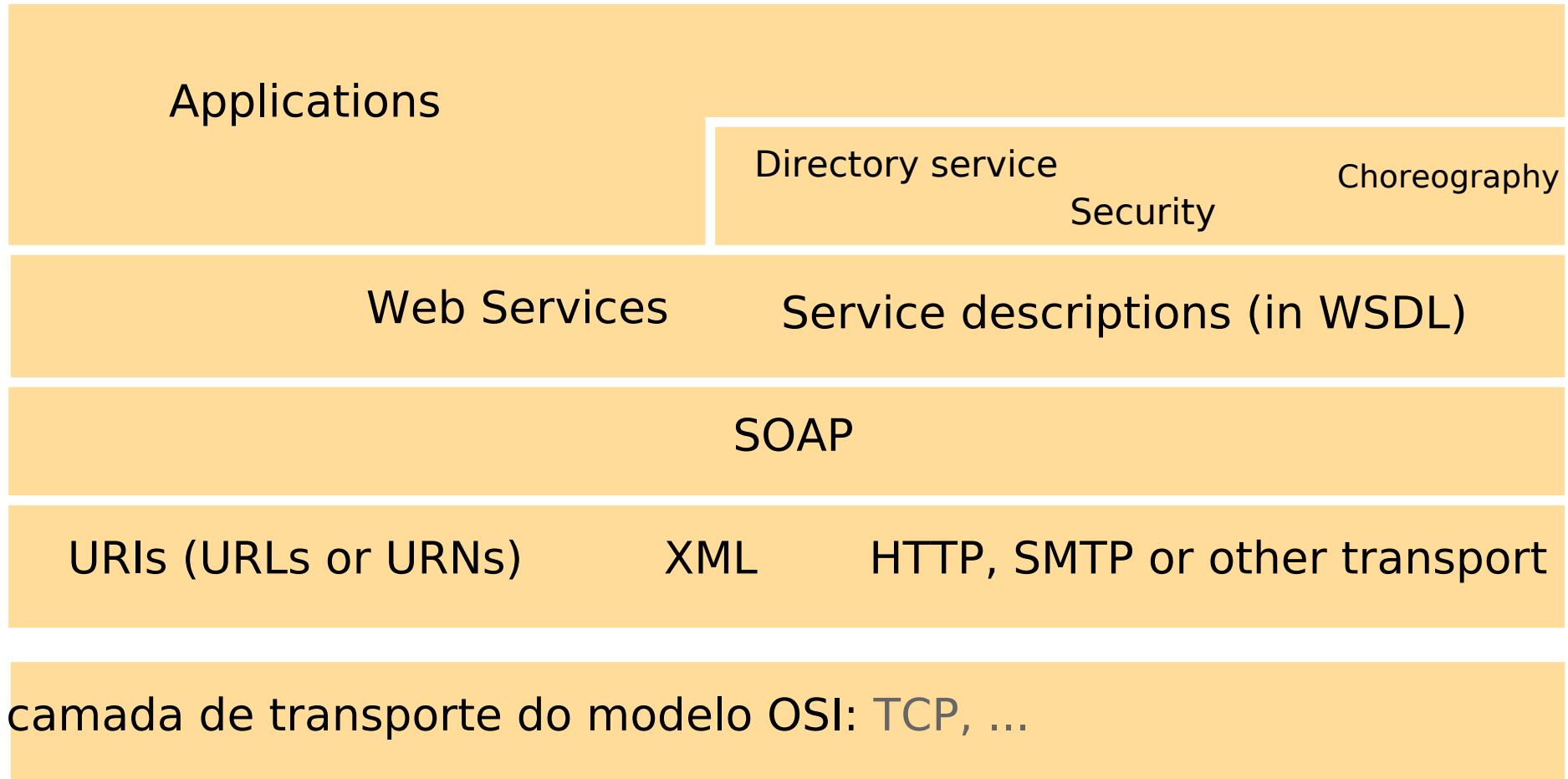
Service Description

- Tal como no CORBA e Java RMI, há uma descrição para a **interface** do web service. Há também a especificação do protocolo de codificação e comunicação das mensagens e a localização do Web Service (URL ou URN)
- A IDL usada é a *Web Services Description Language* (WSDL)

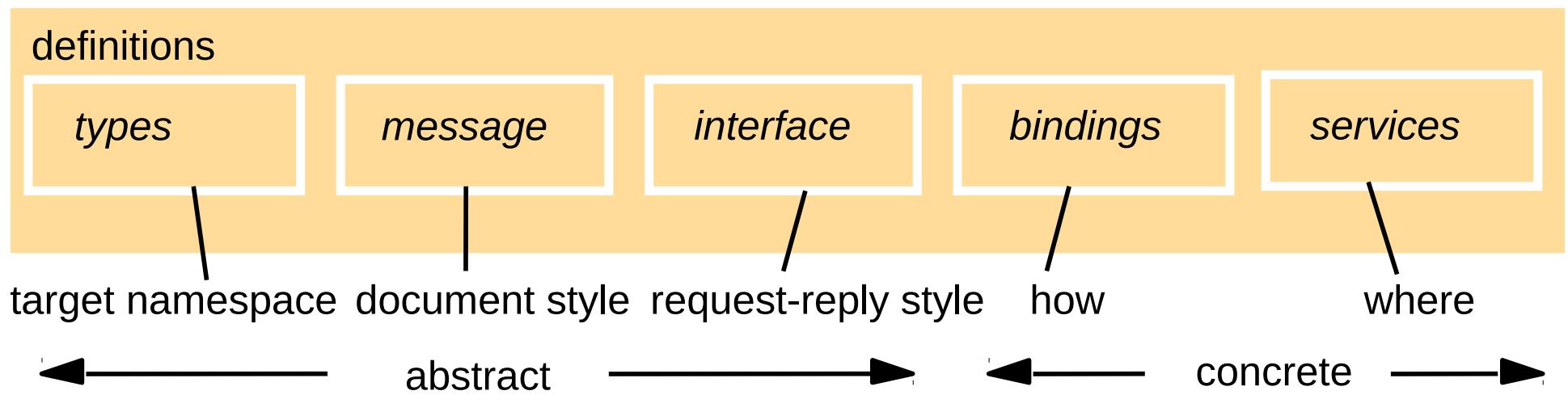
SOAP - Simple Object Access Protocol

- extensão de XML-RPC
- é um protocolo para troca de mensagens, usualmente sobre HTTP, que trata do correto encapsulamento dos dados em XML
- possíveis protocolos para envio das mensagens HTTP, SMTP, TCP, UDP
- não levanta problemas na presença de firewalls

Web Services: camada de software usada pelas aplicações



Principais componentes da *Service Description WSDL*



WSDL para Request e Reply para a operação newShape()

```
message name = "ShapeList_newShape"
```

```
part name = "GraphicalObject_1"  
type = "ns:GraphicalObject "
```

```
message name = "ShapeList_newShapeResponse"
```

```
part name= "result"  
type= "xsd:int"
```

tns – target namespace xsd – XML schema definitions

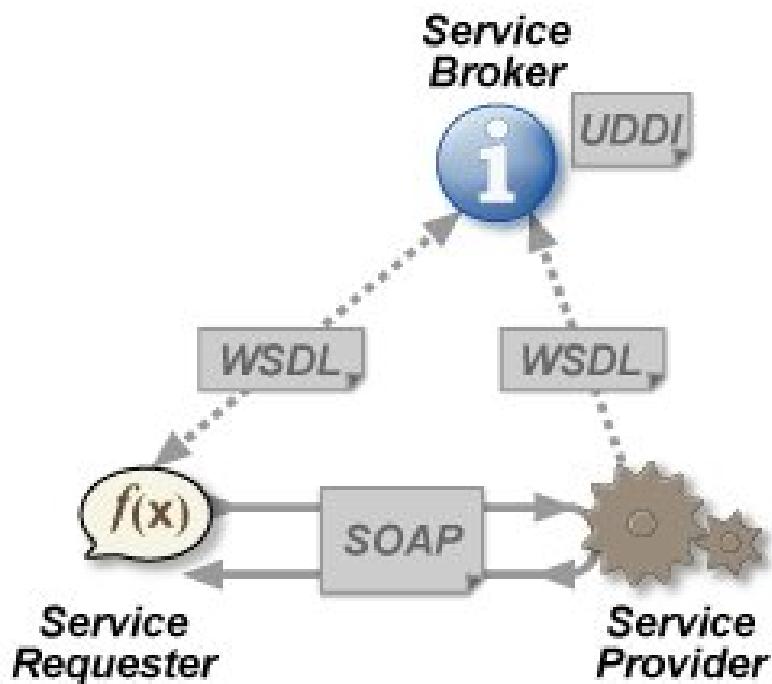
```
operation name = "newShape"  
pattern = In-Out
```

```
input message = tns:ShapeList_newShape
```

```
output message = "tns:ShapeList_newShapeResponse"
```

Web Services: arquitetura

- tal como em Java RMI, o cliente do Web Service poderá consultar a descrição do serviço de nomes ou de diretoria
- **UDDI:Universal Description Discovery and Integration**
 - protocolo para publicar e pesquisar meta-informação sobre web services... permite que uma aplicação **descubra e use** um web service em *runtime*.



Web Services: podem ser encadeados, formando um serviço mais complexo e abrangente

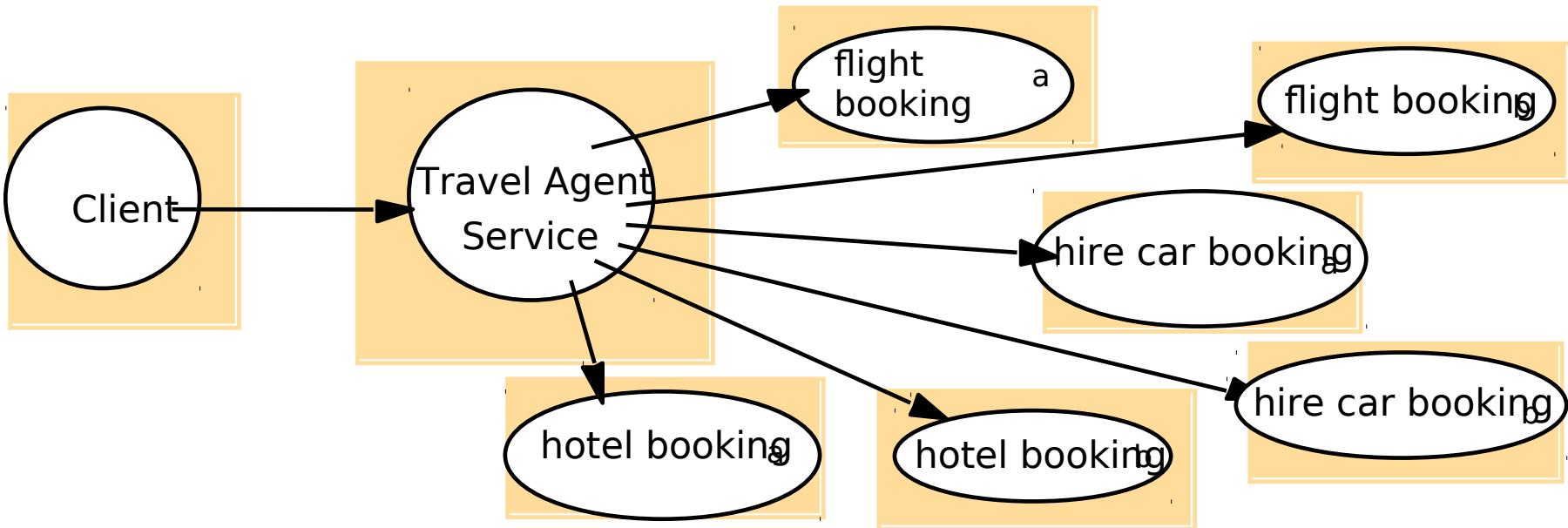


Figure 19.2 The 'travel agent service' combines other web services

Web Services

- servem de suporte à computação distribuída via internet, facilitando a cooperação de aplicações baseadas em diferentes linguagens
- os detalhes de SOAP e XML são usualmente escondidos por APIs (Java, Perl, Python, C++). A *service description* pode ser usada para gerar as rotinas de *marshalling* e *unmarshalling* de forma automática.
- diferença relativamente ao Object Model Distribuído: um *web service* é assegurado por um único objeto
 - o garbage collection neste caso não é relevante
 - a referência remota para o objeto não é relevante (ele é o único associado ao serviço)
 - Para lá do Web Service, podem existir muitos objetos, para funcionalidade de apoio...
- Uso de XML em SOAP e nos dados:
 - vantagem: mais legível, por humanos
 - desvantagem: processamento mais lento que formatos binários

Figure 19.3 SOAP message in an envelope

- | cada mensagem é colocada num *envelope* SOAP
- | mensagem SOAP pode servir para:
 - | enviar uma notificação
 - | comunicação cliente-servidor (protocolo Request-Reply)

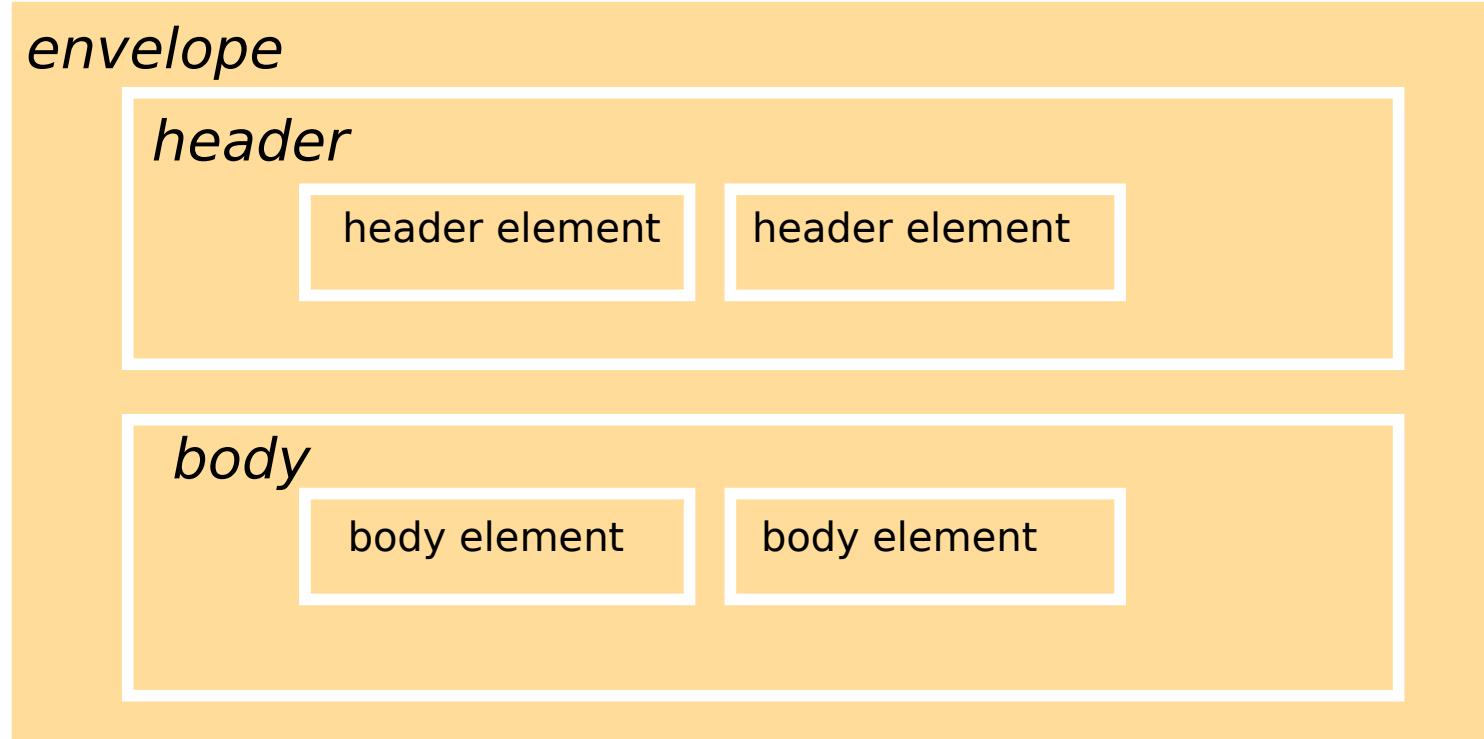


Figure 19.4 Example of a simple request (without headers)

env:envelope xmlns:env =namespace URI for SOAP envelopes

env:body

m:exchange

 xmlns:m = namespace URI of the service description

m:arg1

Hello

m:arg2

World

each XML element is represented by a shaded box with its name in italic followed by any attributes and its content

Note-se a definição de XML Namespaces (env e m)

Os SOAP Headers destinam-se a essencialmente a componentes de software intermediários...

Figure 19.5 Example of a reply corresponding to the request in Figure 19.4

env:envelope

xmlns:env = namespace URI for SOAP envelope

env:body

m:exchangeResponse

xmlns:m = namespace URI for the service description

m:res1

World

m:res2

Hello

Figure 19.6 Use of HTTP POST Request in SOAP client-server communication

POST /examples/stringer ← endpoint address

Host: www.cdk4.net

Content-Type: application/soap+xml

Action: http://www.cdk4.net/examples/stringer#exchange ← action

HTTP header

<env:envelope xmlns:env= namespace URI for SOAP envelope

<env:header> </env:header>

<env:body> </env:body>

</env:Envelope>

Soap message

SOAP em Java

- **JAX-RPC** é uma API que permite construir Web Services e respetivos clientes
- **JAX-WS** significa Java API for XML Web Services e inclui:
 - **JAX-WS 2.0** - é uma extensão do JAX-RPC 1.0.
 - Java Architecture for XML Binding (JAXB)
 - SOAP with Attachments API for Java (SAAJ)
 - ... faz o mapeamento entre tipos Java e definições XML a usar nas mensagens
- A API do JAX-WS esconde a complexidade do protocolo SOAP
 - O programador não precisa gerar ou fazer o parse explícito de mensagens SOAP
- Web Service **Endpoint** é a implementação do Web Service
 - Fica no servidor

SOAP em Java

- JAX-RPC e JAX-WS têm ferramentas de desenvolvimento de Web Services com algumas diferenças
- Por exemplo para o JAX-WS, a implementação do *endpoint* deve obedecer a:
 - *It must carry a javax.jws.WebService annotation*
 - *It may extend java.rmi.Remote either directly or indirectly.*
 - *Any of its methods may carry a javax.jws.WebMethod annotation*
 - *All of its methods may throw java.rmi.RemoteException in addition to any service-specific exceptions.*
 - *All method parameters and return types must be compatible with the JAXB 2.0 Java to XML Schema mapping definition.*
 - *The implementing class must not be declared final and must not be abstract.*
 - *The business methods of the implementing class must be public, and must not be declared static or final.*
 - *Business methods that are exposed to web service clients must be annotated with javax.jws.WebMethod.*
 - *A method parameter or return value type must not implement the java.rmi.Remote interface either directly or indirectly ***
- *** - idem para JAX-RPC, onde as classes desses argumentos ou resultado devem possuir um construtor por defeito, público.*

Figure 19.7 Java web service interface ShapeList

Exemplo do Livro, para JAX-RPC. Aqui, isto é importante

```
import java.rmi.*;  
  
public interface ShapeList extends Remote {  
    int newShape(GraphicalObject g) throws RemoteException;  
    int numberOfShapes() throws RemoteException;  
    int getVersion() throws RemoteException;  
    int getGOVersion(int i) throws RemoteException;  
    GraphicalObject getAllState(int i) throws RemoteException;  
}
```

Figure 19.8 Java implementation of the ShapeList “server”

```
import java.util.Vector;  
  
public class ShapeListImpl implements ShapeList {  
    private Vector theList = new Vector();  
    private int version = 0;  
    private Vector theVersions = new Vector();  
  
    public int newShape(GraphicalObject g) throws RemoteException{  
        version++;  
        theList.addElement(g);  
        theVersions.addElement(new Integer(version));  
        return theList.size();  
    }  
    public int numberOfShapes(){}
    public int getVersion() {}
    public int getGOVersion(int i){ }
    public GraphicalObject getAllState(int i) {}  
}
```

Figure 19.9 Java implementation of the ShapeList client

```
package staticstub;
import javax.xml.rpc.Stub;
public class ShapeListClient {
    public static void main(String[] args) { /* pass URL of service */
        try {
            Stub proxy = createProxy();
            proxy._setProperty
                (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
            ShapeList aShapeList = (ShapeList)proxy;
            GraphicalObject g = aShapeList.getAllState(0);
        } catch (Exception ex) { ex.printStackTrace(); }
    }
    private static Stub createProxy() {
        return
            (Stub) (new MyShapeListService_Impl().getShapeListPort());
    }
}
```

Web Services em Java: Servidor e descrição

Assumindo que se começa por desenvolver a *Service Interface* e a sua implementação (*endpoint*)

- há ferramentas para gerar automaticamente o Skeleton e a descrição do serviço em WSDL
- o serviço vai correr num Servlet Container (exemplo: Apache Tomcat)
- Depois de preparado, o web service deve estar num ficheiro “.war”, que pode ser *deployed* (instalado) no *Servlet Container*
 - o dispatcher do *servlet container* identifica a operação no pedido
 - pelo header http Action
 - e invoca o método apropriado no respetivo Skeleton
- É possível gerar a classe do Proxy, para o cliente, em runtime, a partir da *service description*... mas não é sempre assim.

Segurança no XML

funcionalidade para assinar, cifrar e manipular chaves

- Para determinadas garantias de segurança, não basta proteger o canal... o próprio documento tem de incorporar alguma meta- informação (assinatura, informação de chaves...)
 - para ser validada posteriormente à cessação do canal de comunicação
- o XML permite várias formas sintácticas para os mesmos dados
 - a assinatura digital do XML é precedida de uma conversão para *Canonical XML*
 - *representação normalizada e serializada de XML*

Figure 19.16 Algorithms required for XML signature

<i>Type of algorithm</i>	<i>Name of algorithm</i>	<i>Required</i>	<i>reference</i>
Message digest	SHA-1	Required	Section 7.4.3
Encoding	base64	Required	[Freed and Borenstein 1996]
Signature (asymmetric)	DSA with SHA-1	Required	[NIST 1994]
MAC signature (symmetric)	RSA with SHA-1	Recommended	Section 7.3.2
Canonicalization	HMAC-SHA-1	Required	Section 7.4.2 and Krawczyk <i>et al.</i> [1997]
	Canonical XML	Required	Page 810

Figure 19.17 Algorithms required for encryption(the algorithms in Figure 19.16 are also required)

<i>Type of algorithm</i>	<i>Name of algorithm</i>	<i>Required</i>	<i>reference</i>
Block cipher	TRIPLEDES,	required	Section 7.3.1
	AES-128		
	AES-256		
Encoding	AES-192	optional	
	base64	required	[Freed and Borenstein 1996]
Key transport	RSA-v1.5,	required	Section 7.3.2
	RSA-OAEP		[Kaliski and Staddon 1998]
Symmetric key wrap (signature by shared key)	TRIPLEDES	required	[Housley 2002]
	KeyWrap,		
	AES-128 KeyWrap,		
	AES 256KeyWrap		
Key agreement	AES-192 KeyWrap	optional	
	Diffie-Hellman	optional	[Rescorla, 1999]

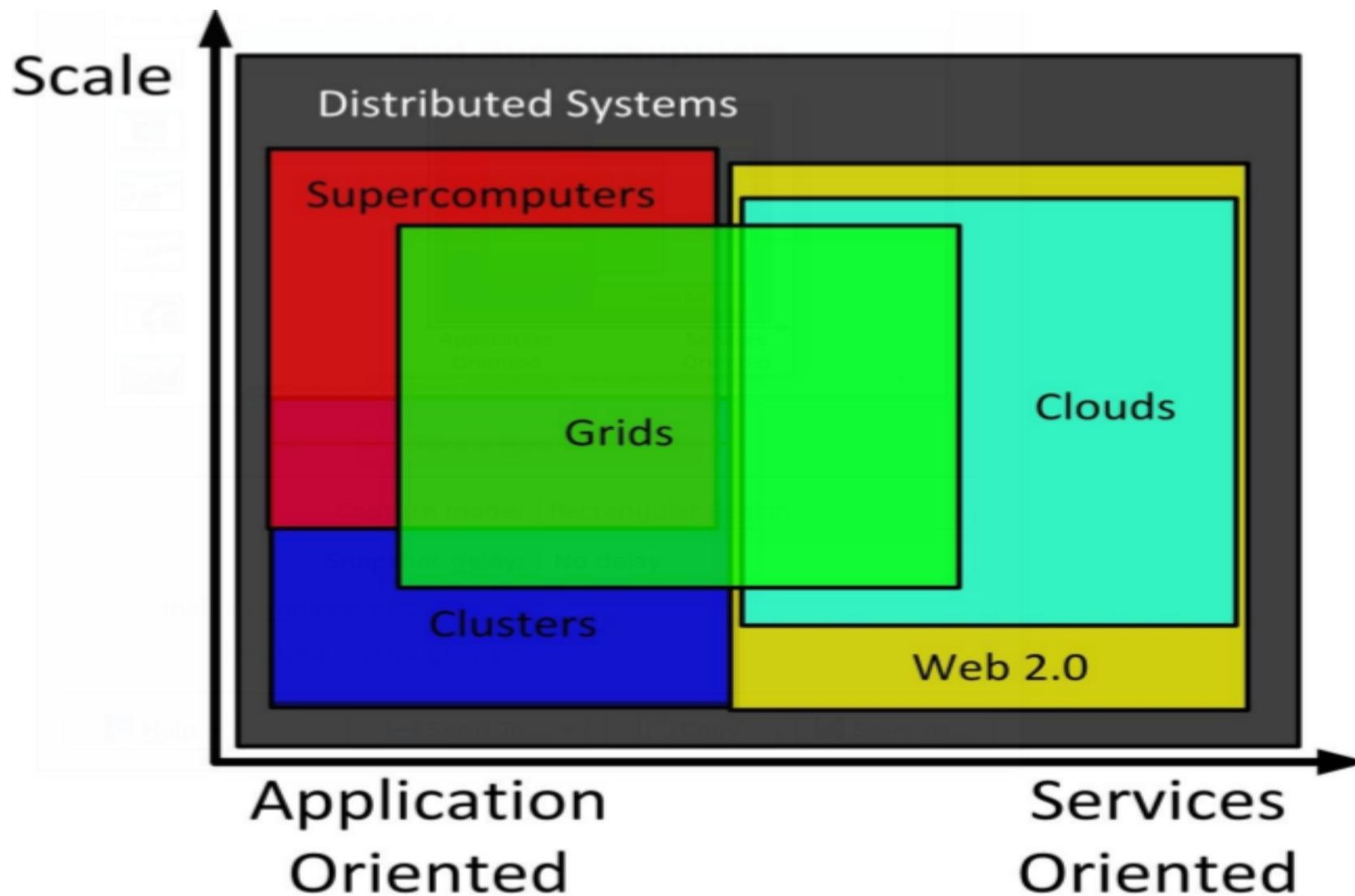
Alguns Web Services da Amazon

<i>Web service</i>	<i>Description</i>
Amazon Elastic Compute Cloud (EC2)	Web-based service offering access to virtual machines of a given performance and storage capacity
Amazon Simple Storage Service (S3)	Web-based storage service for unstructured data
Amazon Simple DB	Web-based storage service for querying structured data
Amazon Simple Queue Service (SQS)	Hosted service supporting message queuing (as discussed in Chapter 6)
Amazon Elastic MapReduce	Web-based service for distributed computation using the MapReduce model (introduced in Chapter 21)
Amazon Flexible Payments Service (FPS)	Web-based service supporting electronic payments

GRID Computing

- *middleware* desenhado para permitir e optimizar a partilha de recursos (computadores, dados, sensores, software) em larga escala
- usualmente os utilizadores destes sistemas (cientistas, engenheiros) colaboram para alcançar um objetivo comum, como um estudo que requer o processamento de grandes quantidades de dados, por exemplo.
- os recursos estão alojados em computadores de diferentes plataformas, com ambiente heterogéneo
- o *middleware* **GRID** pode assentar em **Web Services**

Grid & Cloud



Grid & Cloud

Grid

- Usualmente heterogéneas (mas não necessariamente)
- Lidar com enorme volume de dados ou processamentos complexos
- Geograficamente dispersas
- Exemplos
 - Open Science Grid (OSG) - <http://www.opensciencegrid.org/>
 - LHC Computing Grid (CERN)
 - Middleware: Globus Toolkit

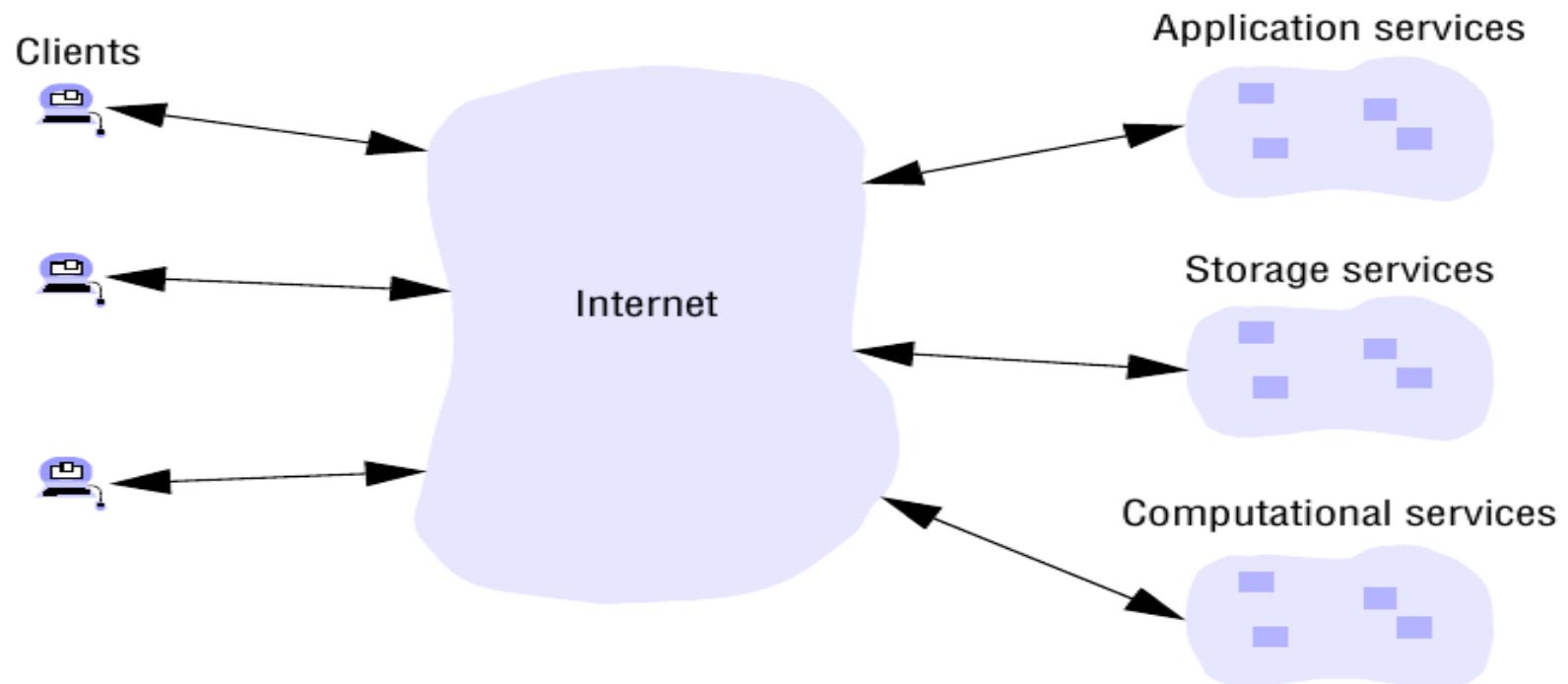
Cloud

- Paradigma “emergente” de computação distribuída de larga escala
 - Motivações:
 - Uso mais geral (muitos ou poucos dados); maior leque de utilizadores
 - vocação para os serviços
 - **Virtualização**
 - Alocação de recursos dinâmica em função das necessidades
 - Exemplos:
 - Amazon, GoogleApps, Windows Azure

Cloud Computing

Cloud é um serviço de armazenamento e/ou computação baseado na Internet. Reduz a necessidade de armazenamento, software e capacidade de processamento do lado dos terminais junto do utilizador.

Cloud computing



Arquitetura REST

- REST - *Representational State Transfer*
 - É uma arquitetura para interação em sistemas distribuídos
 - Surgiu em 2000, da tese de doutoramento de Roy Thomas Fielding
- Objetos têm estado, cuja representação é transportada por pedidos HTTP
 - PUT; POST; GET; DELETE
 - Cada pedido tem um significado próprio (criar, alterar, consultar e apagar objetos)
- Em comparação com os SOAP based Web Services:
 - Mais leve
 - Ainda intelegrável
 - Formato dos dados pode ser JSON ou XML (mantém flexibilidade)

Arquitetura REST

- Analogia com SOAP:
 - mensagem de consulta de dados sobre o utilizador nº 12345
- SOAP WS:

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:body pb="http://www.acme.com/phonebook">
    <pb:GetUserDetails>
      <pb:UserID>12345</pb:UserID>
    </pb:GetUserDetails>
  </soap:Body>
</soap:Envelope>
```

- REST:
 - <http://www.acme.com/phonebook/UserDetails/12345>
 - O url inclui os parâmetros da consulta

Sistemas Operativos II

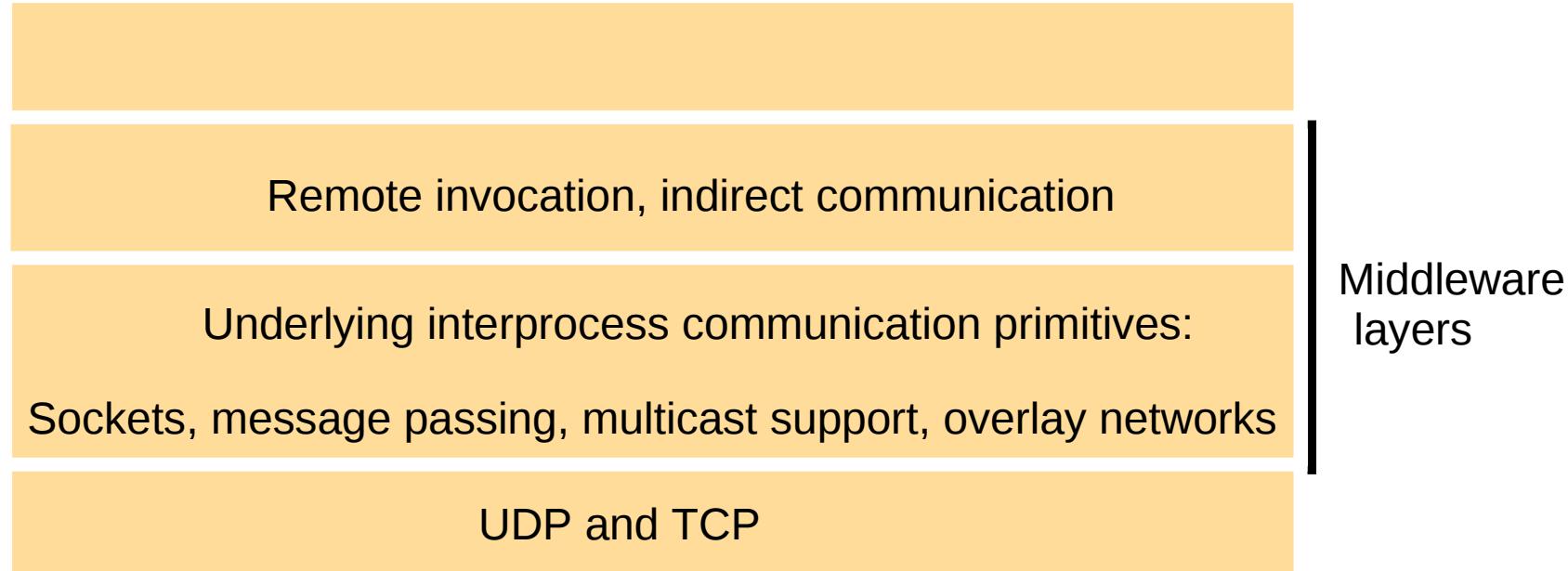
Objetos Distribuídos
e
Invocação Remota

Introdução

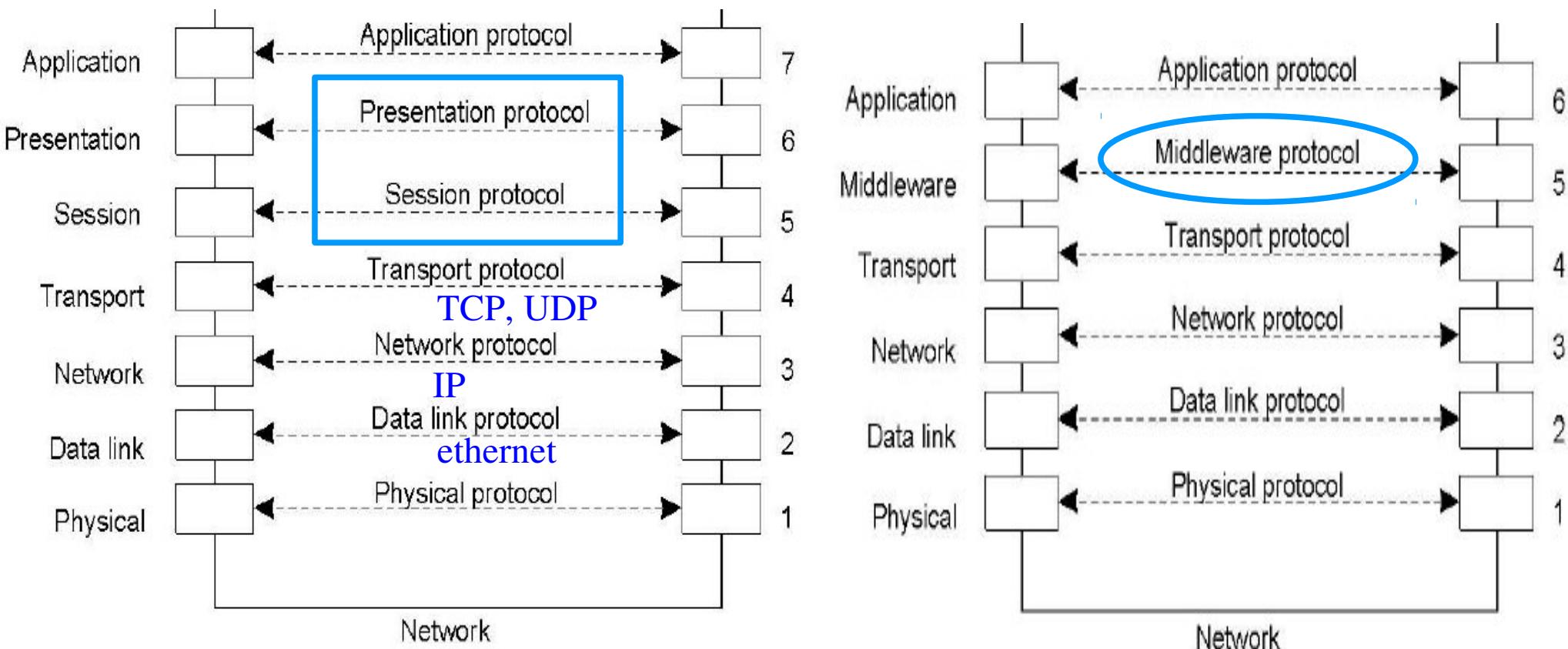
- ◆ modelos de programação para aplicações distribuídas
 - ◆ compostas por vários programas que cooperam e são executados em distintos **processos** simultâneos
- ◆ necessidade de um processo invocar uma operação num outro processo
- ◆ MODELOS:
 - ◆ *Remote Procedure Call (RPC)*
 - ◆ programa cliente executa um procedimento do lado do servidor
 - ◆ *Remote Method Invocation (RMI) (JAVA RMI, CORBA)*
 - ◆ OO
 - ◆ um objeto pode invocar métodos de outro objeto num processo diferente
 - ◆ Modelo baseado em eventos
 - ◆ OO
 - ◆ objetos recebem notificações de eventos que acontecem num outro objeto em que estão interessados

Camada superior do *Middleware*

HOJE →



Modelo OSI e o Middleware



Abstração Middleware

- ◆ Transparência face à localização
 - ◆ o processo cliente não sabe o endereço do servidor
- ◆ Independência dos protocolos de comunicação
 - ◆ o Request-Reply pode ser implementado sobre TCP ou UDP
- ◆ Independência do Hardware
 - ◆ standards para Representação Externa de Dados escondem diferenças
- ◆ Independência dos SO
 - ◆ as abstrações de nível superior fornecidas pelo Middleware não dependem do SO
- ◆ Utilização de diferentes linguagens de programação
 - ◆ CORBA: cliente numa linguagem pode invocar métodos em servidores escritos noutra linguagem – graças à CORBA IDL

Interfaces

- ◆ Interface
 - ◆ forma de controlo sobre a comunicação entre módulos ou processos
 - ◆ especifica um formato para as mensagens
- ◆ A interface de um módulo especifica os métodos e variáveis acessíveis para outros módulos (e o restante em cada módulo fica escondido do exterior)
- ◆ Tipos de Interface:
 - ◆ em RPC: *Service Interface* (*em modelos cliente-servidor*)
 - ◆ descrição dos procedimentos disponíveis e respectivos argumentos (input/output)
 - ◆ Não se podem passar apontadores como argumentos
 - ◆ em RMI: *Remote Interface*
 - ◆ métodos de um objeto disponíveis para Inv. Remota
 - ◆ podem passar-se referências para objetos remotos (*diferente de pointers*)

IDL

- ◆ *Interface Definition Language*
 - ◆ linguagem de programação adequada para definir interfaces
- ◆ Java RMI
 - ◆ todas as aplicações programadas na mesma linguagem: **Java**
- ◆ Sistemas baseados em várias linguagens de programação
(ex: CORBA)
 - ◆ requerem uma IDL que forneça a notação para as interfaces que poderão ser usadas pelas diferentes aplicações

Exemplo de CORBA IDL

Em CORBA IDL cada argumento é marcado como “de entrada”, “de saída” ou “de entrada e saída”

```
// In file Person.idl
struct Person {
    string name;
    string place;
    long year;
};

interface PersonList {
    readonly attribute string listname;
    void addPerson(in Person p);
    void getPerson(in string name, out Person p);
    long number();
};
```

Object Model (1)

- ◆ Programa OO
 - ◆ conjunto de objetos, cada um contendo dados e métodos
 - ◆ os objetos comunicam via invocação de métodos, passando argumentos e recebendo o resultado
 - ◆ algumas linguagens (Java,C++) permitem acesso direto a algumas variáveis (num SD os dados devem ser acessados apenas pelos métodos)
- ◆ Os objetos têm uma referência (em Java: *1st class value*)
- ◆ Interfaces
 - ◆ declaração de métodos (argumentos, retorno e exceções) sem especificar a implementação
- ◆ Ações: iniciadas pelas invocações de métodos (podem alterar estado)
- ◆ Excepções: situação anómala
- ◆ *Garbage Collection*: libertar espaço ocupado por objetos quando eles já não são necessários

Objetos Distribuídos

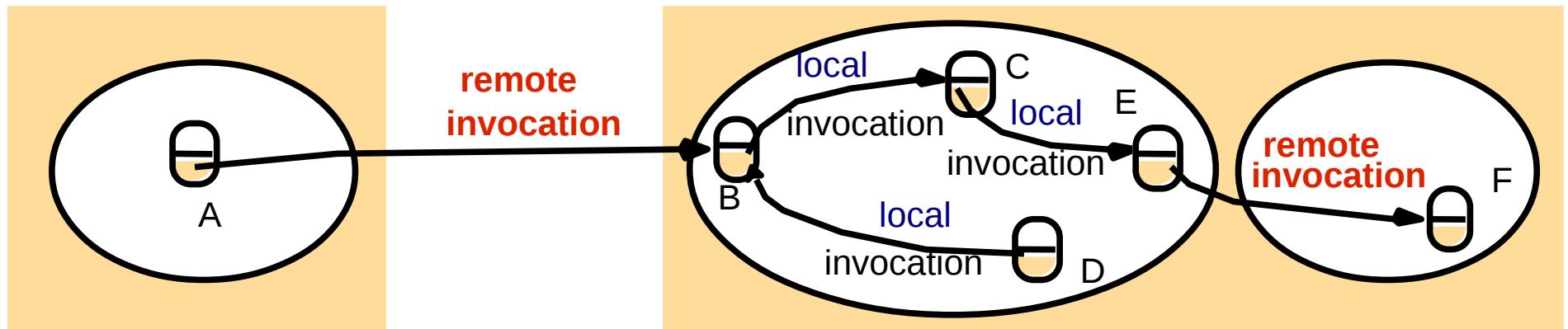
- ◆ Exemplo: arquitetura cliente-servidor
 - ◆ cliente envia um pedido ao servidor com o método que deseja invocar sobre um objeto pertencente ao servidor. O resultado é-lhe devolvido.
- ◆ Estado do Objeto: definido pelos valores das suas variáveis de instância
- ◆ quando o objeto pertence a um processo diferente, o seu estado só pode ser consultado/alterado através de métodos desse objeto
 - ◆ vantagem: facilita o mecanismo de proteção contra acessos incorretos (simultâneos/perigosos)

Object Model Distribuído

- ◆ processo
 - ◆ tem um conjunto de objetos, alguns dos quais recebem invocações locais e remotas, outros apenas recebem invocações locais
- ◆ Objeto Remoto
 - ◆ aquele que pode receber invocações remotas
- ◆ Referência Remota do Objeto
 - ◆ necessária para a invocação remota sobre o OR
 - ◆ identificação única no SD para aquele objeto
 - ◆ pode ser passada como argumento (Java)
- ◆ Interface Remota
 - ◆ especificação dos métodos que podem ser invocados remotamente

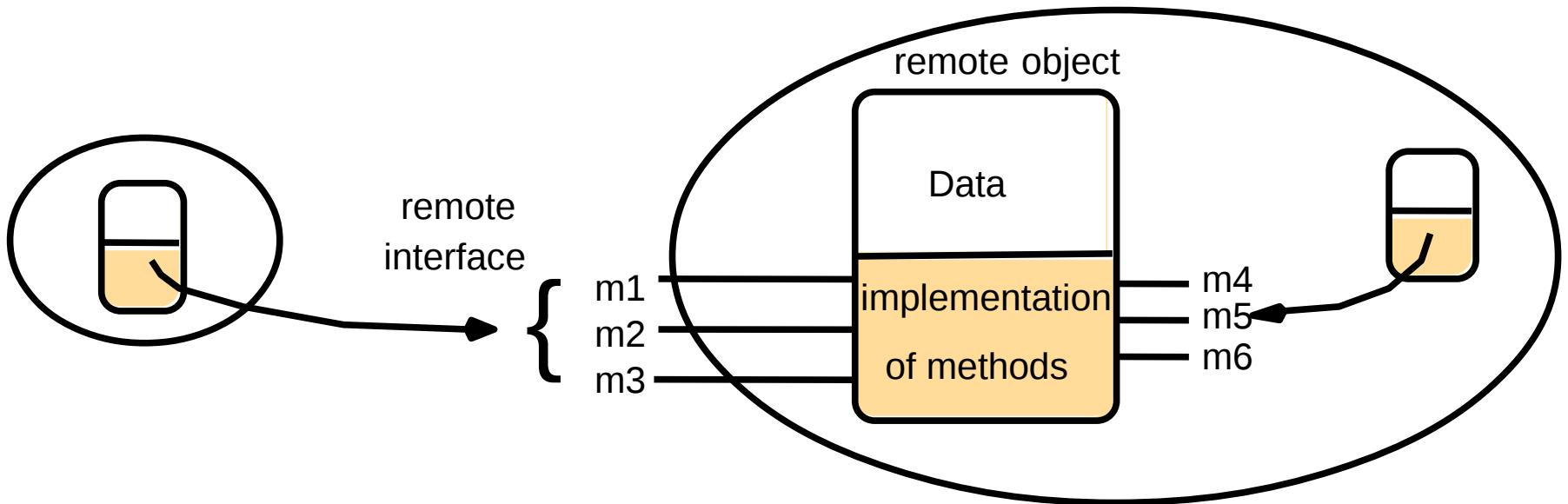
Invocação: local vs remota

- ◆ Invocação é Remota quando incide sobre um objeto de **outro** processo (que pode ou não estar na mesma máquina)



Object Model Distribuído

- Interface Remota



objetos do mesmo processo: podem invocar m1, ..., m4..., m6

objetos noutro processo: podem apenas invocar os métodos da interface remota m1,..., m3

Object Model Distribuído

- ◆ Ações
 - ◆ iniciadas pela invocação* de métodos
 - * - que agora pode ser remota e desencadear outras invocações remotas em cadeia
- ◆ *Garbage Collection*
 - ◆ a nível distribuído, é necessário considerar as referências remotas para o objeto
- ◆ Exceções
 - ◆ podem surgir da mesma forma que na invocação local
 - ◆ Java: `RemoteException`

Detalhes na conceção da abstração RMI

- ◆ Semântica na invocação em RMI
- ◆ garantias de *doOperation* no protocolo RR:
 - ◆ reenvio do pedido
 - ◆ reenviar a mensagem com o pedido para o servidor até a resposta chegar ou se detectar que o servidor está com problemas
 - ◆ filtragem de duplicados
 - ◆ decidir se o duplicado deve ser processado para reenvio ou ignorado
 - ◆ retransmissão de resultados
 - ◆ através de um histórico de resultados para evitar uma nova execução da operação

Semântica da Invocação

<i>medidas</i>			<i>semântica de invocação</i>
<i>reenvio do pedido</i>	<i>filtragem de duplicados</i>	<i>executar de novo ou retransmitir resultado do histórico</i>	
Não	Não aplicável	Não aplicável	<i>Maybe</i>
Sim	Não	executar de novo	<i>At-least-once</i>
Sim	Sim	retransmitir do histórico	<i>At-most-once</i>

- ◆ Possíveis Falhas:
 - ◆ Maybe: omissão, crash
 - ◆ At-least-once: crash, arbitrárias (retransmissão do pedido, reexecução)
 - ◆ At-most-once: falhas evitam-se com os mecanismos de tolerância a falhas

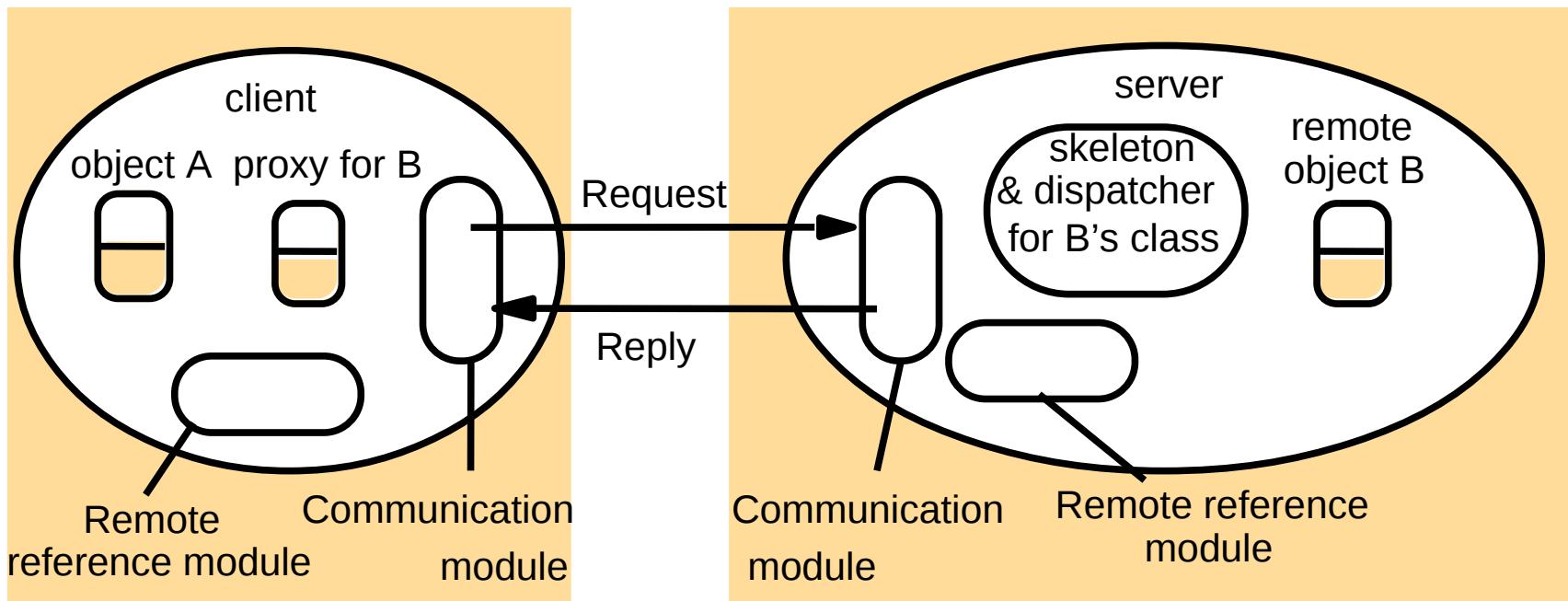
Implementação de RMI: **componentes**

- ◆ módulo de comunicação
 - ◆ protocolo RR, semântica da invocação, do lado do servidor escolhe o *dispatcher*
- ◆ módulo de referências remotas
 - ◆ tradução entre referências remotas e locais, criação de refs. remotas
- ◆ Software RMI
 - ◆ Proxy
 - ◆ torna a invocação remota transparente para o cliente
 - ◆ *marshalling* de argumentos, *unmarshalling* de resultado da invocação
 - ◆ existe um proxy para cada objeto remoto que um processo referencia
 - ◆ implementa os métodos da interface remota do objeto, mas cada método faz *marshall* da referência do objeto, *methodId*, e argumentos, aguardando a resposta para o *unmarshall*

Implementação de RMI: componentes

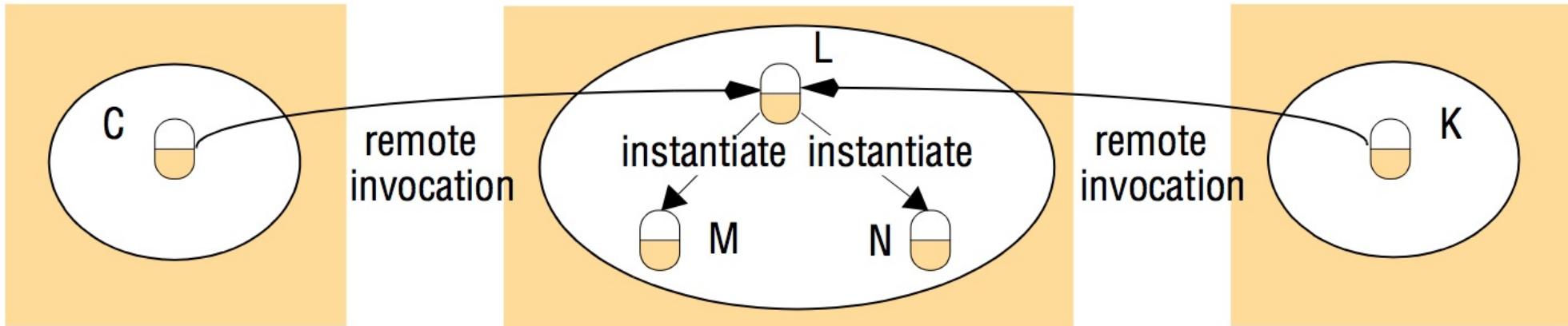
- ◆ Software RMI - *continuação*
 - ◆ Dispatcher
 - ◆ um para cada classe de objeto remoto, no servidor
 - ◆ recebe a mensagem e pelo *methodId* seleciona o método apropriado no Skeleton
 - ◆ Skeleton
 - ◆ um por cada classe que representa um objeto remoto, no servidor
 - ◆ implementa os métodos na interface remota, mas efetuando *unmarshall* a argumentos no pedido, invocando o método no objeto remoto (localmente) e devolvendo o *marshall* do resultado e eventual exceção na resposta ao proxy
- ◆ Orbix CORBA e Java RMI: proxy, dispatcher e skeleton são gerados automaticamente

RMI: Proxy e Skeleton



Objetos Remotos: podem ser criados *on-demand*

- novas instâncias ou ativações



Objeto Remoto pode estar em **estado**:

- **Ativo:** disponível para invocação a qualquer momento
 - permanece junto ao processo a que pertence
- **Passivo:** não disponível, mas pode ativar-se
 - inclui os seus métodos e uma representação serializada/*marshalled* do seu estado

Servidor, Cliente, Binder: quem tem e faz o quê?

- ◆ Servidor
 - ◆ classes para dispatcher e skeleton
 - ◆ classes com implementação para todos os objetos remotos *Servant ou Impl*
 - ◆ zona de inicialização:
 - ◆ criar pelo menos um objeto remoto e inicializá-lo
 - ◆ registrar o objeto no *binder*
 - ◆ para evitar demoras, cada invocação remota é tratada numa nova Thread
- ◆ Cliente
 - ◆ classes dos proxies dos objetos remotos usados
 - ◆ lookup no *binder* para obter a referência remota do objeto
- ◆ Binder
 - ◆ serviço que guarda pares (nome, referência remota do objeto)
 - ◆ “servidor de nomes”

Garbage Collection Distribuído

- ◆ Se um objeto tem uma referência local ou remota no conjunto de objetos distribuídos, então deve continuar a existir.
- ◆ Cooperação com o GC local:
 - ◆ cada servidor mantém uma lista com o conjunto de processos com referências para os seus objetos
 - ◆ quando um cliente cria um Proxy para um objeto, é adicionado ao conjunto de processos com referências para aquele objeto
 - ◆ quando o GC do cliente deteta que o Proxy do objeto já não é necessário/referido, envia uma mensagem ao servidor (`removeRef(O)`) e elimina o proxy. O servidor remove o processo da lista.
 - ◆ Quando a lista estiver vazia, o GC do servidor recupera o espaço do objeto, excepto se existirem referências locais.

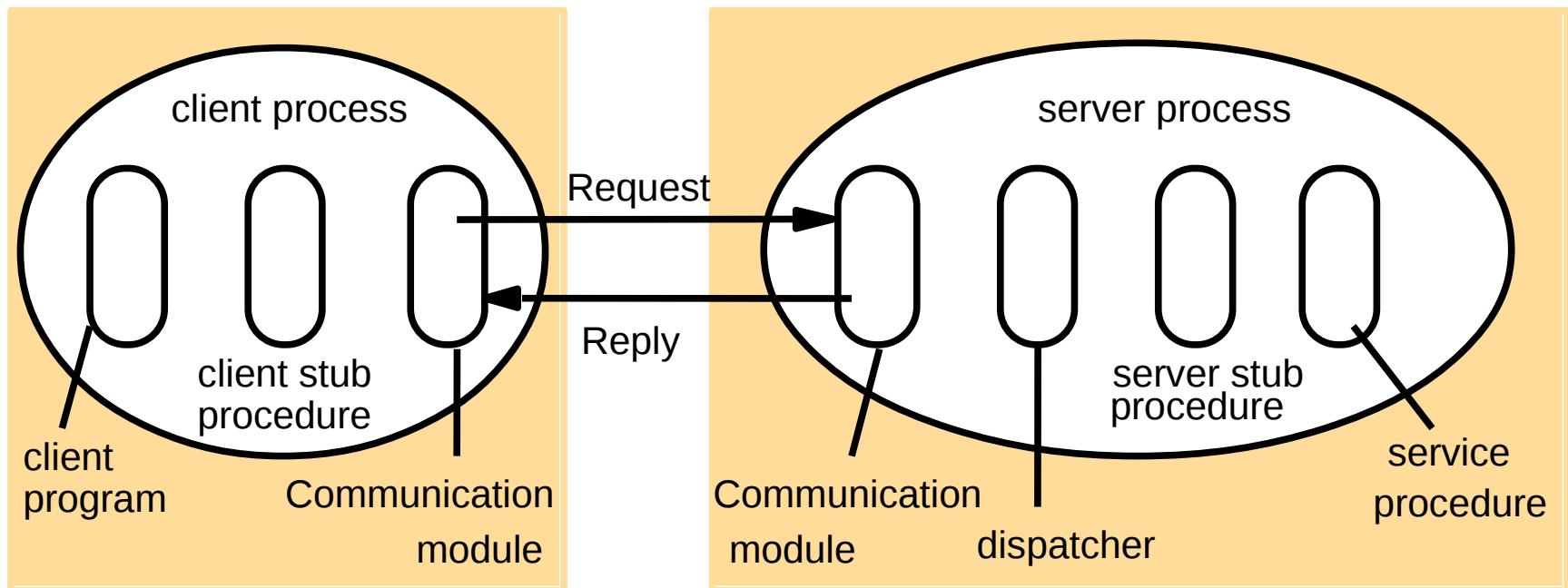
Garbage Collection Distribuído

- ◆ protocolo RR
- ◆ semântica de invocação *At-most-once*
- ◆ **Java Distributed GC**
 - ◆ tolera falhas no cliente
 - ◆ o servidor (com objetos remotos) atribui um intervalo de tempo (*lease*) ao cliente
 - ◆ a contagem é válida até o tempo expirar ou o cliente pedir *removeRef()*
 - ◆ o cliente é responsável por renovar periodicamente o seu *lease*
 - ◆ e assim ser contabilizado na lista de processos com referências para os objetos remotos

Abstração RPC

- ◆ *service interface*
- ◆ semânticas de invocação usualmente implementadas:
 - ◆ *at-least-once*
 - ◆ *at-most-once*
- ◆ sobre o protocolo RR
- ◆ Software RPC
 - ◆ semelhante ao anterior, mas sem Módulo de Referências Remotas, uma vez que não há a noção de objetos

RPC: stub de cliente e de servidor, dispatcher



RPC

- ◆ Stub de Cliente
 - ◆ como o Proxy: tem métodos locais que fazem marshalling, (unmarshalling) no pedido (resposta) para o servidor
- ◆ Server Stub
 - ◆ como o Skeleton (unmarshalling, marshalling)
- ◆ Dispatcher
 - ◆ recebe o pedido do cliente e escolhe o stub adequado
- ◆ Existe um *Stub Procedure* por procedimento na *Service Interface*
- ◆ Stubs e Dispatcher
 - ◆ gerados automaticamente a partir da interface do serviço

RPC

- ◆ Sun RPC (aka ONC RPC)
 - ◆ sobre TCP ou UDP
 - ◆ *interface language*: XDR (external data representation)
- ◆ SunRPC / XDR
 - ◆ não permite dar o nome à interface
 - ◆ números de programa e versão
 - ◆ o procedimento a invocar é identificado por um nº, no pedido
 - ◆ é permitido um único parâmetro de input (*vários: uma estrutura*)
 - ◆ o output do procedimento sai num único resultado/valor

Interface para ler e escrever ficheiros – Sun XDR

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

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

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

Sun RPC

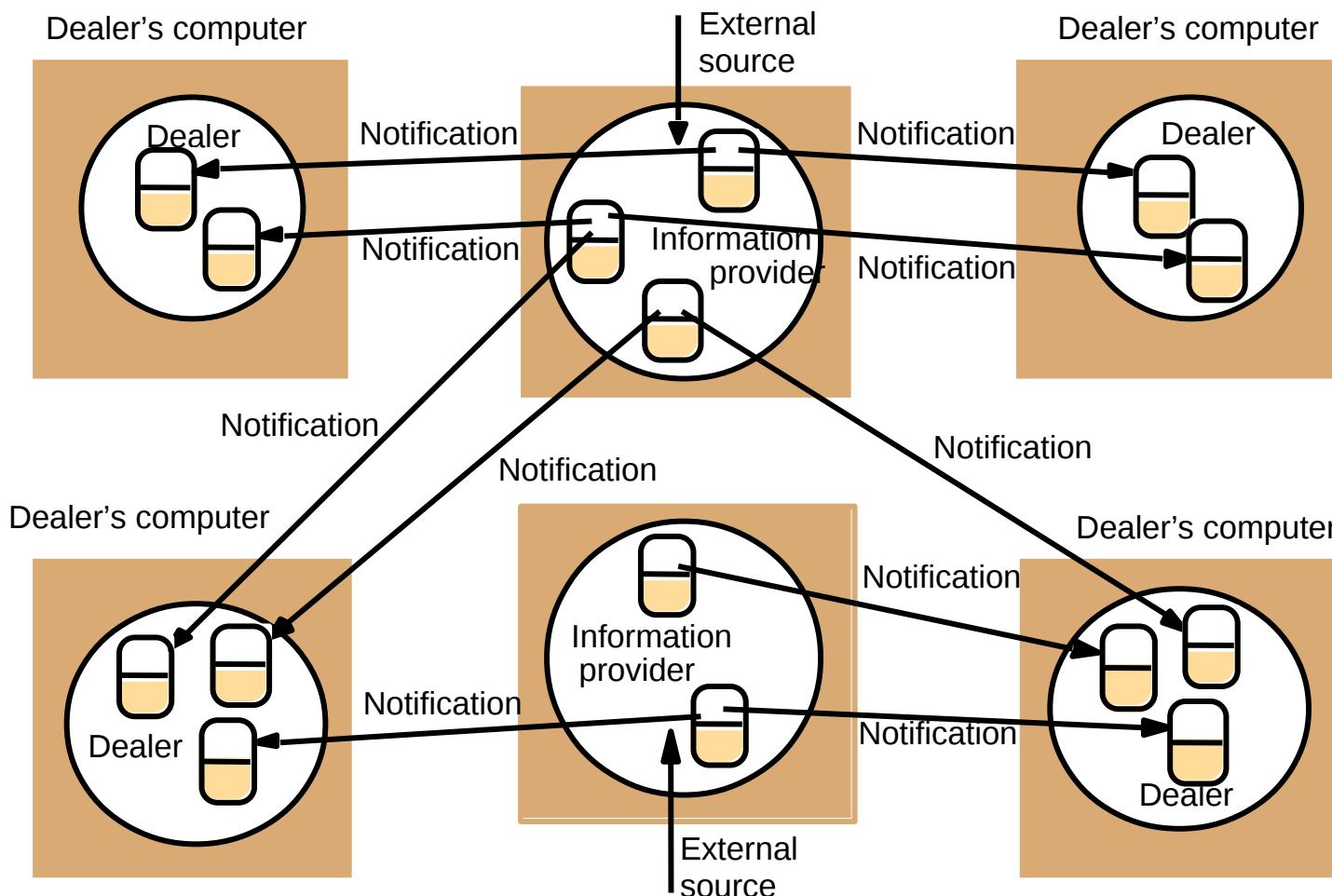
- ◆ XDR
 - ◆ permite definir constantes, `typedefs*`, estruturas*, enumerações*, unions e programas.
(* com a sintaxe da linguagem C)
- ◆ Binding: *port mapper*
 - ◆ ao iniciar, o servidor regista os nº de programa, versão e porto no port mapper
- ◆ Autenticação
 - ◆ mensagens Sun RPC incluem campos adicionais dedicados à autenticação de cliente e servidor

Eventos e Notificações

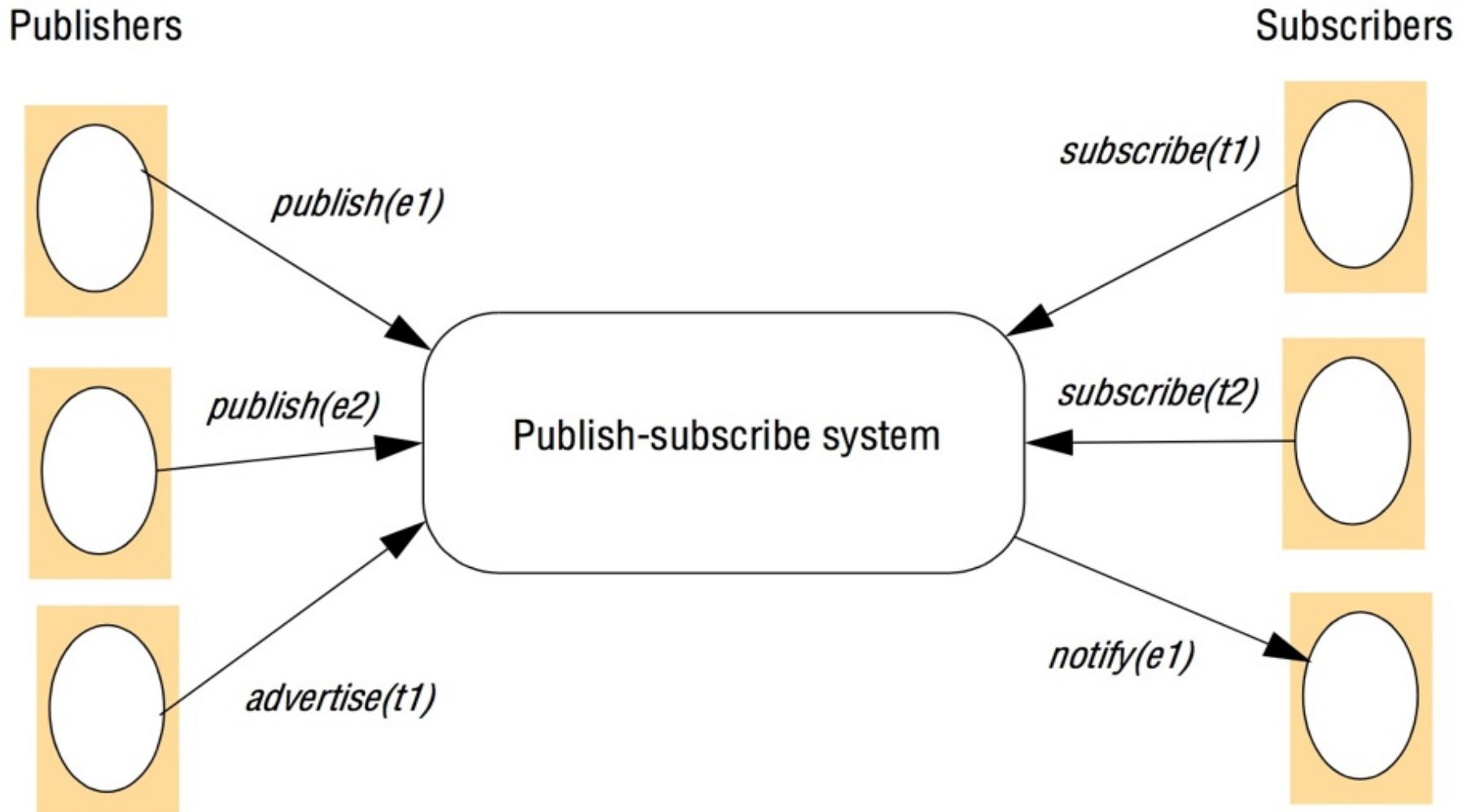
- ◆ Evento: provoca alteração do estado de um objeto
- ◆ Notificações: objetos que representam eventos

- ◆ Características de SD baseados em Eventos
 - ◆ usam paradigma *publish-subscribe*
 - ◆ os eventos são publicados: ficam disponíveis para observação por outros objetos
 - ◆ os objetos que querem receber notificações subscrevem o tipo de eventos que lhes interessa
 - ◆ heterogêneos
 - ◆ os componentes têm conhecimento dos outros componentes (a quem subscrevem ou para quem notificam)
 - ◆ assíncronos
 - ◆ notificações enviadas de modo assíncrono

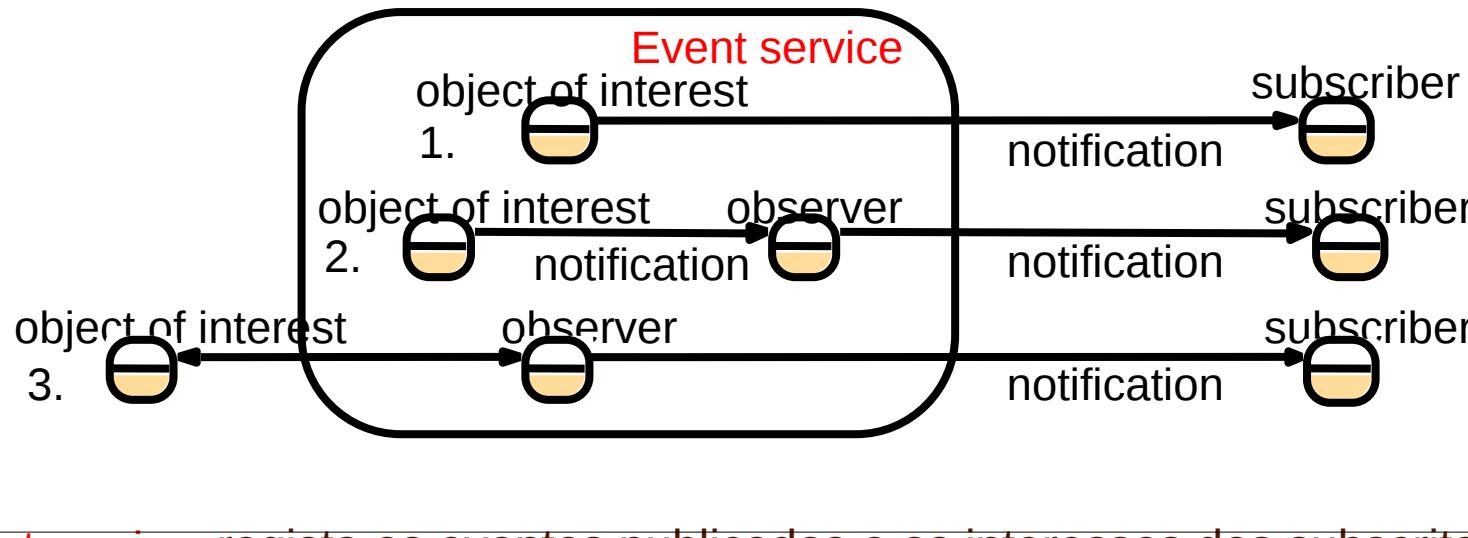
Exemplo: Notificações num Sistema de uma Corretora



Paradigma Publish-Subscribe



Arquitetura para um SD com notificação de eventos



Subscriber: objeto interessado em certo tipo de eventos de outro objeto

Event service: regista os eventos publicados e os interesses dos subscriptores

Object of Interest: objeto que sofre alteração de estado em resultado de operações/invocações

Observer: intermediário entre o *Object of Interest* e o *Subscriber*, tem as funções:

- **Forwarding**: pode haver muitos *subscribers*, libertando o Ool
- **Filtragem de Notificações**
- **Padrões de eventos**: notificação especial na ocorrência de um padrão de eventos
- **Notification Mailbox**: guarda as notificações até o subscriber estar pronto para as receber

Notificação de Eventos em Java

◆ Jini

- ◆ permite a um *subscriber* de uma JVM subscrever e receber notificações de eventos num *Object of Interest* localizado numa JVM diferente (eventualmente noutra máquina)
- ◆ utiliza Java RMI no envio de notificações

Notificação de Eventos em Java: Jini

- ◆ *EventGenerator*: interface com o método *register()*. É implementada pelos objetos cujos eventos geram notificações.
- ◆ *RemoteEvent*: classe que descreve o tipo de evento, o gerador do evento, nº de sequência para aquele tipo de evento e ainda uma representação serializada do evento
- ◆ *RemoteEventListener*: interface com o método *notify()*. Implementada pelos *subscribers* e *third-party agents* para recebem as notificações (pela invocação remota deste método)
- ◆ *Third-party agents*: podem existir entre o Objecto de Interesse e o *subscriber*. Equivalentes aos *observers*.

Java RMI

- ◆ permite a invocação remota com uma sintaxe idêntica à invocação local
- ◆ o cliente tem de *tratar* as `RemoteExceptions`
- ◆ Interface Remota: definida em Java
- ◆ cuidado: prever o comportamento do objeto remoto com a concorrência
- ◆ Exemplo: servidor guarda registo de um quadro com figuras
 - ◆ objetos remotos: `Shape`, `ShapeList` – *ver livro*

Java RMI: interfaces remotas *Shape* e *ShapeList*

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException; 1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException; 2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Java RMI

• Interface Remota

- ◆ interface remota: estender a interface `java.rmi.Remote`
- ◆ declaração de métodos: *throws RemoteException (e eventualmente outras específicas da aplicação)*

• Parâmetros e Resultado

- argumentos do método da interface: parâmetros de input
- retorno do método: único parâmetro de output
- as classes dos argumentos e retorno do método devem ser serializáveis
 - (todos os tipos primitivos e objetos remotos são serializáveis)
- passagem de parâmetros (*ver linhas 1 e 2*)
 - objetos remotos: por referência (remota)
 - objetos não remotos: passagem por valor

Java RMI

- ◆ Download de Classes
 - ◆ em Java as classes podem passar de uma JVM para outra
 - ◆ se o cliente não tem a classe do tipo de retorno no método, o código é obtido do seu interlocutor automaticamente
 - ◆ Vantagem: transparência
 - ◆ Segurança: precisamos de regras para aceitar código – RMISecurityManager
- ◆ RMI registry
 - ◆ binder, serviço de nomes
 - ◆ mapeia nomes* e referências remotas
 - ◆ * rmi://hostname:port/objectName
 - ◆ ver `java.rmi.Naming`

Java RMI Registry: classe Naming

void rebind (String name, Remote obj)

método usado por um servidor para adicionar uma entrada, fazendo o registo de um par (nome,referência para o objeto remoto).

void bind (String name, Remote obj)

método usado por um servidor para adicionar uma entrada, fazendo o registo de um par (nome,referência para o objeto remoto).

Se o nome já existir é lançada uma exceção.

void unbind (String name, Remote obj)

método para eliminar uma entrada

Remote lookup(String name)

método invocado por clientes para obter uma referência para um objeto remoto a partir do seu nome

String [] list()

devolve um array de Strings com os nomes registados neste serviço

Java RMI: um servidor

```
import java.rmi.*;  
public class ShapeListServer{  
    public static void main(String args[]){  
        System.setSecurityManager(new RMISecurityManager());  
        try{  
            ShapeList aShapeList = new ShapeListServant();  
            Naming.rebind("Shape List", aShapeList );  
            System.out.println("ShapeList server ready");  
        }catch(Exception e) {  
            System.out.println("ShapeList server main " + e.getMessage());}  
    }  
}
```

1
2

Java RMI: Servant (*ServiceImpl – classe do objecto remoto*)

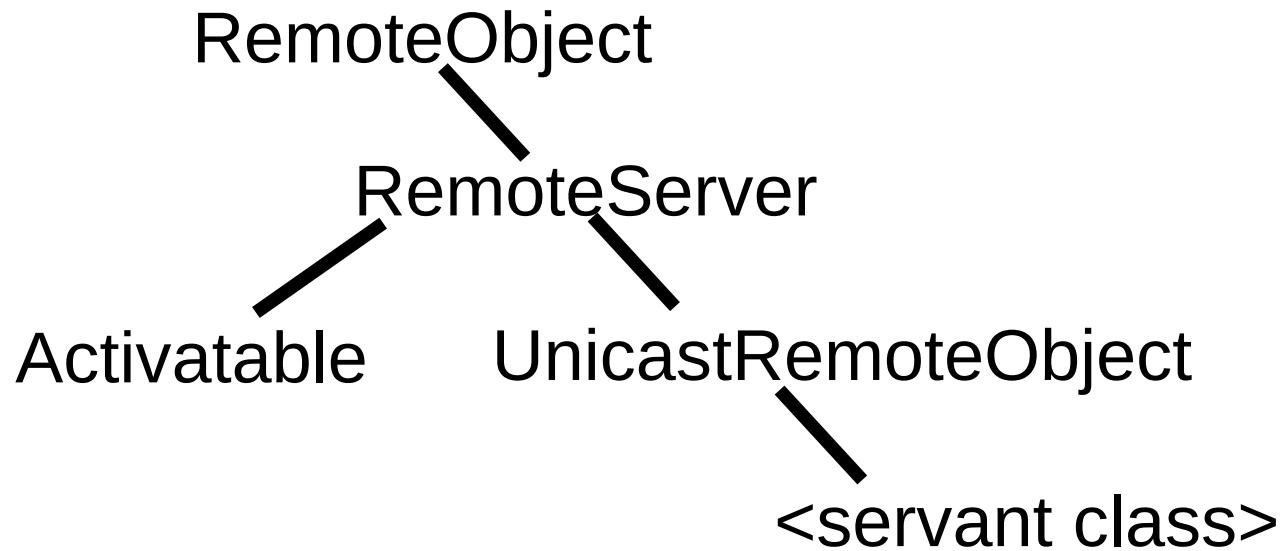
```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
    private Vector theList;                                // contains the list of Shapes 1
    private int version;
    public ShapeListServant() throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException { 2
        version++;
        Shape s = new ShapeServant( g, version);
        theList.addElement(s);
        return s;
    }
    public Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
```

implementação dos 3
métodos sem os detalhes de
comunicação

Java RMI: um cliente

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList");
            Vector sList = aShapeList.allShapes();
        } catch(RemoteException e) {System.out.println(e.getMessage());}
        }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
    1
    2
```

Java RMI: hierarquia de classes (server-side)



Callbacks

- ◆ há situações em que é o servidor que toma a iniciativa de contactar o cliente devido a um evento
 - ◆ evita consultas desnecessárias por parte do cliente
 - ◆ implementável via RMI
 - ◆ cliente cria um objeto* remoto com o método que o servidor há-de invocar
 - * designado *callback object*
 - ◆ cada cliente interessado informa o servidor do seu *callback object*
 - ◆ a cada evento, o servidor invoca o método junto daqueles clientes

Java IDL

- ◆ conectividade e interoperabilidade com ambiente CORBA
- ◆ permite desenvolver uma aplicação em Java, com uma interface definida em CORBA IDL, e inseri-la num ambiente CORBA
 - ◆ (com outras aplicações possivelmente noutras linguagens)

CORBA (*common object request broker arquitecture*)

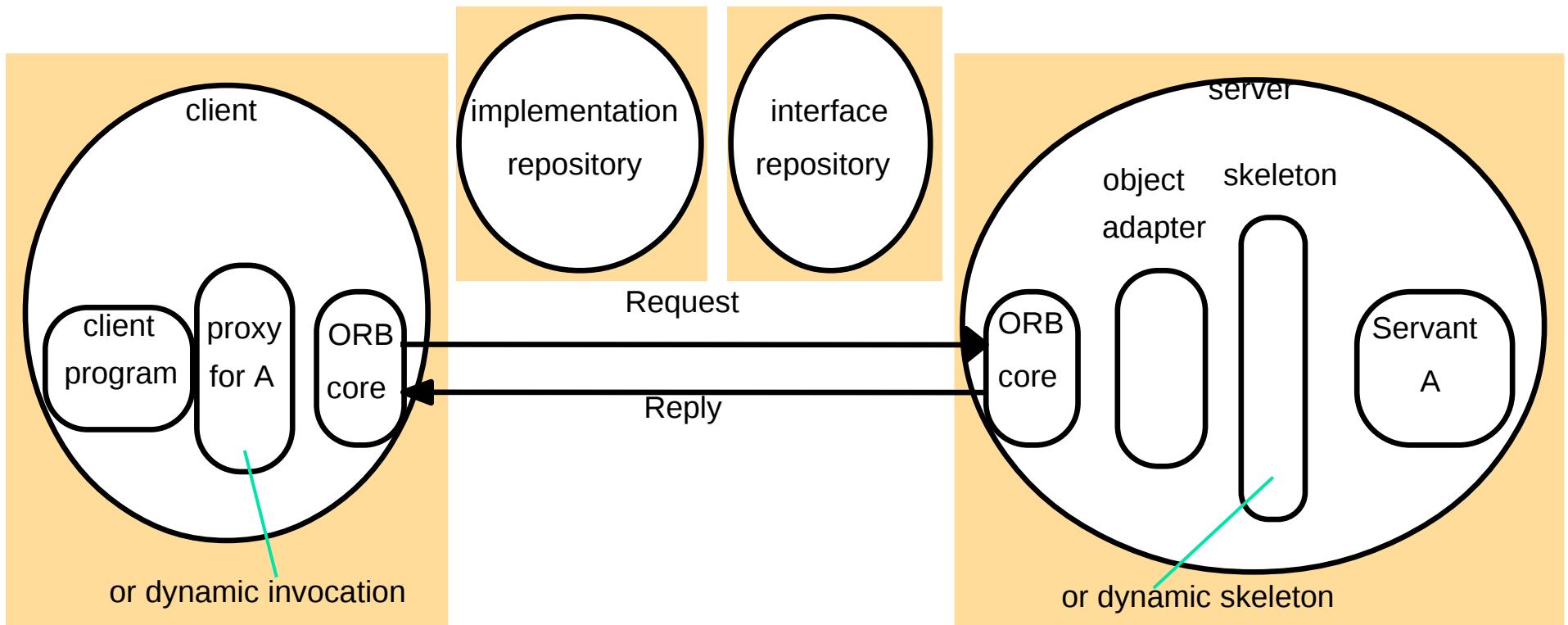
- ◆ permitir que objetos distribuídos e implementados em diferentes linguagens de programação possam comunicar
- ◆ assenta na metáfora do *object request broker*
 - ◆ ajuda um cliente a invocar um método num objeto
 - ◆ localizar o objeto
 - ◆ ativar o objeto, se necessário
 - ◆ comunicar o pedido ao objeto, que por sua vez presta o serviço
 - ◆ devolver a resposta ao cliente

CORBA

- ◆ Componentes principais:
 - ◆ IDL
 - ◆ arquitetura
 - ◆ formato para representação externa de dados CDR e formato das mensagens a trocar no protocolo RR, definidos na norma GIOP (*General InterORB Protocol*)
 - ◆ referência de objetos remotos, definida pela norma IIOP (*Internet Inter-Orb Protocol*)
- ◆ Passagem de parâmetros:
 - ◆ tipo definido pela IDL: é devolvida uma referência remota
 - ◆ tipos primitivos e outros tipos compostos: copiados e passados por valor
- ◆ Semântica de invocação
 - ◆ *at-most-once*

Arquitetura CORBA

- ◆ Semelhante à arquitetura Java RMI... com alguns componentes novos



Arquitetura CORBA

- ◆ **Object Adapter:** faz a ponte entre os objetos CORBA na interface IDL e os objetos do servant com a interface da linguagem de programação
- ◆ **ORB Core:** semelhante ao módulo de comunicação de Java RMI
- ◆ **Implementation repository:** mapeia nomes de *object adapters* em pathnames de ficheiros com a implementação dos objetos, bem como o hostname e porto do servidor em que se encontram
- ◆ **Interface repository:** disponibiliza aos clientes a informação sobre interfaces IDL registadas
- ◆ **skeleton:** gerado pelo compilador de IDL, na linguagem do servidor. Faz *unmarshall* e *marshall* de argumentos e resultado ou exceção, respetivamente
- ◆ **client stub/proxy:** gerado na linguagem do cliente, pelo compilador de IDL. Faz *marshall* e *unmarshall* de argumentos e resultado, respetivamente

Cloud Infrastructure

Existing cloud infrastructure

- The cloud computing infrastructure at Amazon, Google, and Microsoft (as of mid 2012).
 - Amazon is a pioneer in Infrastructure-as-a-Service (IaaS).
 - Google's efforts are focused on Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS).
 - Microsoft is involved in PaaS.
- Private clouds are an alternative to public clouds. Open-source cloud computing platforms such as:
 - Eucalyptus,
 - OpenNebula,
 - Nimbus,
 - OpenStack

can be used as a control infrastructure for a private cloud.

Amazon Web Services (AWS)

- AWS → IaaS cloud computing services launched in 2006.
- Businesses in 200 countries used AWS in 2012.
- The infrastructure consists of compute and storage servers interconnected by high-speed networks and supports a set of services.
- An application developer:
 - Installs applications on a platform of his/her choice.
 - Manages resources allocated by Amazon.

AWS regions and availability zones

- Amazon offers cloud services through a network of data centers on several continents.
- In each *region* there are several availability zones interconnected by high-speed networks.
- An *availability zone* is a data center consisting of a large number of servers.

Region	Location	Availability zones	Cost
US West	Oregon	us-west-2a/2b/2c	Low
US West	North California	us-west-1a/1b/1c	High
US East	North Virginia	us-east-1a/2a/3a/4a	Low
Europe	Ireland	eu-west-1a/1b/1c	Medium
South America	Sao Paulo, Brazil	sa-east-1a/1b	Very high
Asia Pacific	Tokyo, Japan	ap-northeast-1a/1b	High
Asia Pacific	Singapore	ap-southeast-1a/1b	Medium

- Regions do not share resources and communicate through the Internet.

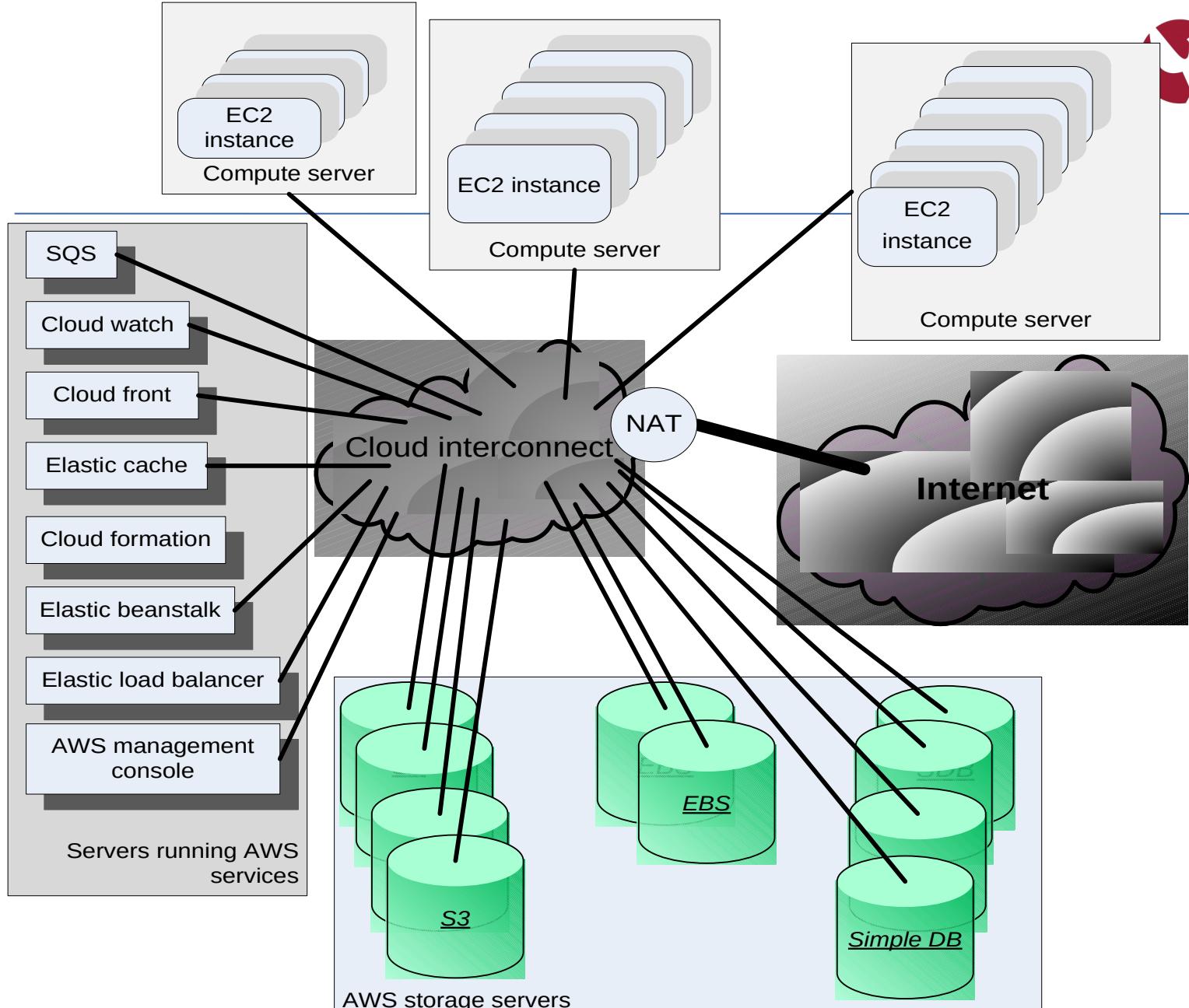


AWS instances

- An instance is a virtual server with a well specified set of resources including: CPU cycles, main memory, secondary storage, communication and I/O bandwidth.
- The user chooses:
 - The region and the availability zone where this virtual server should be placed.
 - An instance type from a limited menu of instance types.
- When launched, an instance is provided with a DNS name; this name maps to a
 - *private IP address* → for internal communication within the internal EC2 communication network.
 - *public IP address* → for communication outside the internal Amazon network, e.g., for communication with the user that launched the instance.

AWS instances (cont'd)

- Network Address Translation (NAT) maps external IP addresses to internal ones.
- The public IP address is assigned for the lifetime of an instance.
- An instance can request an *elastic IP address*, rather than a public IP address. The elastic IP address is a static public IP address allocated to an instance from the available pool of the availability zone.
- An elastic IP address is not released when the instance is stopped or terminated and must be released when no longer needed.



Steps to run an application

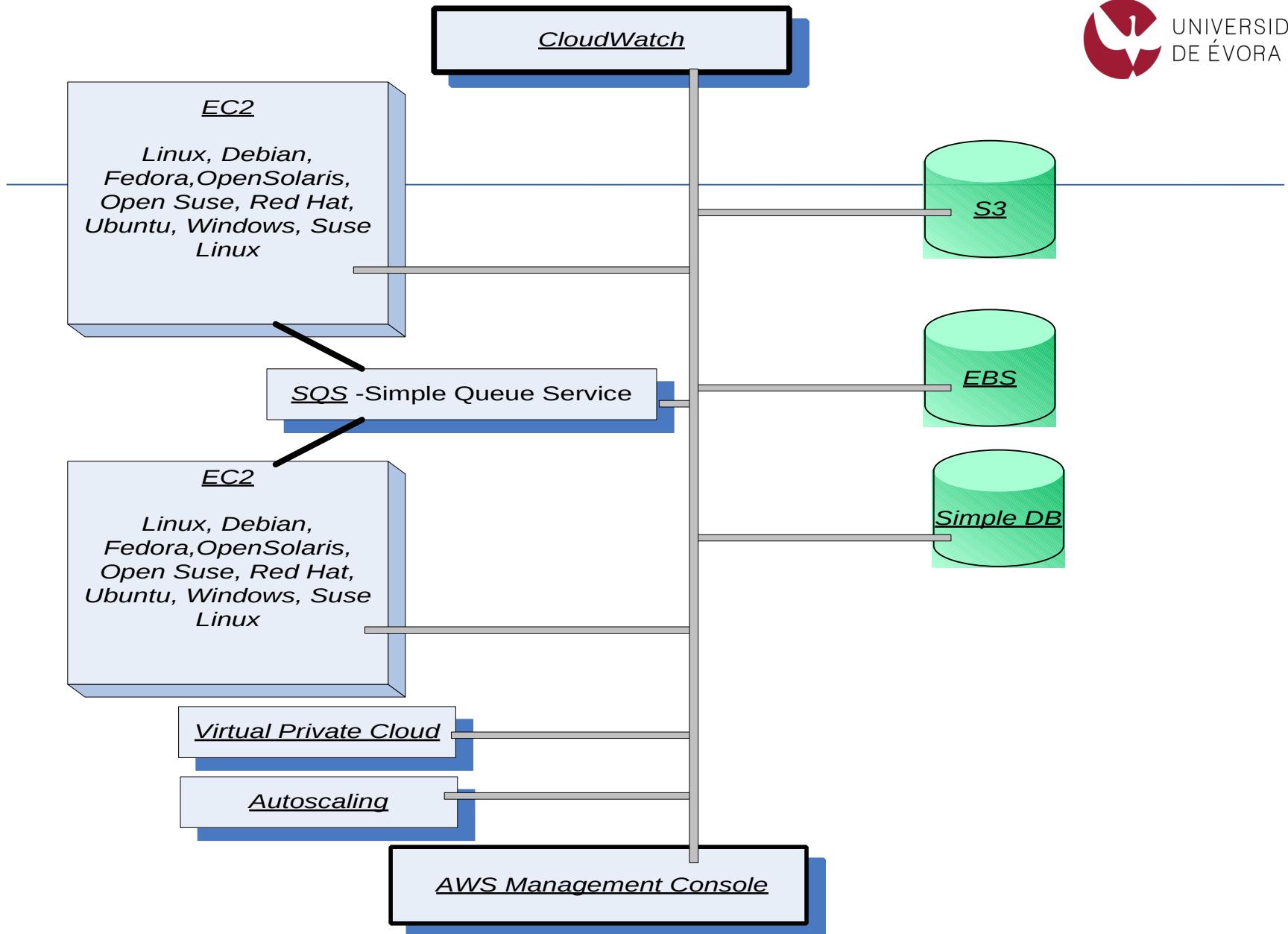
- Retrieve the user input from the front-end.
- Retrieve the disk image of a VM (Virtual Machine) from a repository.
- Locate a system and requests the VMM (Virtual Machine Monitor) running on that system to setup a VM.
- Invoke the Dynamic Host Configuration Protocol (DHCP) and the IP bridging software to set up MAC and IP addresses for the VM.

User interactions with AWS

- The AWS Management Console. The easiest way to access all services, but not all options may be available.
- AWS SDK libraries and toolkits are provided for several programming languages including Java, PHP, C#, and Objective-C.
- Raw REST requests.

Examples of Amazon Web Services

- *AWS Management Console* - allows users to access the services offered by AWS .
- *Elastic Cloud Computing (EC2)* - allows a user to launch a variety of operating systems.
- *Simple Queuing Service (SQS)* - allows multiple *EC2* instances to communicate with one another.
- *Simple Storage Service (S3)*, *Simple DB*, and *Elastic Block Storage (EBS)* - storage services.
- *Cloud Watch* - supports performance monitoring.
- *Auto Scaling* - supports elastic resource management.
- *Virtual Private Cloud* - allows direct migration of parallel applications.



EC2 – Elastic Cloud Computing

- EC2 - web service for launching instances of an application under several operating systems, such as:
 - Several Linux distributions.
 - Microsoft Windows Server 2003 and 2008.
 - OpenSolaris.
 - FreeBSD.
 - NetBSD.
- A user can
 - Load an EC2 instance with a custom application environment.
 - Manage network's access permissions.
 - Run the image using as many or as few systems as desired.

EC2 (cont'd)

- Import virtual machine (VM) images from the user environment to an instance through *VM import*.
- EC2 instances boot from an AMI (Amazon Machine Image) digitally signed and stored in S3.
- Users can access:
 - Images provided by Amazon.
 - Customize an image and store it in S3.
- An EC2 instance is characterized by the resources it provides:
 - VC (Virtual Computers) – virtual systems running the instance.
 - CU (Compute Units) – measure computing power of each system.
 - Memory.
 - I/O capabilities.

Instance types

- Standard instances: micro (StdM), small (StdS), large (StdL), extra large (StdXL); small is the default.
- High memory instances: high-memory extra large (HmXL), high-memory double extra large (Hm2XL), and high-memory quadruple extra large (Hm4XL).
- High CPU instances: high-CPU extra large (HcpuXL).
- Cluster computing: cluster computing quadruple extra large (Cl4XL).

Instance name	API name	Platform (32/64-bit)	Memory (GB)	Max EC2 compute units	I-memory (GB)	I/O (M/H)
StdM		32 and 64	0. 633	1 VC; 2 CUs		
StdS	<code>m1.small</code>	32	1.7	1 VC; 1 CU	160	M
StdL	<code>m1.large</code>	64	7.5	2 VCs; 2 × 2 CUs	85	H
StdXL	<code>m1.xlarge</code>	64	15	4 VCs; 4 × 2 CUs	1,690	H
HmXL	<code>m2.xlarge</code>	64	17.1	2 VCs; 2 × 3.25 CUs	420	M
Hm2XL	<code>m2.2xlarge</code>	64	34.2	4 VCs; 4 × 3.25 CUs	850	H
Hm4XL	<code>m2.4xlarge</code>	64	68.4	8 VCs; 8 × 3.25 CUs	1,690	H
HcpuXL	<code>c1.xlarge</code>	64	7	8 VCs; 8 × 2.5 CUs	1,690	H
Cl4XL	<code>cc1.4xlarge</code>	64	18	33.5 CUs	1,690	H

S3 – Simple Storage System

- Service designed to store large objects; an application can handle an unlimited number of objects ranging in size from 1 byte to 5 TB.
- An object is stored in a bucket and retrieved via a unique, developer-assigned key; a bucket can be stored in a Region selected by the user.
- Supports a minimal set of functions: write, read, and delete; it does not support primitives to copy, to rename, or to move an object from one bucket to another.
- The object names are global.
- S3 maintains for each object: the name, modification time, an access control list, and up to 4 KB of user-defined metadata.

S3 (cont'd)

- Authentication mechanisms ensure that data is kept secure.
- Objects can be made public, and rights can be granted to other users.
- S3 computes the MD5 of every object written and returns it in a field called ETag.
- A user is expected to compute the MD5 of an object stored or written and compare this with the ETag; if the two values do not match, then the object was corrupted during transmission or storage.

Elastic Block Store (EBS)

- Provides persistent block level storage volumes for use with *EC2* instances; suitable for database applications, file systems, and applications using raw data devices.
- A volume appears to an application as a raw, unformatted and reliable physical disk; the range 1 GB -1 TB.
- An *EC2* instance may mount multiple volumes, but a volume cannot be shared among multiple instances.
- EBS supports the creation of snapshots of the volumes attached to an instance and then uses them to restart the instance.
- The volumes are grouped together in Availability Zones and are automatically replicated in each zone.

SimpleDB

- Non-relational data store. Supports store and query functions traditionally provided only by relational databases.
- Supports high performance Web applications; users can store and query data items via Web services requests.
- Creates multiple geographically distributed copies of each data item.
- It manages automatically:
 - The infrastructure provisioning.
 - Hardware and software maintenance.
 - Replication and indexing of data items.
 - Performance tuning.

SQS - Simple Queue Service

- Hosted message queues are accessed through standard SOAP and Query interfaces.
- Supports automated workflows - EC2 instances can coordinate by sending and receiving SQS messages.
- Applications using SQS can run independently and asynchronously, and do not need to be developed with the same technologies.
- A received message is “locked” during processing; if processing fails, the lock expires and the message is available again.
- Queue sharing can be restricted by IP address and time-of-day.

CloudWatch

- Monitoring infrastructure used by application developers, users, and system administrators to collect and track metrics important for optimizing the performance of applications and for increasing the efficiency of resource utilization.
- Without installing any software a user can monitor either seven or eight pre-selected metrics and then view graphs and statistics for these metrics.
- When launching an Amazon Machine Image (AMI) the user can start the CloudWatch and specify the type of monitoring:
 - Basic Monitoring - free of charge; collects data at five-minute intervals for up to seven metrics.
 - Detailed Monitoring - subject to charge; collects data at one minute interval.

AWS services introduced in 2012

- *Route 53* - low-latency DNS service used to manage user's DNS public records.
- *Elastic MapReduce (EMR)* - supports processing of large amounts of data using a hosted Hadoop running on *EC2*.
- *Simple Workflow Service (SWF)* - supports workflow management; allows scheduling, management of dependencies, and coordination of multiple *EC2* instances.
- *ElastiCache* - enables web applications to retrieve data from a managed in-memory caching system rather than a much slower disk-based database.
- *DynamoDB* - scalable and low-latency fully managed NoSQL database service.

AWS services introduced in 2012 (cont'd)

- *CloudFront* - web service for content delivery.
- *Elastic Load Balancer* - automatically distributes the incoming requests across multiple instances of the application.
- *Elastic Beanstalk* - handles automatically deployment, capacity provisioning, load balancing, auto-scaling, and application monitoring functions.
- *CloudFormation* - allows the creation of a stack describing the infrastructure for an application.

Elastic Beanstalk

- Handles automatically the deployment, capacity provisioning, load balancing, auto-scaling, and monitoring functions.
- Interacts with other services including EC2, S3, SNS, Elastic Load Balance and AutoScaling.
- The management functions provided by the service are:
 - Deploy a new application version (or rollback to a previous version).
 - Access to the results reported by CloudWatch monitoring service.
 - Email notifications when application status changes or application servers are added or removed.
 - Access to server log files without needing to login to the application servers.
- The service is available using: a Java platform, the PHP server-side description language, or the .NET framework.

SaaS services offered by Google

- *Gmail* - hosts Emails on Google servers and provides a web interface to access the Email.
- *Google docs* - a web-based software for building text documents, spreadsheets and presentations.
- *Google Calendar* - a browser-based scheduler; supports multiple user calendars, calendar sharing, event search, display of daily/weekly/monthly views, and so on.
- *Google Groups* - allows users to host discussion forums to create messages online or via Email.
- *Picasa* - a tool to upload, share, and edit images.
- *Google Maps* - web mapping service; offers street maps, a route planner, and an urban business locator for numerous countries around the world

PaaS services offered by Google

- *AppEngine* - a developer platform hosted on the cloud.
 - Initially supported Python, Java was added later.
 - The database for code development can be accessed with GQL (Google Query Language) with a SQL-like syntax.
- *Google Co-op* - allows users to create customized search engines based on a set of facets/categories.
- *Google Drive* - an online service for data storage.
- *Google Base* - allows users to load structured data from different sources to a central repository, a very large, self-describing, semi-structured, heterogeneous database.

PaaS and SaaS services from Microsoft

- *Windows Azure* - an operating system; has 3 components:
 - Compute - provides a computation environment.
 - Storage - for scalable storage.
 - Fabric Controller - deploys, manages, and monitors applications.
- *SQL Azure* - a cloud-based version of the SQL Server.
- *Azure AppFabric*, formerly .NET Services - a collection of services for cloud applications.

Open-source platforms for private clouds

- *Eucalyptus* - can be regarded as an open-source counterpart of Amazon's EC2.
- *Open-Nebula* - a private cloud with users actually logging into the head node to access cloud functions. The system is centralized and its default configuration uses the NFS file system.
- *Nimbus* - a cloud solution for scientific applications based on Globus software; inherits from Globus:
 - The image storage.
 - The credentials for user authentication.
 - The requirement that a running Nimbus process can **ssh** into all compute nodes.

Eucalyptus

- *Virtual Machines* - run under several VMMS including Xen, KVM, and VMware.
- *Node Controller* - runs on server nodes hosting a VM and controls the activities of the node.
- *Cluster Controller* - controls a number of servers.
- *Cloud Controller* - provides the cloud access to end-users, developers, and administrators.
- *Storage Controller* - provides persistent virtual hard drives to applications. It is the correspondent of EBS.
- *Storage Service (Walrus)* - provides persistent storage; similar to S3, it allows users to store objects in buckets.

EUCALYPTUS

RIGHTSCALE
MYCLOUD

ECOSYSTEM TOOLS

SECURITY

DOCUMENTATION

GET EUCALYPTUS

FastStart

Free Trial

Eucalyptus

Euca2ools

Community Cloud

Source

Home > Eucalyptus Cloud > Get Eucalyptus

DOWNLOAD EUCALYPTUS

First time using Eucalyptus? Try [Eucalyptus FastStart](#).

1. Download and Install Eucalyptus

Choose a distribution:

[CentOS 5](#)[CentOS 6](#)[RHEL 5](#)[RHEL 6](#)[Ubuntu 10.04 LTS](#)[Ubuntu 12.04 LTS](#)[Source](#)[Versions prior to Eucalyptus 3.1](#)[Nightlies](#)[Release Notes](#) 

Looking for [Euca2ools](#)?

2. Configure Your Cloud

[Documentation](#)[Engage \(Q&A\)](#)[Consulting](#)[Education](#)[Support](#)

3. Use Your Cloud

To help get you started, we have prepared pre-packaged virtual machines ready to run in your Eucalyptus cloud.

[Download images](#)

Or check out a variety of [use cases](#).

Learn About
Eucalyptus For

MANAGERS

ARCHITECTS

APPLICATION ARCHITECTS

ADMINISTRATORS

DEVELOPERS

USERS

Euca2ools

Eucalyptus supported command-line tools.

[Get Euca2ools](#)**Ecosystem Tools**

Find tools developed for Amazon EC2 and S3 which are compatible with Eucalyptus.

[Get tools](#)

Cloud storage diversity and vendor lock-in

- Risks when a large organization relies on a single cloud service provider:
 - Cloud services may be unavailable for a short or an extended period of time.
 - Permanent data loss in case of a catastrophic system failure.
 - The provider may increase the prices for service.
- Switching to another provider could be very costly due to the large volume of data to be transferred from the old to the new provider.
- A solution is to replicate the data to multiple cloud service providers, similar to data replication in RAID.

Energy use and ecological impact

- The energy consumption of large-scale data centers and their costs for energy and for cooling are significant.
- In 2006, the 6,000 data centers in the U.S consumed 61×10^9 KWh of energy, 1.5% of all electricity consumption, at a cost of \$4.5 billion.
- The energy consumed by the data centers was expected to double from 2006 to 2011 and peak instantaneous demand to increase from 7 GW to 12 GW.
- The greenhouse gas emission due to the data centers is estimated to increase from 116×10^9 tones of CO₂ in 2007 to 257 tones in 2020 due to increased consumer demand.
- The effort to reduce energy use is focused on computing, networking, and storage activities of a data center.

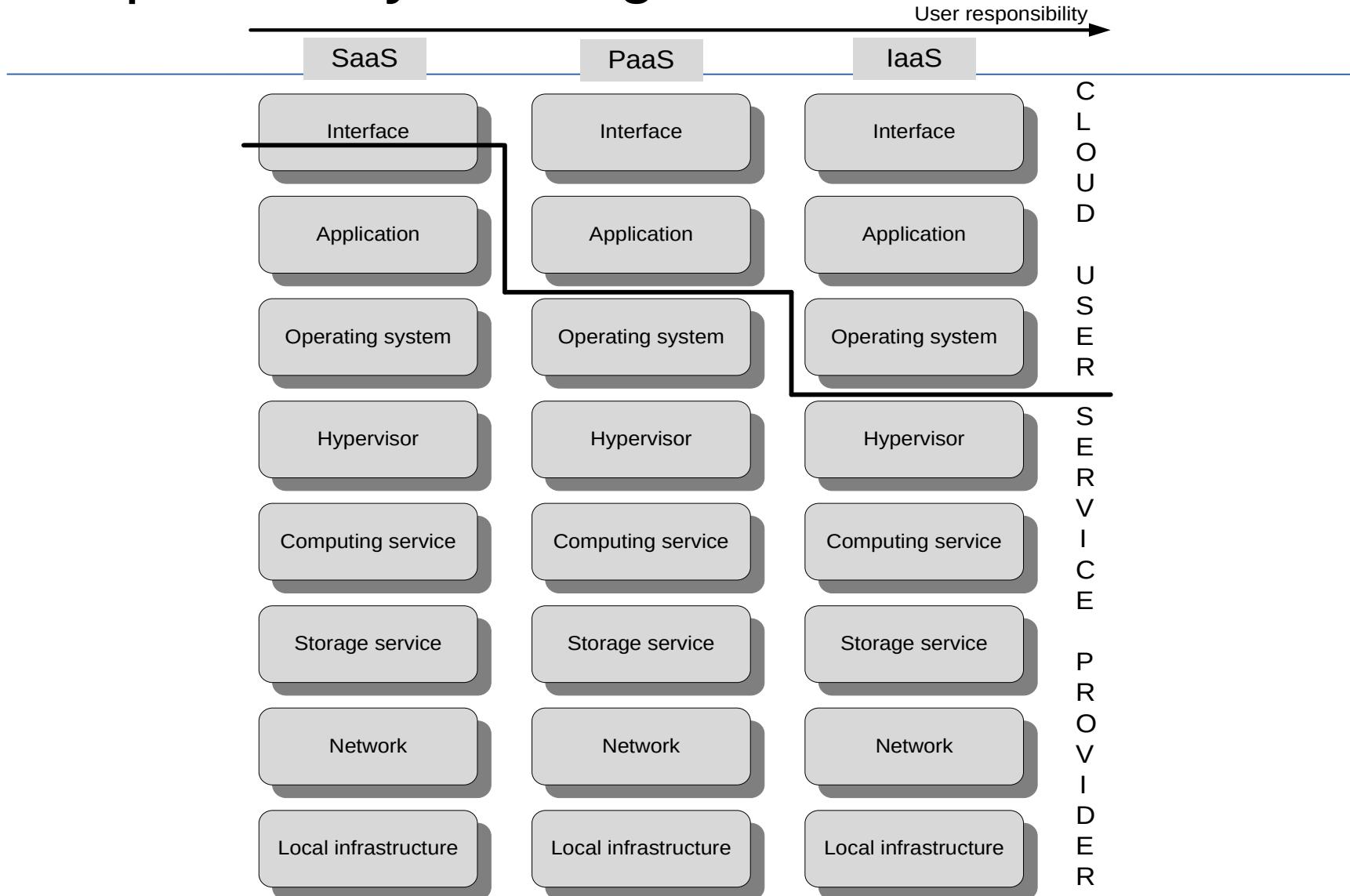
Energy use and ecological impact (cont'd)

- Operating efficiency of a system is captured by the *performance per Watt of power*.
- The performance of supercomputers has increased 3.5 times faster than their operating efficiency – 7,000% versus 2,000% during the period 1998 – 2007.
- A typical Google cluster spends most of its time within the 10-50% CPU utilization range; there is a mismatch between server workload profile and server energy efficiency.

Service Level Agreement (SLA)

- SLA - a negotiated contract between the customer and CSP; can be legally binding or informal. Objectives:
 - Identify and define the customer's needs and constraints including the level of resources, security, timing, and QoS.
 - Provide a framework for understanding; a critical aspect of this framework is a clear definition of classes of service and the costs.
 - Simplify complex issues; clarify the boundaries between the responsibilities of clients and CSP in case of failures.
 - Reduce areas of conflict.
 - Encourage dialog in the event of disputes.
 - Eliminate unrealistic expectations.
- Specifies the services that the customer receives, rather than how the cloud service provider delivers the services.

Responsibility sharing between user and CSP



User security concerns

- Potential loss of control/ownership of data.
- Data integration, privacy enforcement, data encryption.
- Data remanence after de-provisioning.
- Multi tenant data isolation.
- Data location requirements within national borders.
- Hypervisor security.
- Audit data integrity protection.
- Verification of subscriber policies through provider controls.
- Certification/Accreditation requirements for a given cloud service.

Credits, references and reading material

- *Cloud Computing: Theory and Practice*

Dan C. Marinescu

Chapter 3

- *Distributed and Cloud Computing*

K. Hwang, G. Fox and J. Dongarra

Morgan Kaufmann, 2012

Cloud Resource Virtualization

Motivation

There are many physical realizations of the fundamental abstractions necessary to describe the operation of a computing systems.

- Interpreters.
- Memory.
- Communications links.

Virtualization is a basic tenet of cloud computing, it simplifies the management of physical resources for the three abstractions.

The state of a virtual machine (VM) running under a virtual machine monitor (VMM) can be saved and migrated to another server to balance the load.

Virtualization allows users to operate in environments they are familiar with, rather than forcing them to idiosyncratic ones.

Motivation (cont'd)

Cloud resource virtualization is important for:

- System security, as it allows isolation of services running on the same hardware.
- Performance and reliability, as it allows applications to migrate from one platform to another.
- The development and management of services offered by a provider.
- Performance isolation.

Virtualization

Simulates the interface to a physical object by:

- Multiplexing: creates multiple virtual objects from one instance of a physical object. Example - a processor is multiplexed among a number of processes or threads.
- Aggregation: creates one virtual object from multiple physical objects. Example - a number of physical disks are aggregated into a RAID disk.
- Emulation: constructs a virtual object from a different type of a physical object. Example - a physical disk emulates a Random Access Memory (RAM).
- Multiplexing and emulation. Examples - virtual memory with paging multiplexes real memory and disk; a virtual address emulates a real address.

Layering

Layering – a common approach to manage system complexity.

- Minimizes the interactions among the subsystems of a complex system.
- Simplifies the description of the subsystems; each subsystem is abstracted through its interfaces with the other subsystems.
- We are able to design, implement, and modify the individual subsystems independently.

Layering in a computer system.

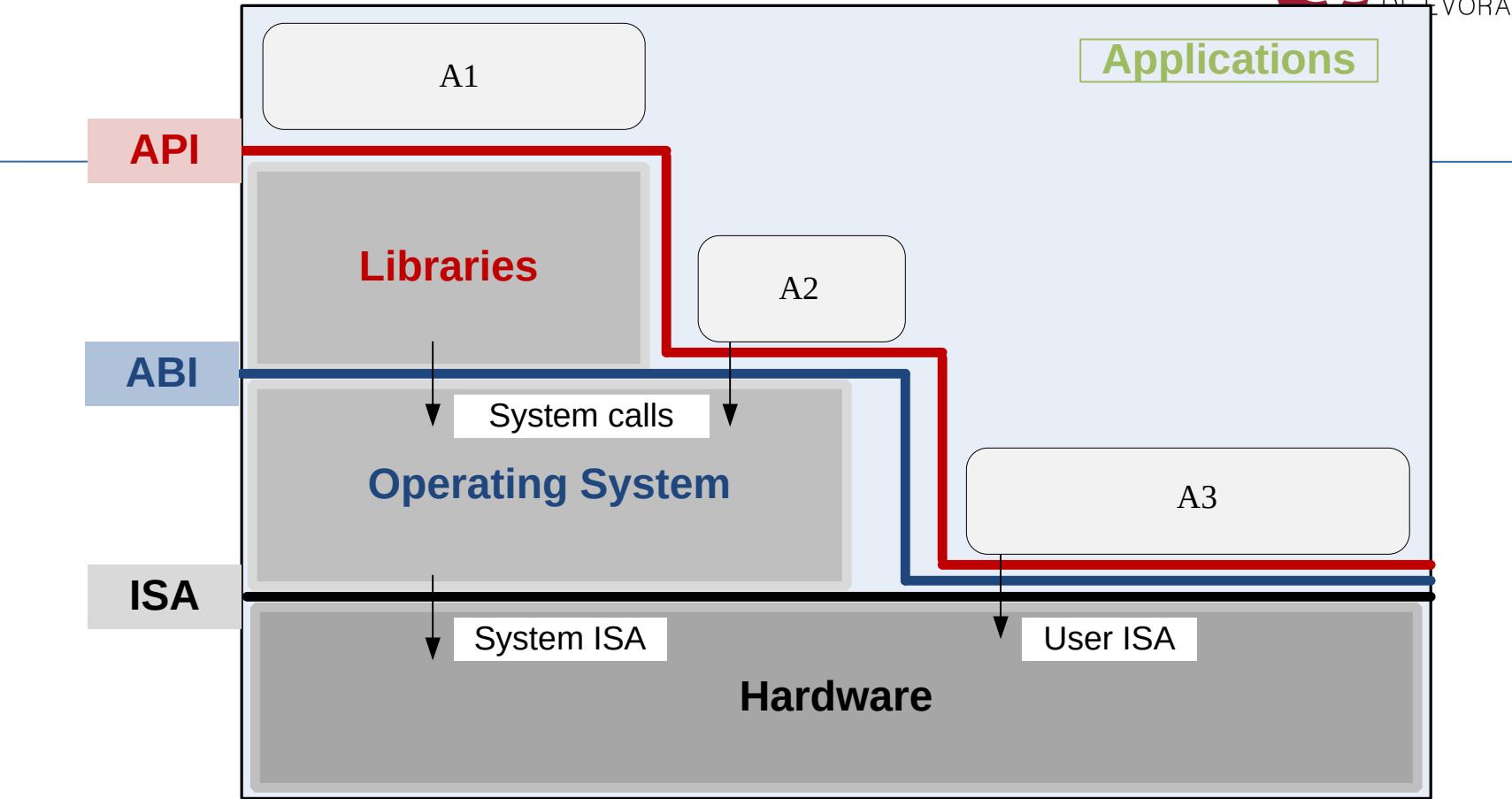
- Hardware.
- Software.
 - Operating system.
 - Libraries.
 - Applications.

Interfaces

Instruction Set Architecture (ISA) – at the boundary between hardware and software.

Application Binary Interface (ABI) – allows the ensemble consisting of the application and the library modules to access the hardware; the ABI does not include privileged system instructions, instead it invokes system calls.

Application Program Interface (API) - defines the set of instructions the hardware was designed to execute and gives the application access to the ISA; it includes HLL library calls which often invoke system calls.



Application Programming Interface, Application Binary Interface, and Instruction Set Architecture . An application uses library functions (A1), makes system calls (A2), and executes machine instructions (A3).

Code portability

Binaries created by a compiler for a specific ISA and a specific operating systems are not portable.

It is possible, though, to compile a HLL program for a virtual machine (VM) environment where portable code is produced and distributed and then converted by binary translators to the ISA of the host system.

A dynamic binary translation converts blocks of guest instructions from the portable code to the host instruction and leads to a significant performance improvement, as such blocks are cached and reused

Virtual machine Monitor (VMM / hypervisor)

Partitions the resources of computer system into one or more virtual machines (VMs). Allows several operating systems to run concurrently on a single hardware platform.

A VMM allows

- Multiple services to share the same platform.
- Live migration - the movement of a server from one platform to another.
- System modification while maintaining backward compatibility with the original system.
- Enforces isolation among the systems, thus security.

Hypervisor

Hypervisors are currently classified in two types:

Type 1 hypervisor (or Type 1 virtual machine monitor) is software that runs directly on a given hardware platform (as an operating system control program). A "guest" operating system thus runs at the second level above the hardware.

- The classic type 1 hypervisor was CP/CMS, developed at IBM in the 1960s, ancestor of IBM's current [z/VM](#). More recent examples are Xen, VMware's ESX Server, and Sun's Hypervisor (released in 2005).

Type 2 hypervisor (or Type 2 virtual machine monitor) is software that runs within an operating system environment. A "guest" operating system thus runs at the third level above the hardware.

- Examples include VMware server and Microsoft Virtual Server.

VMM virtualizes the CPU and the memory

A VMM

- Traps the privileged instructions executed by a guest OS and enforces the correctness and safety of the operation.
- Traps interrupts and dispatches them to the individual guest operating systems.
- Controls the virtual memory management.
- Maintains a shadow page table for each guest OS and replicates any modification made by the guest OS in its own shadow page table. This shadow page table points to the actual page frame and it is used by the Memory Management Unit (MMU) for dynamic address translation.
- Monitors the system performance and takes corrective actions to avoid performance degradation. For example, the VMM may swap out a Virtual Machine to avoid thrashing.

Virtual machines (VMs)

VM - isolated environment that appears to be a whole computer, but actually only has access to a portion of the computer resources.

Process VM - a virtual platform created for an individual process and destroyed once the process terminates.

System VM - supports an operating system together with many user processes.

Traditional VM - supports multiple virtual machines and runs directly on the hardware.

Hybrid VM - shares the hardware with a host operating system and supports multiple virtual machines.

Hosted VM - runs under a host operating system.

Name	Host ISA	Guest ISA	Host OS	guest OS	Company
Integrity VM	<i>x86-64</i>	<i>x86-64</i>	HP-Unix	Linux, Windows HP Unix	HP
Power VM	Power	Power	No host OS	Linux, AIX	IBM
z/VM	z-ISA	z-ISA	No host OS	Linux on z-ISA	IBM
Lynx Secure	<i>x86</i>	<i>x86</i>	No host OS	Linux, Windows	LinuxWorks
Hyper-V Server	<i>x86-64</i>	<i>x86-64</i>	Windows	Windows	Microsoft
Oracle VM	<i>x86, x86-64</i>	<i>x86, x86-64</i>	No host OS	Linux, Windows	Oracle
RTS Hypervisor	<i>x86</i>	<i>x86</i>	No host OS	Linux, Windows	Real Time Systems
SUN xVM	<i>x86, SPARC</i>	same as host	No host OS	Linux, Windows	SUN
VMware EX Server	<i>x86, x86-64</i>	<i>x86, x86-64</i>	No host OS	Linux, Windows Solaris, FreeBSD	VMware
VMware Fusion	<i>x86, x86-64</i>	<i>x86, x86-64</i>	MAC OS <i>x86</i>	Linux, Windows Solaris, FreeBSD	VMware
VMware Server	<i>x86, x86-64</i>	<i>x86, x86-64</i>	Linux, Windows	Linux, Windows Solaris, FreeBSD	VMware
VMware Workstation	<i>x86, x86-64</i>	<i>x86, x86-64</i>	Linux, Windows	Linux, Windows Solaris, FreeBSD	VMware
VMware Player	<i>x86, x86-64</i>	<i>x86, x86-64</i>	Linux Windows	Linux, Windows Solaris, FreeBSD	VMware
Denali	<i>x86</i>	<i>x86</i>	Denali	ILVACO, NetBSD	University of Washington
Xen	<i>x86, x86-64</i>	<i>x86, x86-64</i>	Linux Solaris	Linux, Solaris NetBSD	University of Cambridge

Performance and security isolation

The run-time behavior of an application is affected by other applications running concurrently on the same platform and competing for CPU cycles, cache, main memory, disk and network access. Thus, it is difficult to predict the completion time!

Performance isolation - a critical condition for QoS guarantees in shared computing environments.

A VMM is a much simpler and better specified system than a traditional operating system. Example - Xen has approximately 60,000 lines of code; Denali has only about half, 30,000.

The security vulnerability of VMMs is considerably reduced as the systems expose a much smaller number of privileged functions.

Computer architecture and virtualization

Conditions for efficient virtualization:

- A program running under the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
- The VMM should be in complete control of the virtualized resources.
- A statistically significant fraction of machine instructions must be executed without the intervention of the VMM.

Two classes of machine instructions:

- Sensitive - require special precautions at execution time:
 - Control sensitive - instructions that attempt to change either the memory allocation or the privileged mode.
 - Mode sensitive - instructions whose behavior is different in the privileged mode.
- Innocuous - not sensitive.

Full virtualization and paravirtualization

Full virtualization – a guest OS can run unchanged under the VMM as if it was running directly on the hardware platform.

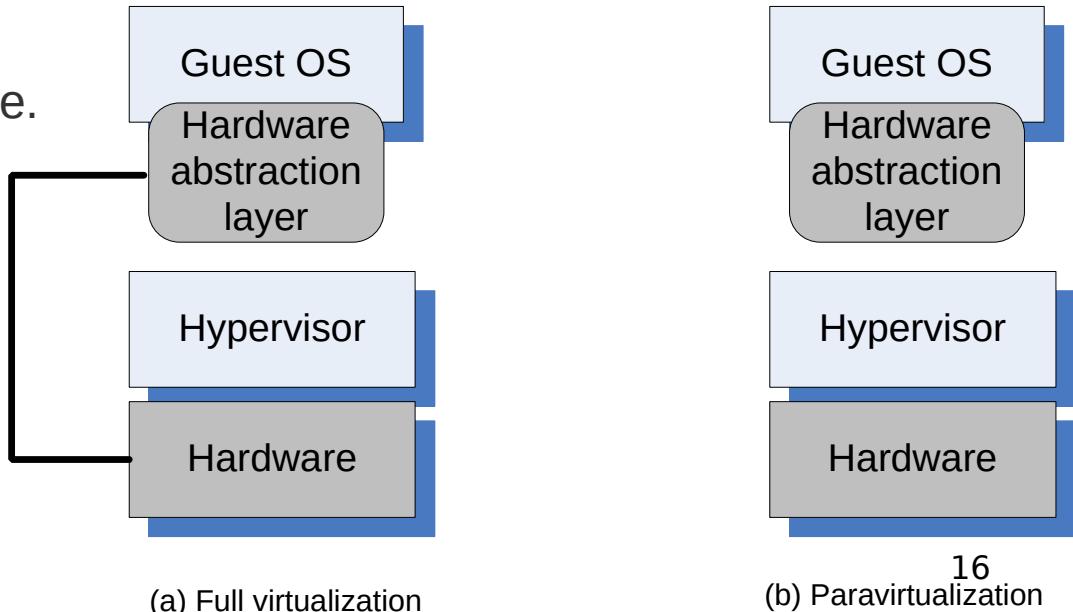
- Requires a virtualizable architecture.

Examples: Vmware.

Paravirtualization - a guest operating system is modified to use only instructions that can be virtualized. Reasons for paravirtualization:

- Some aspects of the hardware cannot be virtualized.
- Improved performance.
- Present a simpler interface.

Examples: Xen, Denaly



Protection Levels

protection levels also known as rings

0 has the highest level privilege and it is in this ring that the operating system kernel normally runs.

- Code executing in ring 0 is said to be running in system space, kernel mode or supervisor mode.
- All other code such as applications running on the operating system operates in less privileged rings, typically ring 3.

Virtualization of x86 architecture

Ring de-privileging - a VMM forces the operating system and the applications to run at a privilege level greater than 0.

Ring aliasing - a guest OS is forced to run at a privilege level other than that it was originally designed for.

Address space compression - a VMM uses parts of the guest address space to store several system data structures.

Non-faulting access to privileged state - several store instructions can only be executed at privileged level 0 because they operate on data structures that control the CPU operation. They fail silently when executed at a privilege level other than 0.

Guest system calls which cause transitions to/from privilege level 0 must be emulated by the VMM.

Interrupt virtualization - in response to a physical interrupt, the VMM generates a ``virtual interrupt'' and delivers it later to the target guest OS which can mask interrupts.

Virtualization of x86 architecture (cont'd)

Access to hidden state - elements of the system state, e.g., descriptor caches for segment registers, are hidden; there is no mechanism for saving and restoring the hidden components when there is a context switch from one VM to another.

Ring compression - paging and segmentation protect VMM code from being overwritten by guest OS and applications. Systems running in 64-bit mode can only use paging, but paging does not distinguish between privilege levels 0, 1, and 2, thus the guest OS must run at privilege level 3, the so called (0/3/3) mode. Privilege levels 1 and 2 cannot be used thus, the name ring compression.

The task-priority register is frequently used by a guest OS; the VMM must protect the access to this register and trap all attempts to access it. This can cause a significant performance degradation.

Linux KVM (Kernel Virtual Machine)

The most recent news out of Linux is the incorporation of the KVM into the Linux kernel (2.6.20).

KVM is a full virtualization solution that is unique in that it turns a Linux kernel into a hypervisor using a kernel module.

This module allows other guest operating systems to then run in user-space of the host Linux kernel (see Figure in the next slide).

The KVM module in the kernel exposes the virtualized hardware through the /dev/kvm character device.

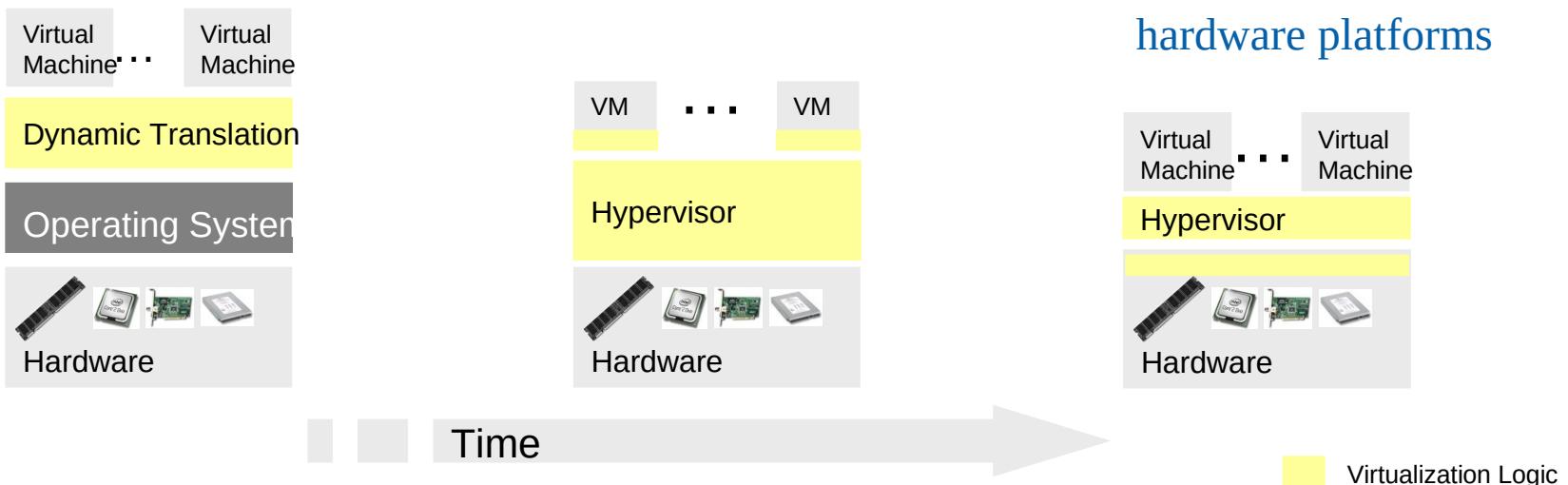
The guest operating system interfaces to the KVM module using a modified QEMU process for PC hardware emulation.

Virtualization Examples

- Disaster recovery
 - Virtual machines can be used as "hot standby" environments for physical production servers. This changes the classical "backup-and-restore" philosophy, by providing backup images that can "boot" into live virtual machines, capable of taking over workload for a production server experiencing an outage.
- Testing and training
 - Hardware virtualization can give root access to a virtual machine. This can be very useful such as in kernel development and operating system courses.
- Portable workspaces
 - Recent technologies have used virtualization to create portable workspaces on devices like USB memory sticks. Examples:
 - Thinstall
 - MojoPac, Ceedo
 - moka5 and LivePC

Evolution of Software solutions*

- **1st Generation:** Full virtualization (Binary rewriting)
 - Software Based
 - VMware and Microsoft
- **2nd Generation:** Paravirtualization
 - Cooperative virtualization
 - Modified guest
 - VMware, Xen
- **3rd Generation:** Silicon-based (Hardware-assisted) virtualization
 - Unmodified guest
 - VMware and Xen on virtualization-aware hardware platforms



The darker side of virtualization

In a layered structure, a defense mechanism at some layer can be disabled by malware running at a layer below it.

It is feasible to insert a *rogue VMM*, a Virtual-Machine Based Rootkit (VMBR) between the physical hardware and an operating system.

Rootkit - malware with a privileged access to a system.

The VMBR can enable a separate malicious OS to run surreptitiously and make this malicious OS invisible to the guest OS and to the application running under it.

Under the protection of the VMBR, the malicious OS could:

- observe the data, the events, or the state of the target system.
- run services, such as spam relays or distributed denial-of-service attacks.
- interfere with the application.

Credits, references and reading material

- *Cloud Computing: Theory and Practice*
Dan C. Marinescu
Chapter 5
- *Hypervisor, Virtualization Stack, And Device Virtualization Architectures*
Mike Neil, Microsoft Corporation
- *Distributed and Cloud Computing*
K. Hwang, G. Fox and J. Dongarra
Morgan Kaufmann, 2012

Cloud Computing Applications and Paradigms (and MapReduce)

Cloud applications

- Cloud computing is very attractive to the users:
 - Economic reasons.
 - low infrastructure investment.
 - low cost - customers are only billed for resources used.
 - Convenience and performance.
 - application developers enjoy the advantages of a just-in-time infrastructure; they are free to design an application without being concerned with the system where the application will run.
 - the execution time of compute-intensive and data-intensive applications can, potentially, be reduced through parallelization. If an application can partition the workload in n segments and spawn n instances of itself, then the execution time could be reduced by a factor close to n .
- Cloud computing is also beneficial for the providers of computing cycles - it typically leads to a higher level of resource utilization.

Challenges for cloud application development

- Performance isolation - nearly impossible to reach in a real system, especially when the system is heavily loaded.
- Reliability - major concern; server failures expected when a large number of servers cooperate for the computations.
- Cloud infrastructure exhibits latency and bandwidth fluctuations which affect the application performance.
- Performance considerations limit the amount of *data logging*; the ability to identify the source of unexpected results and errors is helped by frequent logging.

Existing and new application opportunities

- Three broad categories of existing applications:
 - Processing pipelines.
 - Batch processing systems.
 - Web applications.

- Potentially new applications
 - Batch processing for decision support systems and business analytics.
 - Mobile interactive applications which process large volumes of data from different types of sensors.
 - Science and engineering could greatly benefit from cloud computing as many applications in these areas are compute-intensive and data-intensive.

Processing pipelines

- Indexing large datasets created by web crawler engines.
- Data mining - searching large collections of records to locate items of interests.
- Image processing .
 - Image conversion, e.g., enlarge an image or create thumbnails.
 - Compress or encrypt images.
- Video transcoding from one video format to another, e.g., from AVI to MPEG.
- Document processing.
 - Convert large collections of documents from one format to another, e.g., from Word to PDF.
 - Encrypt documents.
 - Use Optical Character Recognition to produce digital images of documents.

Architectural styles for cloud applications

- Based on the client-server paradigm.
- Stateless servers - view a client request as an independent transaction and respond to it; the client is not required to first establish a connection to the server.
- Often clients and servers communicate using Remote Procedure Calls (RPCs).
- Simple Object Access Protocol (SOAP) - application protocol for web applications; message format based on the XML. Uses TCP or UDP transport protocols.
- Representational State Transfer (REST) - software architecture for distributed hypermedia systems. Supports client communication with stateless servers, it is platform independent, language independent, supports data caching, and can be used in the presence of firewalls.

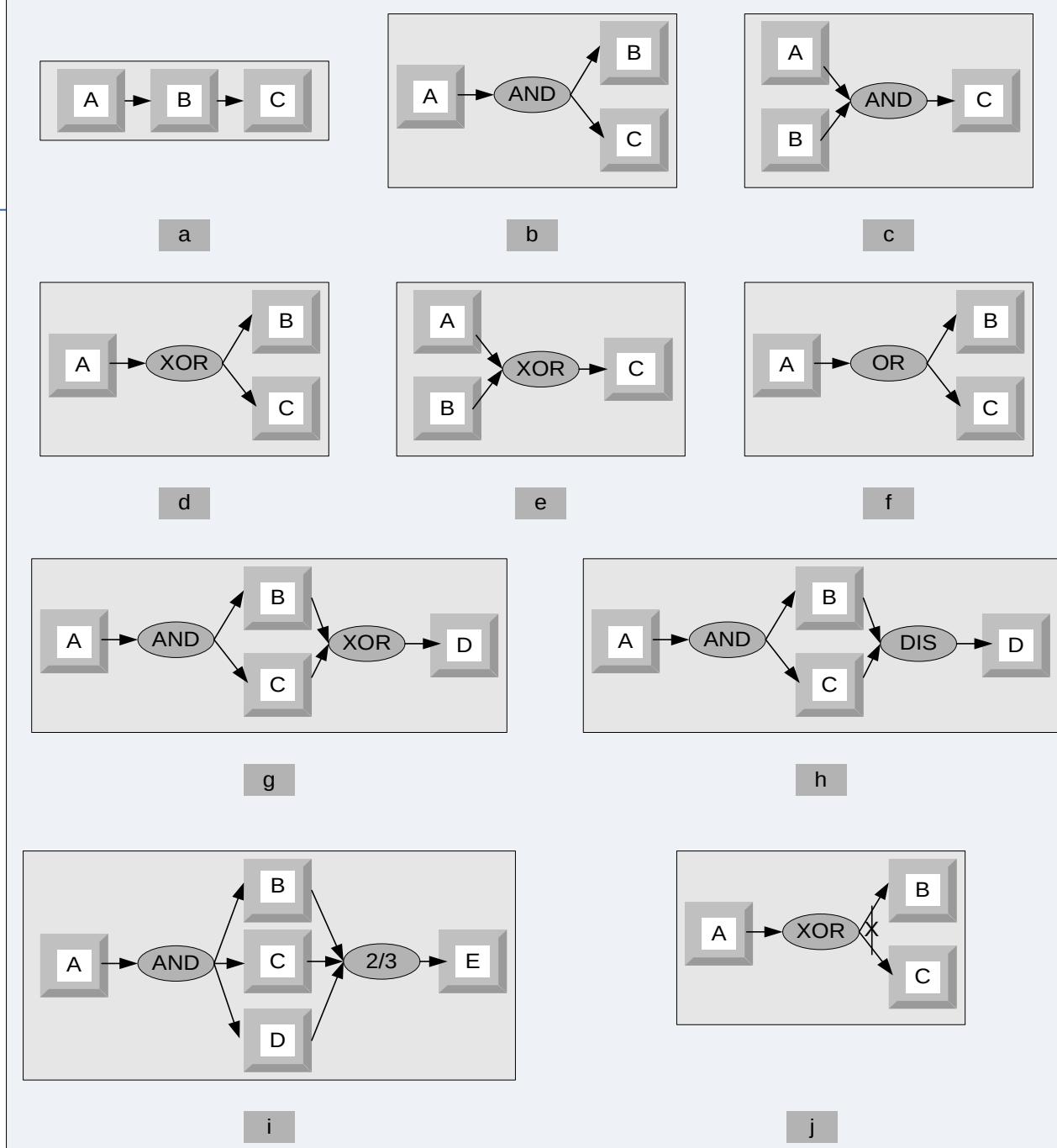
Basic workflow patterns

- Workflow patterns - the temporal relationship among the tasks of a process
 - Sequence - several tasks have to be scheduled one after the completion of the other.
 - AND split - both tasks B and C are activated when task A terminates.
 - Synchronization - task C can only start after tasks A and B terminate.
 - XOR split - after completion of task A, either B or C can be activated.
 - XOR merge - task C is enabled when either A or B terminate.
 - OR split - after completion of task A one could activate either B, C, or both.
 - Multiple Merge - once task A terminates, B and C execute concurrently; when the first of them, say B, terminates, then D is activated; then, when C terminates, D is activated again.
 - Discriminator – wait for a number of incoming branches to complete before activating the subsequent activity; then wait for the remaining branches to finish without taking any action until all of them have terminated. Next, resets itself.

Basic workflow patterns (cont'd)

- N out of M join - barrier synchronization. Assuming that M tasks run concurrently, N ($N < M$) of them have to reach the barrier before the next task is enabled. In our example, any two out of the three tasks A, B, and C have to finish before E is enabled.
- Deferred Choice - similar to the XOR split but the choice is not made explicitly; the run-time environment decides what branch to take.

Cloud Computing: Theory and Practice.
Chapter 4



Coordination - ZooKeeper

- Cloud elasticity → distribute computations and data across multiple systems; coordination among these systems is a critical function in a distributed environment.
- ZooKeeper
 - Distributed coordination service for large-scale distributed systems.
 - High throughput and low latency service.
 - Implements a version of the Paxos consensus algorithm.
 - Open-source software written in Java with bindings for Java and C.
 - The servers in the pack communicate and elect a leader.
 - A database is replicated on each server; consistency of the replicas is maintained.
 - A client connects to a single server, synchronizes its clock with the server, and sends requests, receives responses and watch events through a TCP connection.

Zookeeper communication

- Messaging layer → responsible for the election of a new leader when the current leader fails.
- Messaging protocols use:
 - Packets - sequence of bytes sent through a FIFO channel.
 - Proposals - units of agreement.
 - Messages - sequence of bytes atomically broadcast to all servers.
- A message is included into a proposal and it is agreed upon before it is delivered.
- Proposals are agreed upon by exchanging packets with a quorum of servers, as required by the Paxos algorithm.

Zookeeper communication (cont'd)

■ Messaging layer guarantees:

- Reliable delivery: if a message **m** is delivered to one server, it will be eventually delivered to all servers.
- Total order: if message **m** is delivered before message **n** to one server, it will be delivered before **n** to all servers.
- Causal order: if message **n** is sent after **m** has been delivered by the sender of **n**, then **m** must be ordered before **n**.

ZooKeeper service guarantees

- Atomicity - a transaction either completes or fails.
- Sequential consistency of updates - updates are applied strictly in the order they are received.
- Single system image for the clients - a client receives the same response regardless of the server it connects to.
- Persistence of updates - once applied, an update persists until it is overwritten by a client.
- Reliability - the system is guaranteed to function correctly as long as the majority of servers function correctly.

Zookeeper API

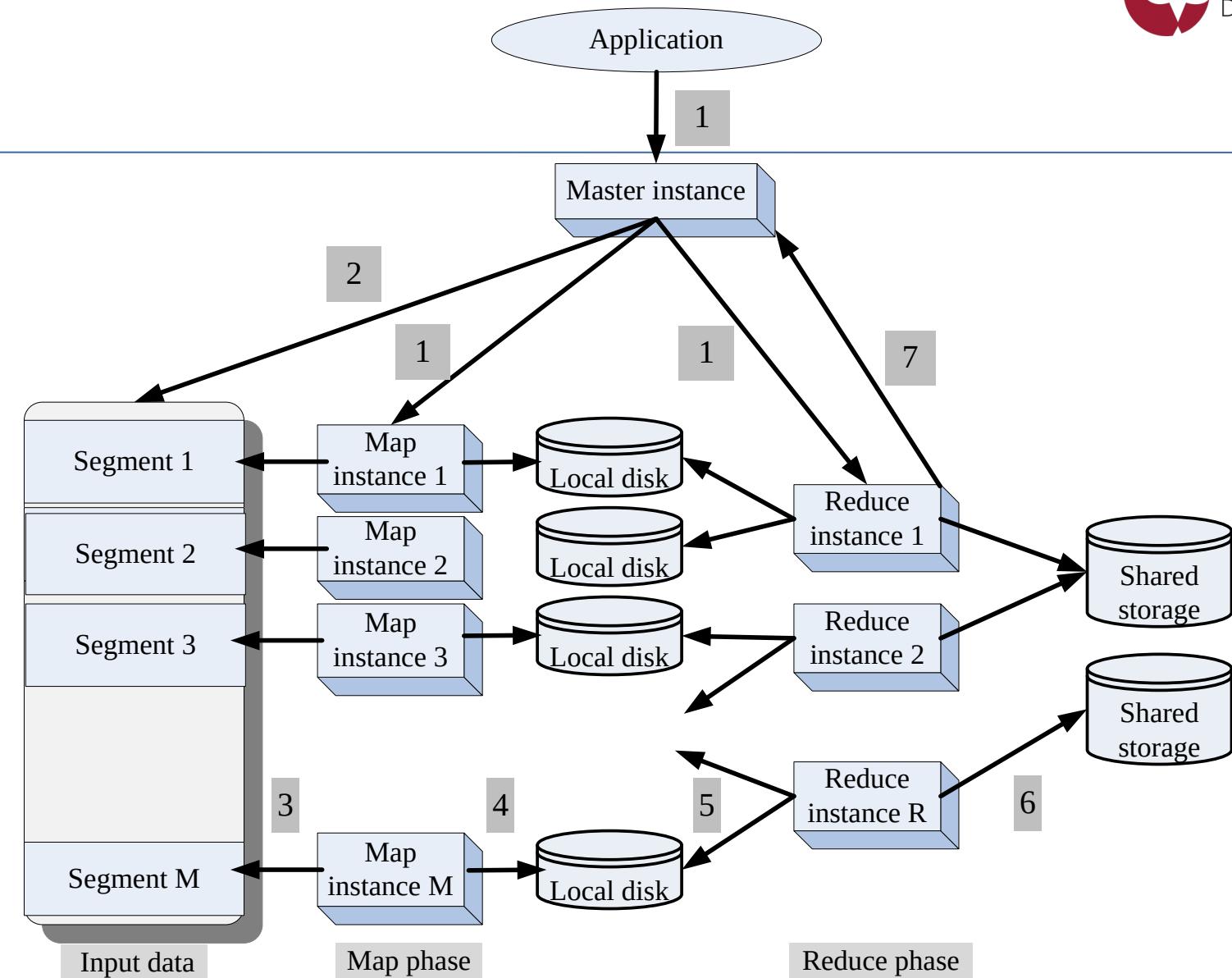
- The API is simple - consists of seven operations:
 - Create - add a node at a given location on the tree.
 - Delete - delete a node.
 - Get data - read data from a node.
 - Set data - write data to a node.
 - Get children - retrieve a list of the children of the node.
 - Synch - wait for the data to propagate.

Elasticity and load distribution

- Elasticity → ability to use as many servers as necessary to optimally respond to cost and timing constraints of an application.
- How to divide the load
 - Transaction processing systems → a front-end distributes the incoming transactions to a number of back-end systems. As the workload increases new back-end systems are added to the pool.
 - For data-intensive batch applications two types of divisible workloads are possible:
 - modularly divisible → the workload partitioning is defined a priori.
 - arbitrarily divisible → the workload can be partitioned into an arbitrarily large number of smaller workloads of equal, or very close size.
- Many applications in physics, biology, and other areas of computational science and engineering obey the arbitrarily divisible load sharing model.

MapReduce philosophy

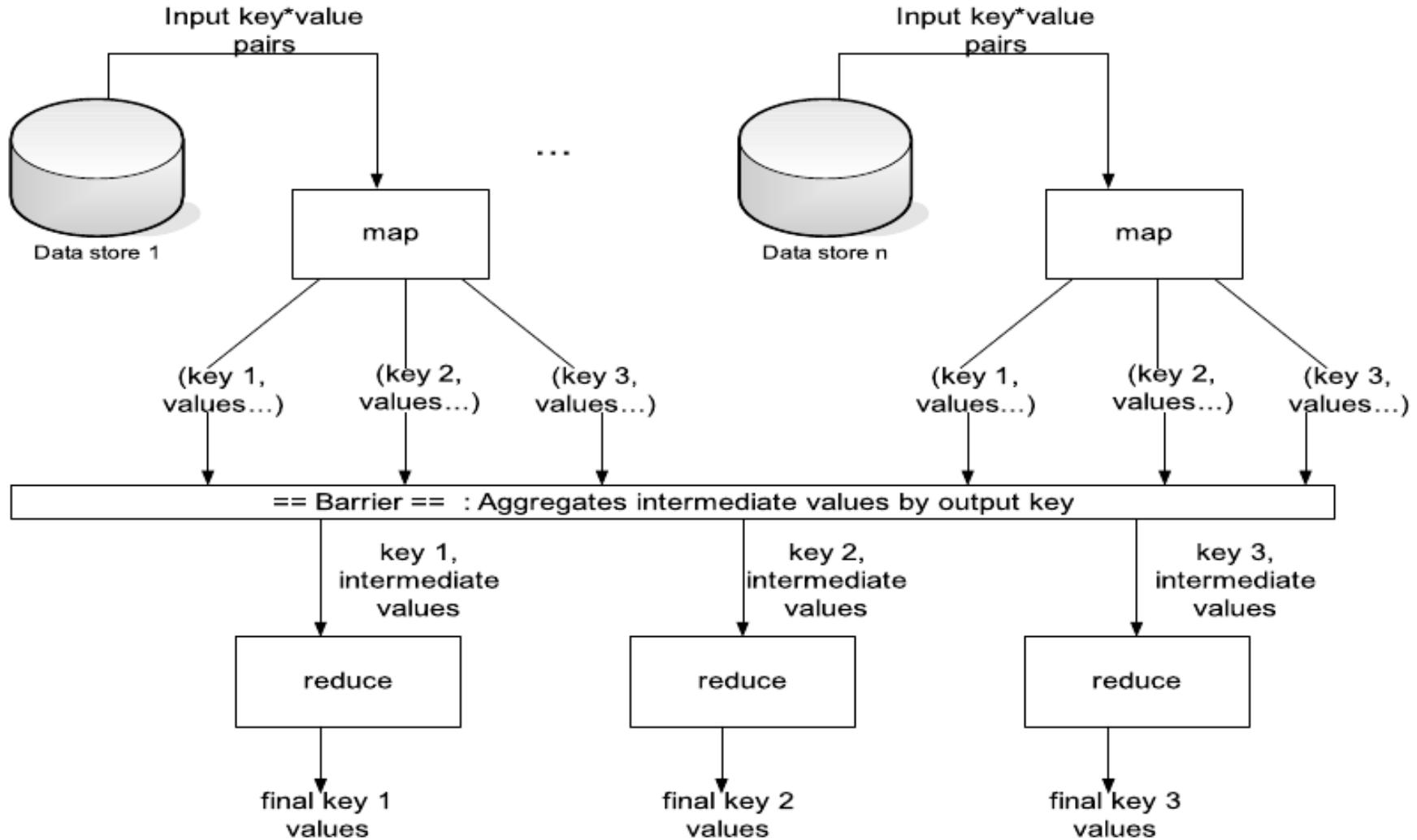
1. An application starts a master instance, M worker instances for the *Map phase* and later R worker instances for the *Reduce phase*.
2. The master instance partitions the input data in M segments.
3. Each *map instance* reads its input data segment and processes the data.
4. The results of the processing are stored on the local disks of the servers where the map instances run.
5. When all map instances have finished processing their data, the R reduce instances read the results of the first phase and merge the partial results.
6. The final results are written by the reduce instances to a shared storage server.
7. The master instance monitors the reduce instances and when all of them report task completion the application is terminated.



MapReduce philosophy

- Inspired by **map** and **reduce** primitives in functional programming
 - mapping a function f over a sequence $x \ y \ z$
 - yields $f(x) \ f(y) \ f(z)$
 - reduce combines sequence of elements using a binary op
- Many data analysis computations can be expressed as
 - applying a map operation to each logical input record
 - produce a set of intermediate (key, value) pairs
 - applying a reduce to all intermediate pairs with same key

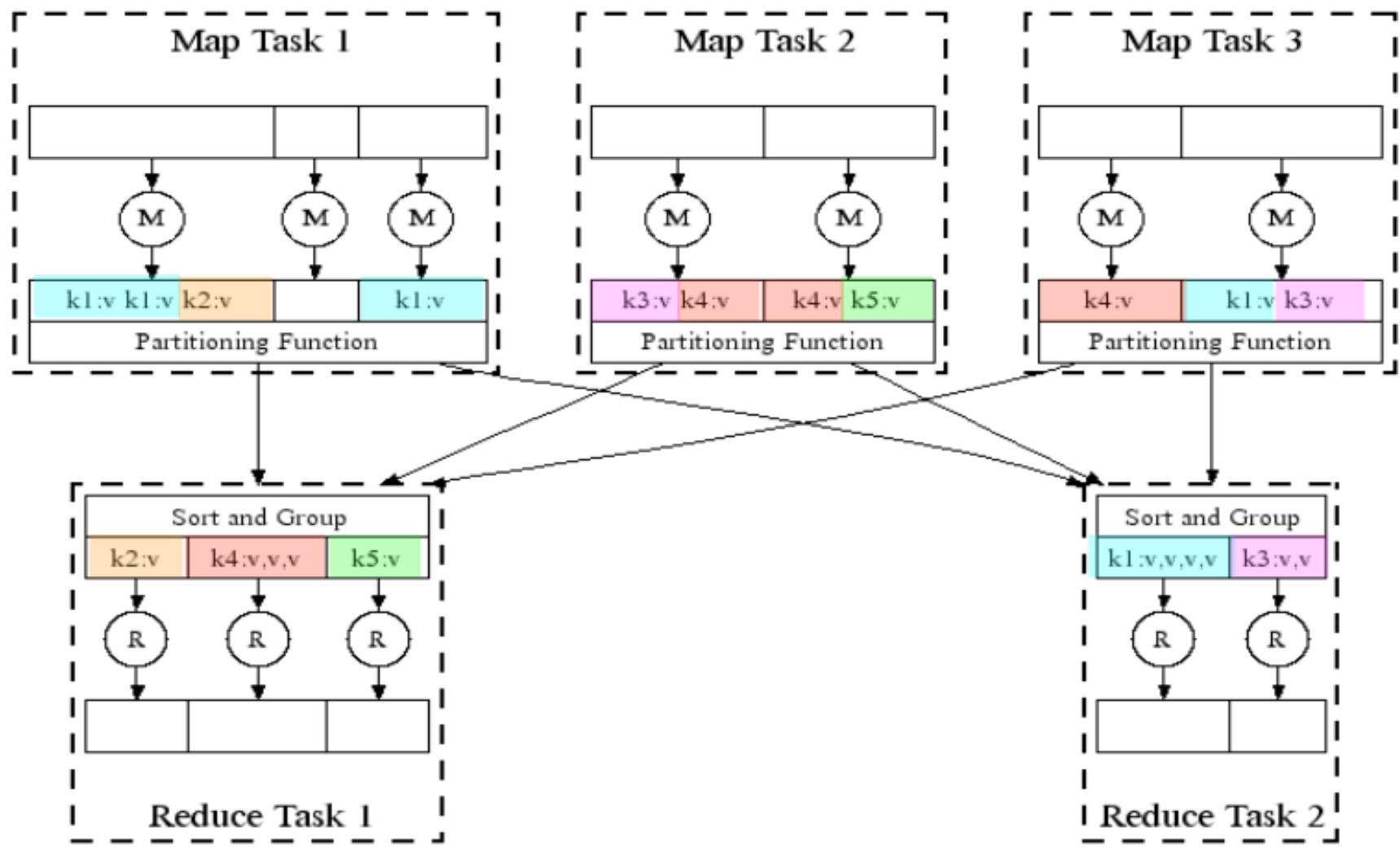
MapReduce: logical view of execution



Case study: GrepTheWeb

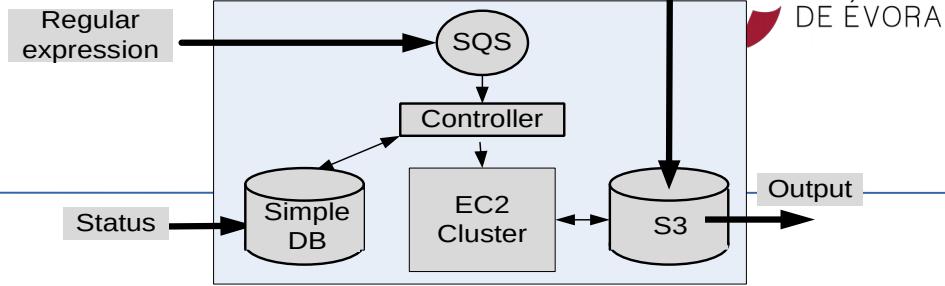
- The application illustrates the means to
 - create an on-demand infrastructure.
 - run it on a massively distributed system in a manner that allows it to run in parallel and scale up and down, based on the number of users and the problem size.
- GrepTheWeb
 - Performs a search of a very large set of records to identify records that satisfy a regular expression.
 - It is analogous to the Unix *grep* command.
 - The source is a collection of document URLs produced by the Alexa Web Search, a software system that crawls the web every night.
 - Uses message passing to trigger the activities of multiple controller threads which launch the application, initiate processing, shutdown the system, and create billing records.

MapReduce: example data flow after map phase



(a) The simplified workflow showing the inputs:

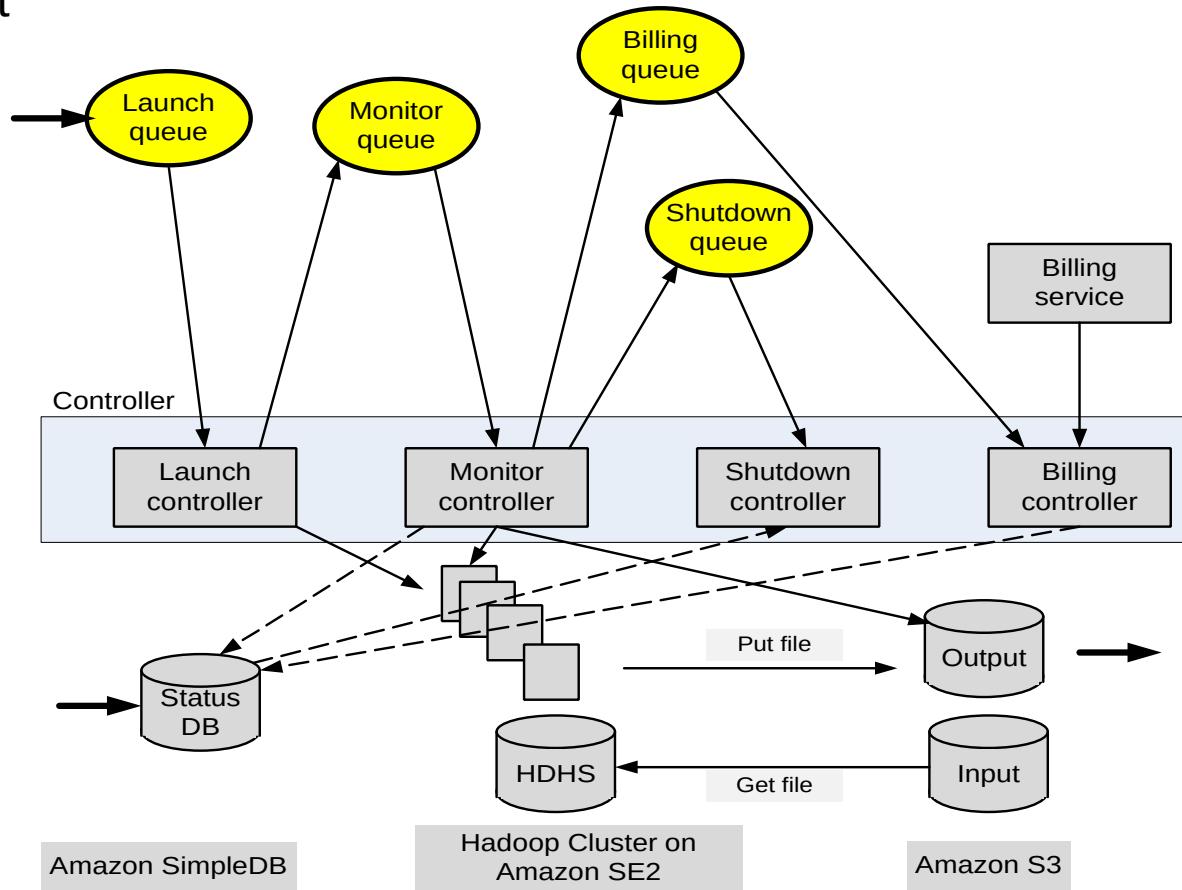
- the regular expression.
- the input records generated by the web crawler.
- the user commands to report the current status and to terminate the processing.



(a)

(b) The detailed workflow.

The system is based on message passing between several queues; four controller threads periodically poll their associated input queues, retrieve messages, and carry out the required actions

Hadoop Cluster on
Amazon SE2

Amazon S3

Apache Hadoop



- framework **distributed processing of large data sets**, across clusters of computers, using **simple programming models**
 - An open-source implementation of MapReduce
- is an ecosystem composed of modules for computing, storage and coordination in a distributed system:
 - Hadoop Common: The common utilities that support the other Hadoop modules.
 - Hadoop Distributed File System (HDFS™): A distributed file system that provides high-throughput access to application data.
 - Hadoop YARN: A framework for job scheduling and cluster resource management.
 - Hadoop MapReduce: A YARN-based system for parallel processing of large data sets.
 - Other Hadoop-related projects:
 - ZooKeeper, Pig, Hive, Cassandra...

Clouds for science and engineering

- The generic problems in virtually all areas of science are:
 - Collection of experimental data.
 - Management of very large volumes of data.
 - Building and execution of models.
 - Integration of data and literature.
 - Documentation of the experiments.
 - Sharing the data with others; data preservation for a long periods of time.
- All these activities require “big” data storage and systems capable to deliver abundant computing cycles.
Computing clouds are able to provide such resources and support collaborative environments.

Social computing and digital content

- Networks allowing researchers to share data and provide a virtual environment supporting remote execution of workflows are domain specific:
 - MyExperiment for biology.
 - nanoHub for nanoscience.
- Volunteer computing - a large population of users donate resources such as CPU cycles and storage space for a specific project:
 - Mersenne Prime Search
 - SETI@Home,
 - Folding@home,
 - Storage@Home
 - PlanetLab
- Berkeley Open Infrastructure for Network Computing (BOINC) → middleware for a distributed infrastructure suitable for different applications.

Credits, references and reading material

- *Cloud Computing: Theory and Practice*
Dan C. Marinescu
- <http://research.google.com/archive/mapreduce.html>
- <http://hadoop.apache.org/>
- Hadoop: The Definitive Guide – 3rd edition
Tom White. O'Reilly Media. May 2012
- *Data Analysis with MapReduce*
John Mellor-Crummey, DCS, Rice University

