RESUMO DO LIVRO *DISTRIBUTED SYSTEMS*

Capítulo 1

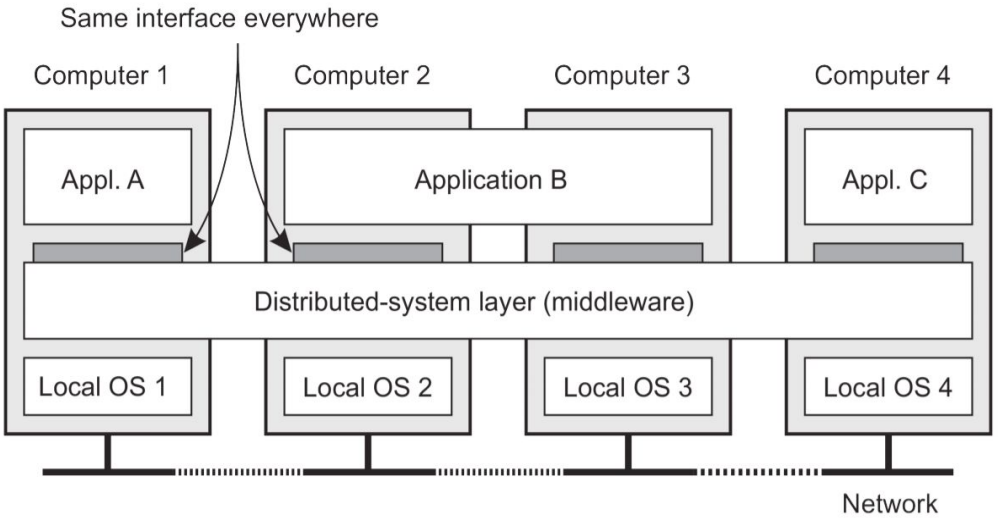
Sistema distribuído:

Consiste num conjunto de computadores autónomos (cada um sendo um nó) que trabalham em conjunto, de forma a parecerem um único sistema coerente. Os nós trocam mensagens para atingirem um objetivo comum. Existem dois tipos de grupos:

1. Open group - cada nó pode enviar mensagens para qualquer outro nó do sistema.
2. Closed group - apenas os nós membros daquele grupo podem comunicar entre si, sendo necessário um mecanismo secundário, de forma a permitir que um nó se junte ou abandone um determinado grupo.

Um sistema, além de único, deve ser coerente, e só o será se se comportar da forma que é esperada pelos seus utilizadores.

Um sistema distribuído pode ser atingido através da recolha de protocolos independentes para a **middleware-** camada de software logicamente colocada entre os sistemas operativos e as aplicações distribuídas. Os protocolos aí incluídos são de comunicação, transição, composição de serviço e confiabilidade.
A *middleware* funciona como gestora de recursos, oferecendo às suas aplicações uma forma eficiente de partilhar e instalar os mesmos por toda a rede.



As metas de desenvolvimento de um sistema distribuído incluem a partilha de recursos e garantia de abertura. Os *designers* procuram esconder as complexidades existentes, mas manter os recursos com fácil acesso. Ainda assim, a transparência traz custo de performance e pode nunca ser alcançada.
Um aspeto específico, difícil de alcançar, é a escalabilidade - isto acontece, por exemplo, na escalabilidade geográfica, onde esconder a latência e restrições de largura de banda pode ser quase impossível.

Transparência:

Procura-se esconder o facto que os processos e recursos de um sistema estão distribuídos, fisicamente, de forma transversal, ao longo de computadores (possivelmente separados por longas distâncias).

Distribuição transparente:

Transparência	Descrição
Acesso	Esconde diferenças na representação dos dados e na forma como os mesmos são acedidos
Localização	Esconde onde um objecto está localizado
Relocalização	Esconde que um objecto pode ser movido para outra localização enquanto usado
Migração	Esconde que um objecto pode ser movido para outra localização
Replicação	Esconde que um objecto pode ser replicado
Concorrência	Esconde que um objecto pode ser partilhado entre diversos utilizadores concorrentes
Falhas	Esconde as falhas e recuperações de um objecto

Abertura:

Um sistema oferece componentes que podem facilmente ser usados, ou integrados, por outros sistemas. Devem ser seguidos 3 princípios:

1. Inter-operação: duas implementações de um sistema ou componentes de diferentes construtores podem coexistir e trabalhar em conjunto.
2. Portabilidade: um sistema distribuído A pode ser executado, sem modificações, num sistema distribuído diferente B, que implementa as mesmas interfaces que A.
3. Extensibilidade: um sistema deve ser fácil de configurar com componentes diferentes (possivelmente de outros criadores).

Escalabilidade:

Pode ser medida consoante 3 diferentes dimensões:

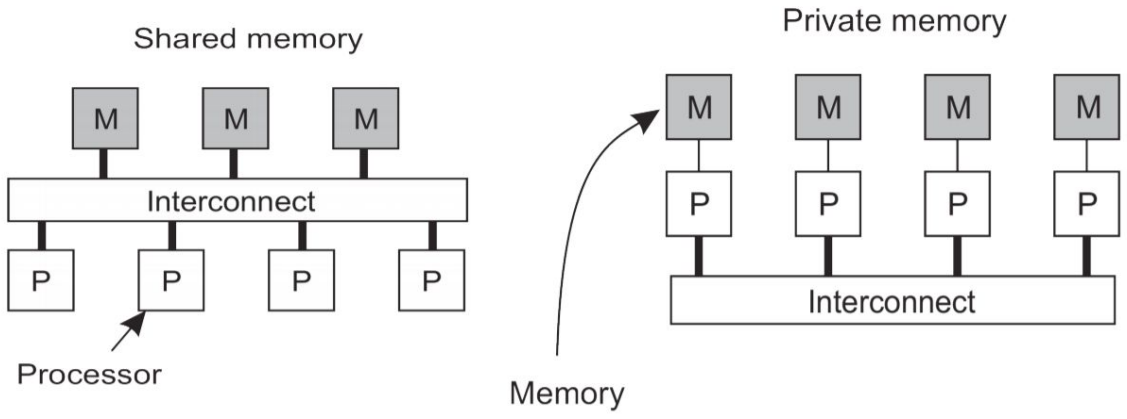
1. Tamanho: podemos, facilmente, adicionar mais utilizadores e recursos ao sistema, sem perdas notáveis ou perda de performance. O problema aqui é que, dado um servidor, ou um grupo de servidores, pode dar-se um *bottleneck* (atraso na transmissão de informação) aquando de um elevado número de pedidos.
2. Geográfica: os utilizadores e recursos podem estar bastante distantes, mas o facto de que a comunicação pode ter atraso não é notável. Um dos problemas, aqui, é o facto da comunicação ser síncrona - um cliente bloqueia até que tenha uma resposta do servidor, a implementar o serviço. Outro problema é que comunicação em redes grandes é muito menos fiável do que em redes locais.
3. Administrativa: o sistema pode ser facilmente gerenciado, mesmo que abranja várias organizações administrativas independentes. Um dos problemas são os conflitos entre políticas.

Pitfalls:

Falsas suposições que todos fazem ao desenvolver uma aplicação distribuída pela primeira vez:

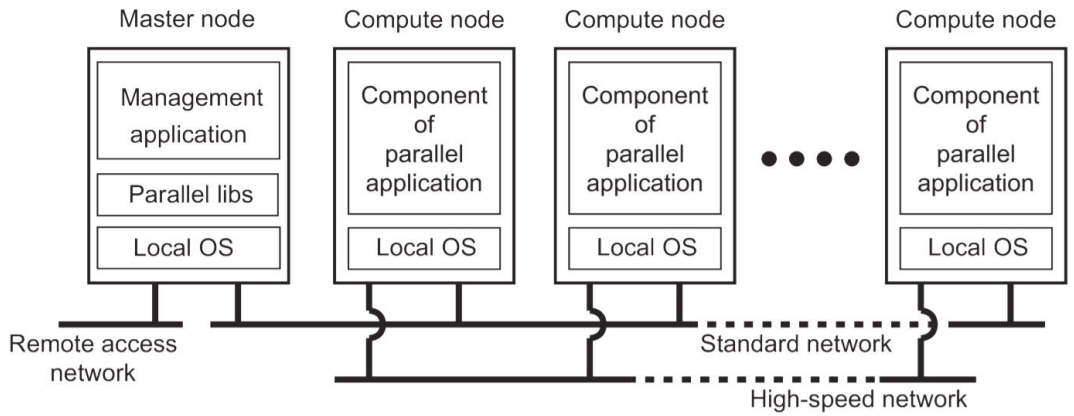
- A rede é confiável, segura e homogénea;
- A topologia não muda;
- A latência é 0;
- A largura de banda é infinita;
- O custo de transporte é 0;
- Existe um único administrador.

Arquiteturas de software:



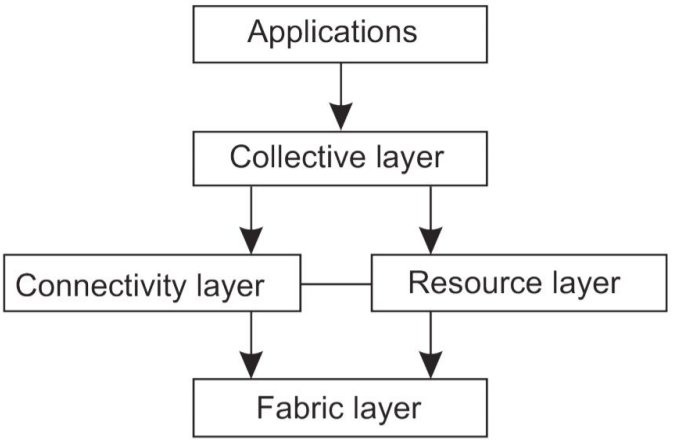
- Sistemas multi-processador: múltiplos CPUs estão organizados de forma a que todos têm acesso à mesma memória física.
- Sistemas multi-computador: vários computadores estão ligados através de uma rede e, nesta, não existe partilha da memória principal.

Arquitetura Cluster Computing:



Usado para programar em paralelo (tendo em vista *high performance*), onde um único programa corre em paralelo, em múltiplas máquinas.
É caracterizado pela sua homogeneidade. Os computadores de um *cluster*, por norma, têm o mesmo SO e estão conectados através da mesma rede.

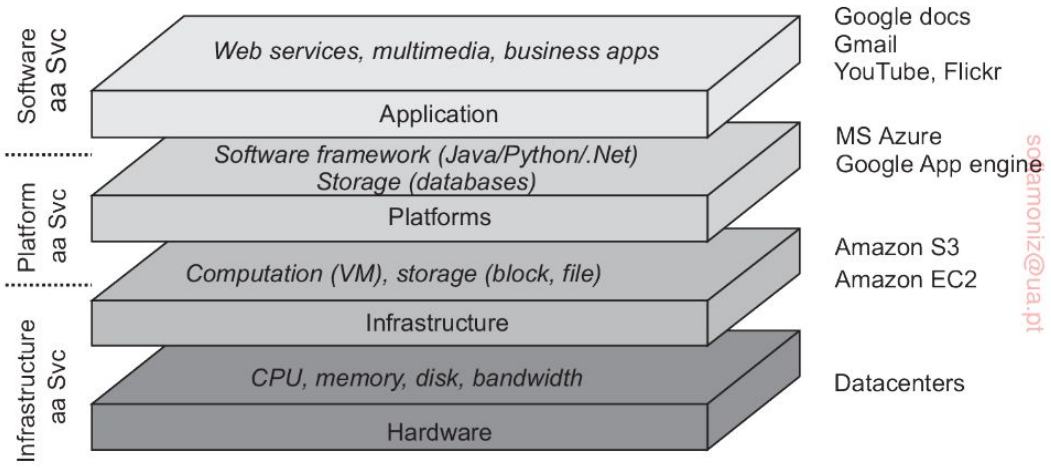
Arquitetura Grid Computing:



O problema chave, aqui, é o facto de os recursos, de diferentes organizações, serem reunidos de forma a permitir a colaboração de um grupo de pessoas de diferentes instituições. Esta colaboração é feita na forma de organização virtual- os processos que pertencem à mesma organização virtual têm acesso aos recursos que são fornecidos àquela organização. Constituída por 4 camadas:

1. Fabric layer: fornece interfaces para recursos locais, numa página específica.
2. Connectivity layer: consiste na comunicação de protocolos, de forma a suportar transações de rede que abrangem o uso de múltiplos recursos.
3. Resource layer: responsável por gerir um único recurso.
4. Collective layer: lida com o processo de manusear o acesso a múltiplos recursos, consistindo em serviços para descoberta de recursos, alocação e calendarização de tarefas.
5. Application layer: aplicações que operam em organização virtual, fazendo uso da rede do ambiente computacional.

Arquitetura Cloud Computing



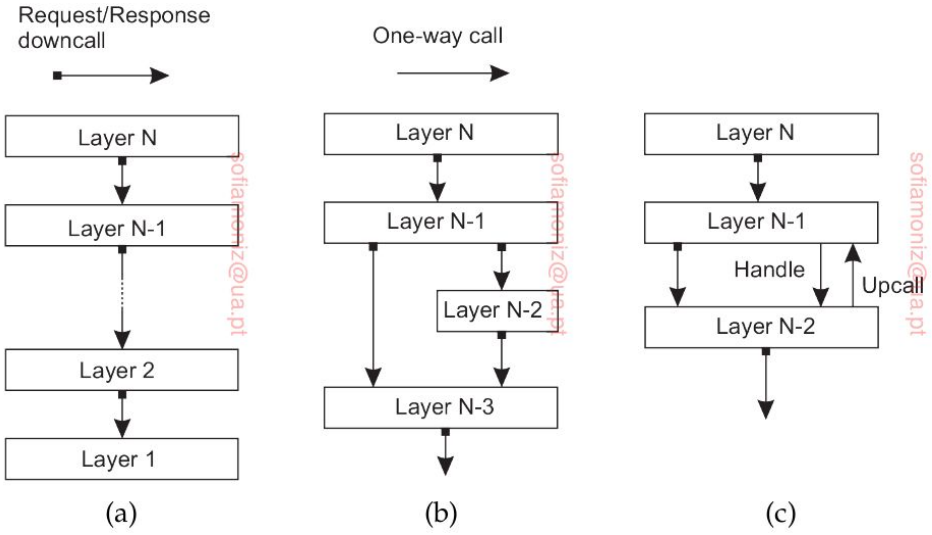
"Piscina" de recursos virtualizados, facilmente usáveis e acessíveis, fornecendo a base de escalabilidade - se mais trabalho necessitar de ser feito, um cliente pode, simplesmente, adquirir mais recursos. Está organizada em 4 camadas:

1. Hardware: implementada em *data centers* e contém os recursos que os clientes normalmente nunca veem diretamente.
2. Infraestrutura: instala técnicas de virtualização, de forma a fornecer aos utilizadores uma infraestrutura que consiste em armazenamento virtual e recursos computacionais. A *Cloud Computing* desenvolve-se em torno de alocação e administração virtual de dispositivos de armazenamento e servidores virtuais.
3. Plataforma: disponibiliza abstrações de alto nível para armazenamento e afins.
4. Aplicação: as aplicações correm nesta camada e são oferecidas aos utilizadores para customização adicional.

Capítulo 2

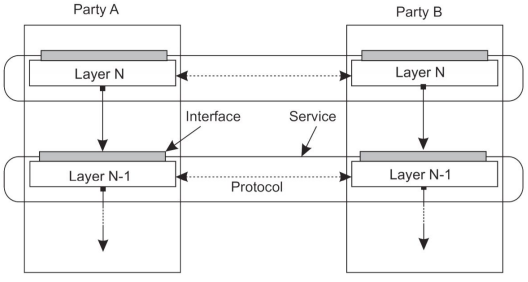
Os sistemas distribuídos podem ser organizados de diferentes formas. Pode ser feita a distinção entre arquitetura de software e arquitetura do sistema. A arquitetura do sistema considera uma arquitetura onde os componentes, que constituem um sistema distribuído, estão localizados ao longo de várias máquinas. A arquitetura de software está mais preocupada com a organização lógica do software.
O estilo da arquitetura reflete o princípio básico que é seguido na organização das interações entre componentes de software.

Arquitetura por camadas:



Uma camada mais alta pode fazer um *downcall* para uma camada mais baixa, esperando uma resposta. Apenas em casos excecionais pode ser feita uma *upcall*, para um componente de um nível mais alto.

- Fig.a): organização onde apenas *downcalls* para a próxima camada mais baixa são feitas.
- Fig.b): quando uma aplicação A faz uso de uma biblioteca, de forma a interagir com o SO. Ao mesmo tempo, a aplicação usa uma outra biblioteca.
- Fig.c): por vezes, é conveniente ter uma camada mais baixa, de forma a fazer um *upcall* para a próxima camada mais alta.

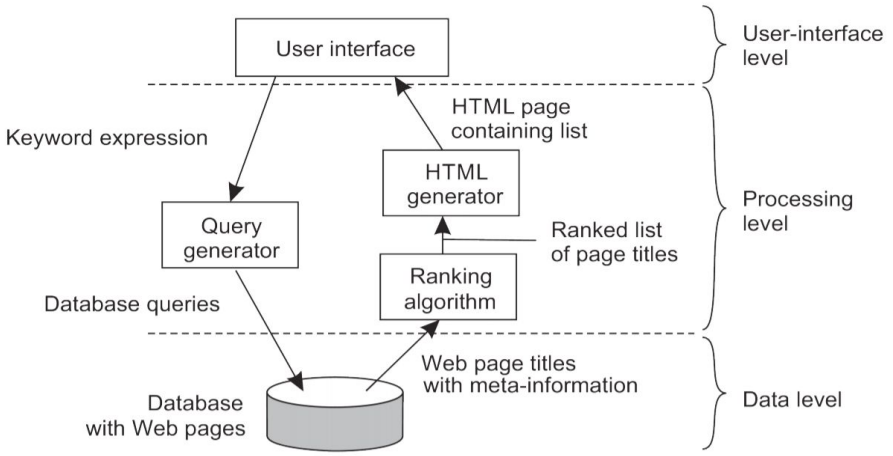


Camadas aplicacionais:

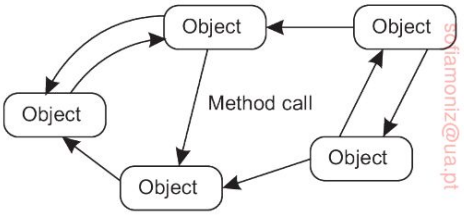
Foi feita a distinção entre 3 níveis lógicos seguindo, essencialmente, uma arquitetura em camadas:

1. Camada de interface: trata da interação com o utilizador ou alguma aplicação externa;
2. Camada de processamento: contém a funcionalidade crucial da aplicação;
3. Camada de dados: opera numa base de dados ou ficheiro de sistema.

Um exemplo passa pelo uso de um motor de pesquisa. O utilizador escreve uma série de *strings*, e é-lhe retornado um conjunto de páginas. O *back end* é formado por uma extensa base de dados de páginas web. A funcionalidade crucial passa pelo programa que transforma as *strings* do utilizador em *queries* de base de dados. Subsequentemente, avalia os resultados numa lista, e transforma essa lista numa série de páginas *HTML*:

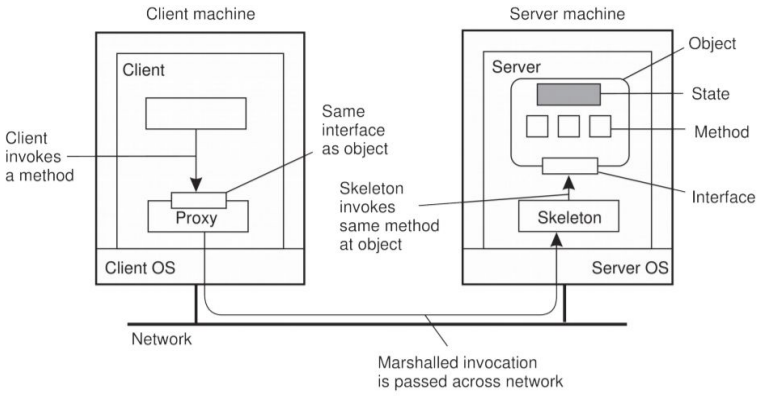


Arquiteturas baseadas em objetos e orientadas a serviço:



Cada objeto corresponde a um componente, sendo que estes componentes estão conectados através de um mecanismo de *procedure call*- a chamada ao objeto não precisa de ser executada na mesma máquina onde se situa o objeto chamado.

A interface oferecida por um objeto oculta os detalhes de implementação - um objeto é completamente independente do seu ambiente.
Esta separação entre interfaces e objetos permite colocar as interfaces numa máquina, enquanto o objeto em si está localizado noutra máquina - *distributed object*.



Quando um cliente faz *bind* de um objeto distribuído, a implementação da interface de um objeto - ***proxy***- é carregada para o espaço de morada do cliente. A *proxy* faz *marshal* dos métodos invocados para mensagens, e realiza *unmarshal* às mensagens de resposta, de forma a retornar o resultado do método invocado ao cliente. O objeto em causa reside na máquina do servidor, que oferece a mesma interface que a máquina do cliente.

Do lado do servidor surge uma entidade, o ***skeleton*** - permite que a *middleware* do servidor tenha acesso aos objetos definidos pelo utilizador. Por vezes, estes contêm código incompleto, na forma de uma classe de linguagem específica, que precisa de ser especializada mais profundamente, pelo desenvolvedor.

Apenas as interfaces implementadas pelo objeto ficam disponíveis noutras máquinas - ***remote objects***. Num objeto distribuído, em geral, o seu estado pode estar fisicamente distribuído por várias máquinas, mas também esta distribuição está escondida dos clientes, por detrás da interface do objeto.

Arquiteturas baseadas em recursos:

Um sistema distribuído pode ser visto como uma coleção enorme de recursos que, individualmente, são gerenciados pelos componentes. Os recursos podem ser adicionados ou removidos por aplicações remotas e, desta forma, também recuperados ou modificados. Este conceito foi adotado como **Representational State Transfer (REST)**. As arquiteturas que seguem este conceito apresentam 4 características:

1. Os recursos são identificados através de um esquema de identificadores único;
2. Todos os servidores oferecem a mesma interface;
3. As mensagens são enviadas de/para um serviço que é completamente auto descritivo;
4. Após executar uma operação, o componente esquece tudo sobre quem o chamou.

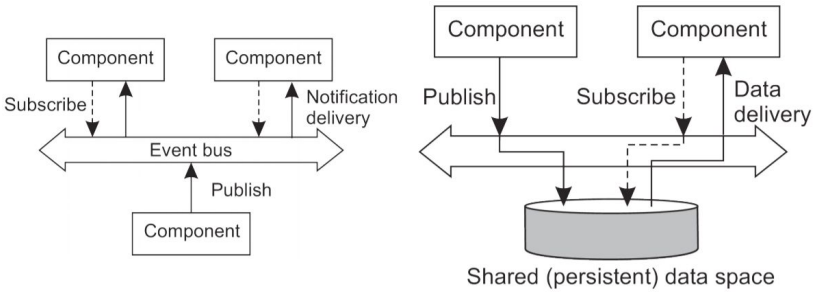
Operação	Descrição
PUT	Cria um recurso
GET	Acede ao estado de um recurso
DELETE	Destrói um recurso
POST	Altera um recurso, substituindo o estado

Arquiteturas Publish-Subscribe:

Existe uma grande separação entre processar e coordenar. Existem diferentes formas de coordenação:

	Temporally coupled	**Temporally decoupled**
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

- Processos temporariamente e referencialmente acoplados (direct): acontece coordenação direta. Por exemplo, um processo pode comunicar apenas se souber o nome/identificador do(s) processo(s) com quem pretende trocar informação. Estar acoplado temporariamente significa que os processos que se encontram a comunicar estão ambos a correr.
- Processos temporariamente desacoplados, mas referencialmente acoplados (Mailbox): não existe necessidade de comunicação que implique estarem ambos a executar ao mesmo tempo - a comunicação põe as mensagens na *mail box*.
- Processos temporariamente acoplados, mas desacoplados referencialmente (event-based): os processos não se conhecem, explicitamente. Um processo apenas pode publicar uma notificação descrevendo o evento que ocorreu.
- Processos temporariamente e referencialmente desacoplados(shared data space): os processos comunicam através de tuplos. Quando um processo insere um tuplo no *data space*, os utilizadores correspondentes são notificados. Aqui, os processos não têm referência explícita uns para os outros.

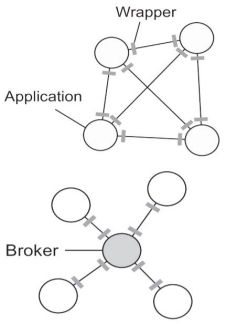


a): event-base ; b): shared data space

Organizações de middleware:

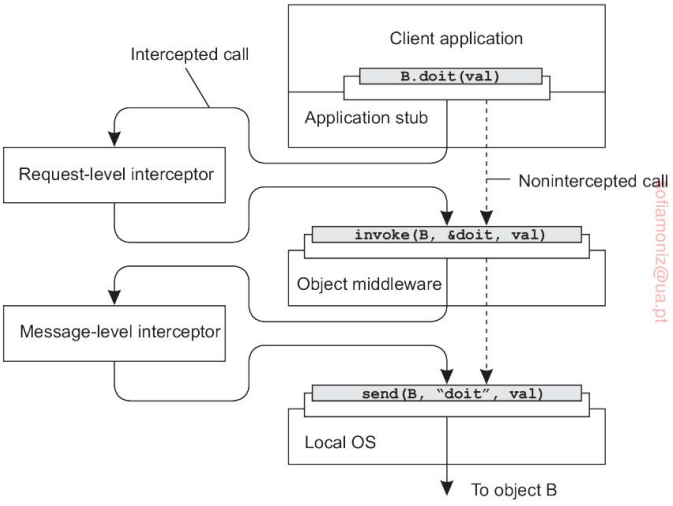
A *middleware* é independente da organização geral de um sistema distribuído ou aplicação. Assim, existem dois padrões de design, ***wrappers e interceptors***, que ajudam a *middleware* a atingir a abertura.

Wrappers/adapters: componente especial, que oferece uma interface aceitável para uma aplicação do utilizador, onde as funções são transformadas naquelas que estiverem disponíveis no componente - resolve o problema de incompatibilidade entre interfaces.



Uma forma de diminuir o número de *wrappers* (evitar explosão do número) é através da *middleware* - é implementado um ***broker***, que consiste num componente centralizado que trata dos acessos entre diferentes aplicações. O *broker*, tendo informação relativa às aplicações relevantes, contacta as aplicações apropriadas, transformando a resposta e retornando o resultado da aplicação inicial.

Interceptors: construção de software que vai quebrar o fluxo normal de controlo, e permitir que outro código seja executado. São meios fundamentais que permitem adaptar a *middleware* às necessidades específicas de uma aplicação.

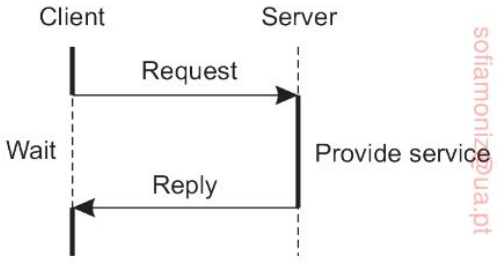


- Ao objeto “A” é oferecida uma interface local;
- A chamada de “A” é transformada num objeto genérico pela *middleware*, disponível na máquina “A”;
- Finalmente, o objeto genérico é transformado numa mensagem enviada pela rede.

Arquitetura do sistema:

Organizações centralizadas:

Arquitetura cliente-servidor: as máquinas estão divididas entre clientes e servidores.
Um servidor é um processo que implementa um serviço específico, enquanto que um cliente é um processo que solicita um serviço a um servidor, enviando um pedido e esperando pela resposta a este.



Um cliente envia um pedido a um servidor, identificando o serviço que pretende (que irá produzir um resultado que será retornado ao cliente). O servidor encontra-se à espera de um pedido. Quando o recebe, processa-o, e envia uma mensagem de resposta ao cliente.

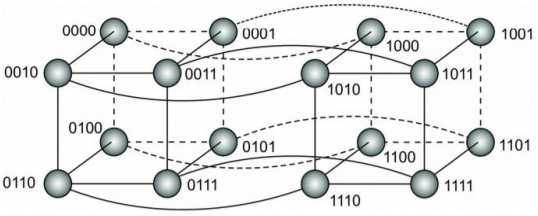
Esta arquitetura reflete a forma tradicional de modularizar *software*, onde cada módulo chama funções que estão disponíveis noutro módulo. É completamente centralizada.

Organizações descentralizadas:

Arquitetura peer-to-peer: Os processos estão organizados numa rede de sobreposição, que é uma rede lógica, onde cada processo tem uma lista local de outros *peers* com quem pode comunicar. Um nó pode não estar disponível para comunicar diretamente com um nó arbitrário, mas deve enviar as mensagens através dos canais de comunicação disponíveis.

A lista de *peers* é mais ou menos *random*, implicando que a procura de algoritmos precisa de ser implementada para localizar informação ou outros processos. De uma perspetiva de alto nível, os processos que constituem sistemas *peer-to-peer* são iguais.

Podem ser:



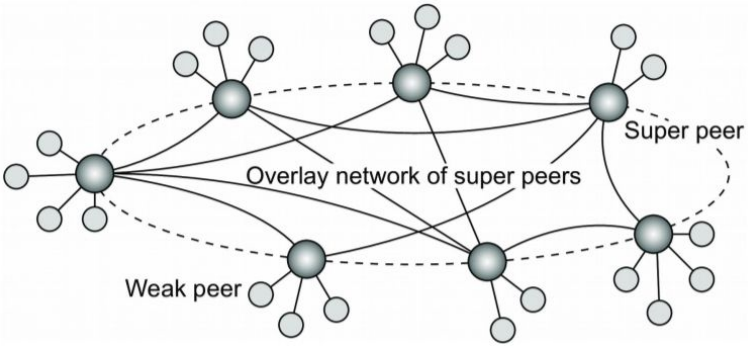
- Estruturados: os nós estão organizados num *overlay* que adere a uma topologia específica, usada para procurar informação de forma eficiente. Estes sistemas estruturados são baseados em *index* sem semântica - cada item de informação está

unicamente associado a uma chave, e essa chave está, subsequentemente, usada como um *index*.

- Não estruturados: cada nó mantém uma lista de vizinhos- isto vai resultar num grafo aleatório onde, para cada par de nós, existe uma probabilidade de esses nós formarem um vértice. No geral, um nó muda a sua lista quase que continuamente. A procura de informação pode ser de vários tipos, de entre os quais:

    - Flooding: o nó em questão passa o pedido de informação para todos os seus vizinhos.Se um vértice recebeu a informação, pode responder diretamente ao nó que pediu, ou pode enviar de volta ao expedidor original, e assim sucessivamente. Este método pode ser bastante dispendioso, tendo em conta que cada pedido tem um TTL (número de saltos) associado.

    - Random walks: o nó em questão pode simplesmente tentar encontrar a informação perguntando de forma aleatória a um vizinho - se este não contiver a informação, passa-a a um vizinho aleatório, e assim sucessivamente. Isto leva a um tráfego reduzido, mas pode demorar mais até que um nó seja atingido.

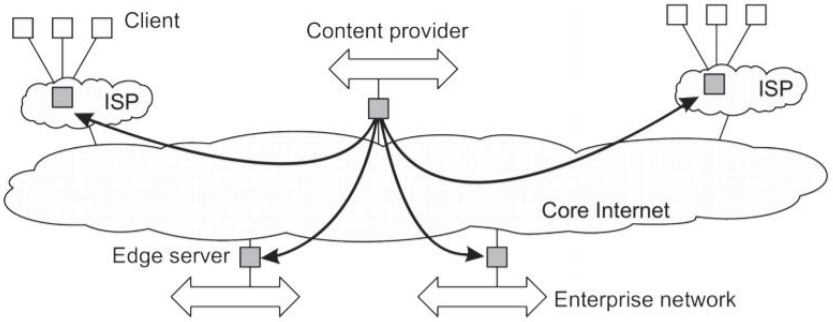


- Hierárquico: faz-se uso de nós especiais que mantêm um *index* dos itens de informação. Os nós que mantêm um *index* ou que agem como *brokers*, são designados de super peers. Os week peers serão os nós normais, que estão conectados como clientes aos *super peers*.Toda a comunicação de/para o *week peer* passa pelo *super peer* respetivo.

Organizações híbridas:

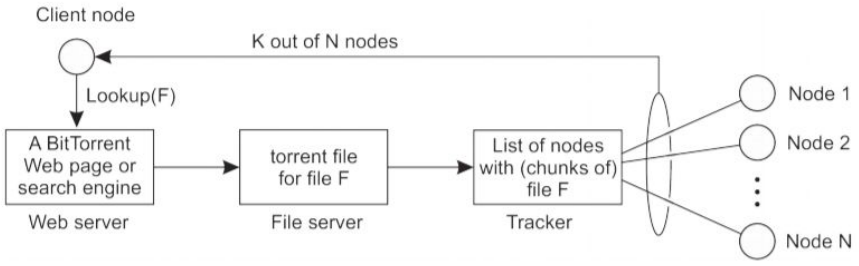
Os elementos de arquiteturas centralizadas e descentralizadas são combinados. Um componente centralizado é, normalmente, usado para lidar com os pedidos iniciais.

Sistemas edge-server: estes sistemas são implementados na internet, onde os servidores são colocados no *edge* da rede. Este *edge* é formado por um limite entre redes de empresa e a internet em si.



O propósito principal destes sistemas é fornecer conteúdo, normalmente depois de aplicar filtragens e funções transformantes. Vários servidores *edge* podem ser usados para otimizar o conteúdo e a aplicação distribuída. Um servidor *edge* age sempre como servidor original, de onde todo o conteúdo provém.

Sistemas de distribuição colaborativa: a partir do momento em que um nó se juntou ao sistema, pode usar um esquema totalmente descentralizado para colaboração.



Um exemplo bastante popular é o BitTorrent, um sistema de *download peer-to-peer*. A ideia base é quando um utilizador procura por um ficheiro, ele faz download de pedaços que podem ser unidos, formando o ficheiro em si. O objetivo de design é garantir a colaboração.

Capítulo 3

Processos:

Programa em execução num dos SO dos processadores virtuais. A partilha de CPU e outros hardwares por parte de vários processos é feita de forma transparente.

O SO contém uma tabela de processos, de forma a manter-se a par dos processadores virtuais, cada um correndo um programa diferente.

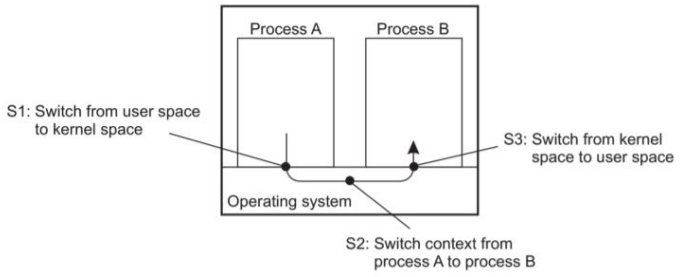
Formam a base de comunicação entre diferentes máquinas. São criados pelo SO, e podem ter múltiplas *threads*.

Threads:

Útil para que o uso do *CPU* continue mesmo quando uma operação de bloqueio do I/O é executada. Desta forma, torna-se possível construir servidores muito eficientes, que conseguem correr múltiplas *threads* em paralelo (uma *thread* tenta manter o mínimo de informação, para que isto seja possível) , sendo que várias podem bloquear, de forma a esperar que o disco *I/O* ou a rede de comunicação fique completa. Em geral, as *threads* são preferíveis ao uso de processos, quando se fala em performance.

Threads em sistemas não distribuídos:

A solução, para evitar que o sistema não bloqueie, é ter, pelo menos, duas *threads* de controlo: uma, para tratar da interação com o utilizador, e outra, para manter a *spreadsheet* atualizada. Pode ainda ser usada uma terceira, para fazer *backup* da *spreadsheet* para o disco, enquanto as outras duas *threads* fazem o seu trabalho.



Aplicações multithreading podem ser desenvolvidas como uma coleção de programas que cooperam entre si, cada um sendo executado por um processo, separadamente. Isto é implementado com mecanismos de interprocess communication (IPC), típicos de sistemas *Unix*. Como IPC necessita de intervenção da kernel (componente central do SO, que faz a ponte entre aplicações e processamento de dados a nível de hardware), um processo vai, primeiro, necessitar de trocar de modo de utilizador para modo *kernel*.

Implementação de threads:

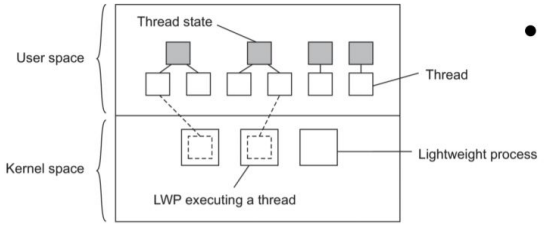
Por vezes, as *threads* são implementadas em forma de pacotes- estes contêm operações para criar e destruir *threads*, bem como operações com variáveis de sincronização, como *mutexes* e variáveis de condição.

Existem duas formas para implementar um *thread package*:

1. Através de uma biblioteca de threads, ao nível do utilizador: criar e destruir *threads* não é dispendioso, pois o custo de criar uma *thread* é determinado pelo custo de alocar memória para um *stack* de threads (destruir *threads* envolve esvaziar a *stack*).
2. Fazer com que a kernel saiba a quantidade de threads existentes e que as agende.

Lightweight processes (LWP):

Em alternativa aos dois extremos de *threading*, existe uma solução híbrida, sendo um modelo many-to-many threading. O modelo é implementado na forma de LWP- corre no contexto de um único processo, e podem existir vários *LWP* por processo. Um sistema também oferece as operações habituais para criar e destruir *threads*. Além disso, o *package* fornece facilidades para a sincronização de *threads*, como *mutexes* e variáveis de condição. Todas as operações de *threads* são feitas sem intervenção da kernel.



Cada *LWP* pode correr na sua própria *thread* (ao nível do utilizador). Atribuir uma *thread* a um *LWP* é, normalmente, implícito, e encontra-se escondido do programador.

O *LWP*, ao encontrar uma *thread* executável, troca de contexto para essa *thread*. Nesse entretanto, outros *LWPs* podem, também, estar à procura de outras *threads* executáveis. A beleza de tudo isto reside no facto de que um *LWP,* a executar a *thread*, não necessitar de ser informado - a troca de contexto é implementada, por completo, no espaço do utilizador, e aparece ao *LWP* como se de código normal se tratasse.

Quando uma *thread* faz uma chamada de bloqueio ao sistema, a execução passa de modo de utilizador para modo *kernel* mas, ainda assim, continua no contexto atual de *LWP*. No caso deste não conseguir continuar, o SO pode decidir mudar o contexto para outro *LWP*, o que implica voltar para modo de utilizador.

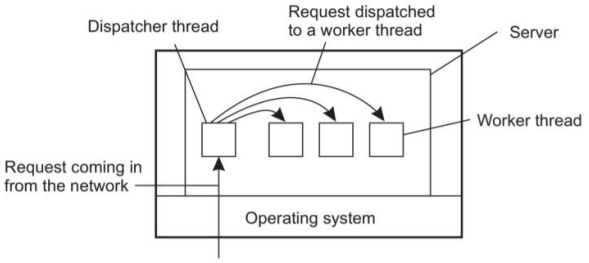
Vantagens de usar *LWP*:

1. Criar, destruir e sincronizar *threads* é pouco dispendioso e não envolve intervenção da *kernel*;
2. Uma chamada de bloqueio ao sistema não vai suspender o processo por completo;
3. Não há necessidade de as aplicações conhecerem os *LWPs*;
4. Os *LWPs* podem ser, facilmente, usados em ambientes *multithreading*.

Scheduler activations:
Quando uma *thread* bloqueia, numa chamada ao sistema, a *kernel* faz um *upcall* ao pacote da *thread*, chamando o *scheduler routine*, para selecionar a próxima *thread* executável. O mesmo processo é repetido quando uma *thread* está desbloqueada.

A vantagem, aqui, é o facto de salvar a gestão de *LWPs* por parte da *kernel*. Ainda assim, o uso de *upcalls* viola a estrutura de sistemas estruturados, onde apenas chamadas ao próximo nível mais baixo são permitidas.

Threads em sistemas distribuídos:



Além de simplificar o código, o *multithreading* torna bastante mais fácil o desenvolvimento de servidores que tiram partido do paralelismo, de forma a alcançar alto desempenho.

Na figura acima, existe uma *thread*, o *dispatcher*, que lê pedidos recebidos para uma operação com ficheiros. Os pedidos são enviados pelos clientes, para um ponto de chegada bem conhecido por este servidor. Após examinar o pedido, o servidor escolhe bloquear a *thread* que está a trabalhar (worker thread) e trata do pedido.

O *worker* avança, executando um bloqueio de leitura no sistema de ficheiros local, o que pode causar que a *thread* fique suspensa até que a informação seja retirada do disco. Se a *thread* for efetivamente suspensa, outra *thread* pode ser selecionada para ser executada.

Se se considerar o servidor de ficheiros a ser escrito na ausência de ficheiros, pode-se considerar a hipótese de operar com uma única *thread*. O *loop* principal do servidor de ficheiros recebe um pedido, examina-o, e leva-o para fora, para que seja concluído, antes que receba um próximo pedido.

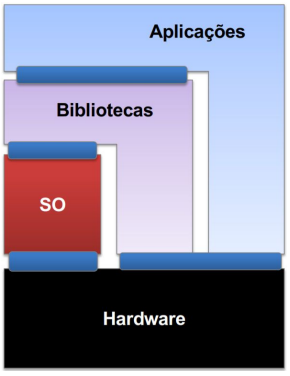
Existe ainda uma terceira alternativa - uma máquina de uma única thread, de estado finito. Quando um pedido chega, a única *thread* existente examina-o. Se este puder ser realizado com a cache de memória, assim é feito. Caso contrário, a *thread* necessita de aceder ao disco. Mas, ao invés de bloquear, a *thread* agenda uma operação de disco assíncrona, que será, mais tarde, interrompida pelo SO. De forma a que isto funcione, a *thread* vai gravar o estado do pedido, e continuar, para ver se existem outros pedidos que necessitam de ser tratados.

Em suma:

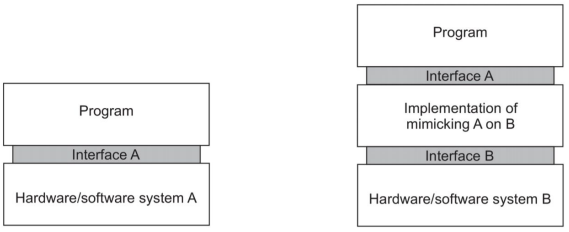
Modelo	Caracteristicas
Threads	Paralelismo, Chamadas ao sistema bloqueiam
Processo Singe-thread	Sem paralelismo, Chamadas ao sistema bloqueiam
Maquina de estados finitos	Paralelismo, Chamadas ao sistema não bloqueiam

Virtualização:

Permite que os utilizadores corram um conjunto de aplicações em cima do seu SO favorito e configura sistemas completamente distribuídos na *cloud*. O desempenho mantém-se perto das aplicações que se encontram a correr no SO atual.



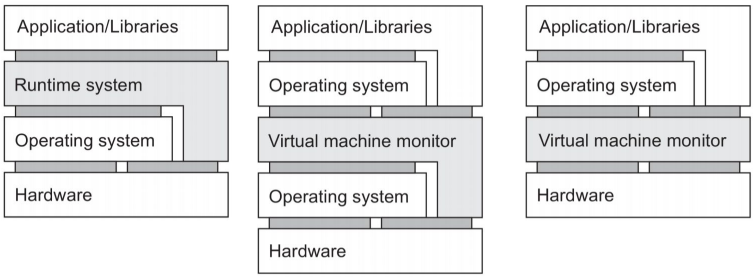
Esta separação entre ter um único CPU e o facto de ser capaz de fingir que existem mais, pode ser estendida a outros recursos, levando a resource virtualization.



- Fig.a): um sistema de computação distribuída oferece uma interface de programação para software de alto nível. Existem diferentes tipos de interfaces.
- Fig.b): a virtualização lida com estender ou trocar uma interface existente, de forma a imitar o comportamento de outro sistema.

Tipos de virtualização:

- Instruction set architecture (ISA): interface entre o *hardware* e o *software*, que forma o conjunto de instruções máquina.
- System calls: uma interface consiste em *system calls*, como oferecido pelo SO.
- Application programming interface (API): consiste em chamadas de bibliotecas. Por vezes, chamadas ao sistema estão escondidas por detrás da *API*.



- Fig.a): Máquina de processos virtuais
- Fig.b): Monitor de máquina virtual nativa
- Fig.c): Monitor de máquina virtual acolhido (hosted)

Uma abordagem alternativa à virtualização, como mostrado na fig.b), é fornecer um sistema que é implementado como uma camada que protege o hardware original, mas que também oferece um conjunto completo de instruções, como que de uma interface se tratasse.

Isto leva a um monitor de máquina virtual nativa- é chamada de nativa pois é implementada diretamente no topo do *hardware* que se encontra por baixo. A interface fornecida por este monitor pode ser oferecida, simultaneamente, a diferentes programas. Como resultado, é possível ter múltiplos, e diferentes, sistemas operativos “convidados”, que correm independentemente, e concorrentemente, na mesma plataforma.

Tudo isto implica ter dispositivos de drivers para os recursos. De forma a evitar todo este esforço, um monitor de máquina virtual acolhido vai correr no topo do sistema operativo acolhido, confiável, como mostrado na fig.c). Neste caso, o monitor pode fazer uso das facilidades fornecidas pelo SO acolhido.

Clientes:

Os processos clientes, geralmente, implementam interfaces do utilizador, que podem variar entre *displays* bastante simples a interfaces avançadas. O *software* do cliente procura atingir a transparência, escondendo os detalhes relativos à comunicação com servidores - onde estes estão localizados atualmente e se estão replicados.

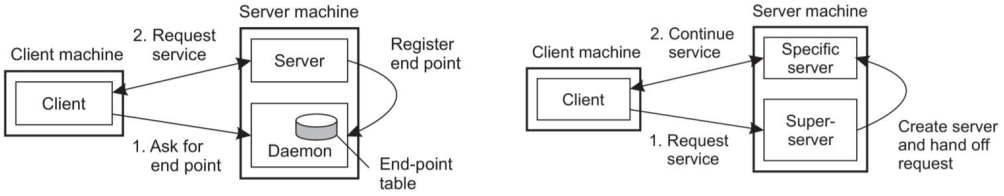
A maior tarefa das máquinas clientes é fornecer meios para os utilizadores interajam com servidores remotos. Existem duas maneiras desta interação ser suportada:

1. Para cada serviço remoto, a máquina cliente vai ter um correspondente, separado, que lhe permite contactar o serviço respetivo na rede.
2. Fornecer acesso direto a serviços remotos, oferecendo uma única interface de cliente, estável. Isto significa que a máquina cliente apenas pode ser usada enquanto terminal, sem haver necessidade de armazenamento local. Tudo é processado e armazenado no servidor.

Servidores:

Processo que implementa um serviço específico em prol de uma coleção de clientes. Um servidor espera por um pedido de um cliente e, subsequentemente, assegura-se de que o pedido é tratado e, seguidamente, espera pelo próximo pedido.

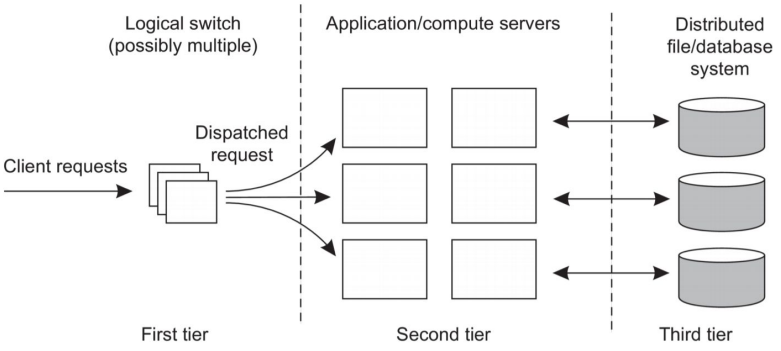
Estão apenas sujeitos a alguns problemas de design. Por exemplo, os servidores podem ser iterativos (trata do pedido por si e, se necessário, envia uma resposta ao cliente) ou concorrentes (passa o pedido a outro processo ou thread), implementar um ou mais serviços, e podem ser sem estado, ou dinâmicos. Outra questão passa por endereçar os serviços e mecanismos que interrompem um servidor após um pedido de serviço ter sido realizado e, possivelmente, já está a ser processado.



- Fig.a): Por vezes, o cliente necessita de procurar um *end point*. Uma solução passa por ter um daemon especial, a correr em cada máquina que corre servidores. O *daemon* mantém-se a par do *end point* atual, que cada serviço implementa, através de um servidor co-alocado. O *daemon* vai "ouvir" o pedido, num *end point* bem conhecido. O primeiro cliente a contactar o *daemon*, faz um pedido pelo *end point* e, de seguida, contacta o servidor específico.
- Fig.b): Em vez de rastrear todos os processos, poderá ser mais eficiente ter um único superserver a "ouvir" cada *end point*, que estará associado a um pedido específico.

Clusters de servidores:

Coleção de máquinas ligadas através de uma rede, onde cada máquina corre um ou mais servidores. Os *clusters* aqui considerados são aqueles em que as máquinas estão conectadas através de uma rede local, muitas vezes oferecendo uma grande largura de banda e baixa latência.



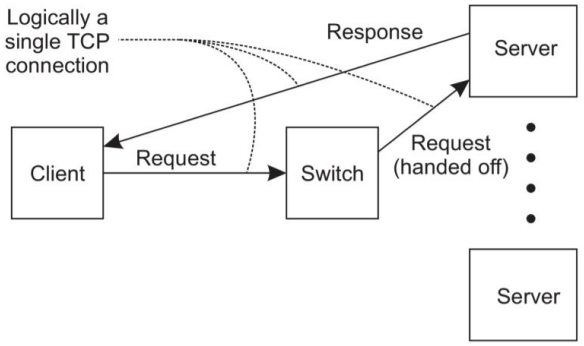
A imagem encontra-se dividida em três patamares.

1. O primeiro patamar consiste num *switch* lógico através do qual os pedidos do cliente são encaminhados.
2. Tal como em várias arquiteturas multi nível cliente-servidor, muitos *clusters* de servidores também contêm servidores dedicados ao processamento da aplicação. Normalmente, são servidores que executam com elevado desempenho de hardware, dedicados a entregar potência computacional.
3. O último patamar consiste em servidores que processam informação, tal como ficheiros e servidores de database. Estes servidores podem estar a correr em máquinas especializadas, configuradas para grande velocidade aquando do acesso ao disco, e têm uma grande cache do lado do servidor.

Quando um *cluster* de servidor oferece múltiplos serviços, pode-se dar o caso de diferentes máquinas correrem diferentes servidores de aplicação. Assim, o *switch* vai ter que conseguir distinguir serviços ou, de outra forma, não conseguirá encaminhar pedidos para as máquinas corretas.

O objetivo desta organização em *clusters* é esconder o “interior” do *cluster* do “mundo lá fora”- a organização de um *cluster* deve estar protegida das aplicações. Para que isto aconteça, a maioria dos *clusters* usam um único ponto de entrada que passa as mensagens do servidor para o *cluster*. O desafio é substituir, de forma transparente, esta única entrada por uma solução completamente distribuída.

Assim, uma forma standard de aceder a um *cluster* de servidor é montar uma ligação TCP, onde cada pedido ao nível da aplicação é enviado como parte da sessão. Uma sessão termina quando a ligação é deitada abaixo. Desta forma, o cliente constrói a ligação TCP, de forma a que todos os pedidos e respostas passem pelo *switch*. O *switch*, em resposta, vai construir a ligação TCP com um servidor selecionado, e passa os pedidos do cliente àquele servidor, sendo que também aceita respostas do servidor (que vai passar ao cliente). Efetivamente, o *switch* localiza-se no centro de uma ligação TCP entre o cliente e um servidor selecionado. Esta aproximação é uma forma de NAT.



Em alternativa, o *switch* pode passar a ligação para um servidor selecionado, de forma a que todas as respostas sejam diretamente comunicadas ao cliente, sem passarem pelo server - TCP *handoff* (tal como na figura acima).

Aqui, quando o *switch* recebe um pedido para ligação TCP, primeiramente, identifica o servidor mais indicado para tratar daquele pedido e, seguidamente, envia o pacote de pedido para esse servidor, mas inserindo o IP do *switch* como campo fonte no cabeçalho do pacote IP, transportando o segmento TCP. É de notar que a reescrita deste endereço é necessária, de forma a que o cliente continue a executar o protocolo TCP - espera resposta de volta do *switch*, e não de um servidor arbitrário de quem nunca ouviu falar.

O *TCP handoff* é especialmente eficiente quando as respostas são muito maiores que os pedidos, como é o caso de servidores de web.

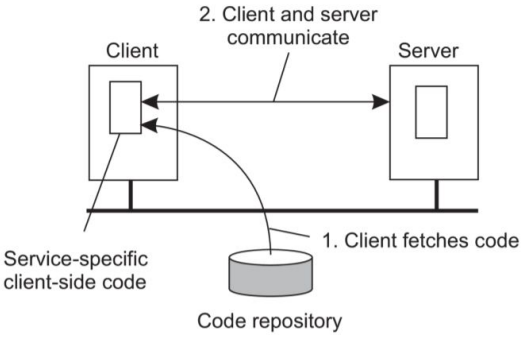
Já pode ser visto que o *switch* tem um papel bastante importante na distribuição da "carga" pelos diferentes servidores. Decidindo para onde encaminhar um pedido, o *switch* está também a decidir qual servidor vai tratar de outros processos de pedidos que ainda estão para acontecer.

No entanto, manter este estado de tratar de segmentos de TCP subsequentes, que pertencem à mesma ligação TCP, poderá fazer com que o *switch* fique mais lento.

Migração de código:

Permitir migração de código leva a aumento de performance e flexibilidade (partir a aplicação em diferentes partes, decidindo, de primeira mão, qual parte deve ser executada). Quando uma comunicação é muito dispendiosa, pode-se reduzir a comunicação através do envio de operações do servidor para o cliente, deixando este último realizar o máximo de processamento local possível.

Se o código puder ser movido entre máquinas diferentes, torna-se possível configurar os sistemas distribuídos de forma dinâmica.



Uma alternativa é permitir que o servidor forneça a implementação do cliente apenas quando é estritamente necessário, ou seja, quando o cliente faz *bind* do servidor. Nessa altura, o cliente faz download dinâmico da implementação, passa por os processos necessários para a inicialização e, subsequentemente, invoca o servidor.

Este modelo, de dinamicamente mover código de sites remotos, requer que o protocolo, para fazer download e inicializar código, se encontre standarizado. Também é necessário que o código de que se fez download consiga correr na máquina do cliente.

A vantagem do modelo de fazer download dinâmico do software do lado do cliente é que os clientes não necessitam de ter software pré instalado para comunicar com os servidores. Em vez disso, o software pode ser movido quanto necessário, como também descartado quando não é mais preciso.

Outra vantagem é que, ao serem interfaces standarizadas, pode-se mudar o protocolo cliente-servidor e implementação tantas vezes quanto quisermos.

Capítulo 4

Protocolos por camadas:

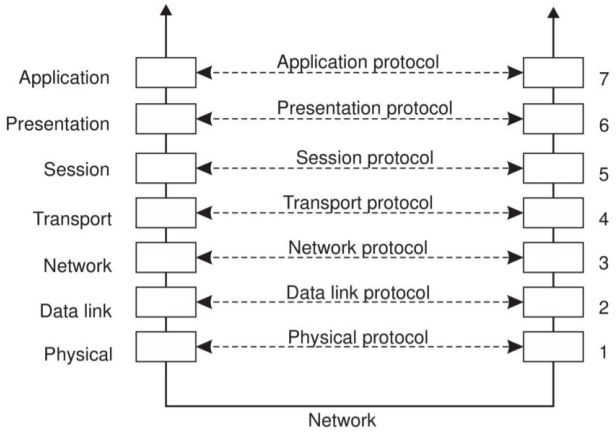
Devido à falta de memória partilhada, toda a comunicação, em sistemas distribuídos, é baseada em enviar e receber mensagens de baixo nível. Quando um processo quer comunicar com outro, começa por construir a mensagem no seu próprio espaço de endereçamento. Seguidamente, faz uma chamada ao sistema, que faz com que o SO envie a mensagem para o processo pretendente. Tanto o processo que envia, como aquele que recebe, devem concordar com a quantidade de bits enviada.

Pilha OSI:
Identifica os vários níveis envolvidos, dá-lhes nomes standard e aponta qual nível deve fazer o quê.

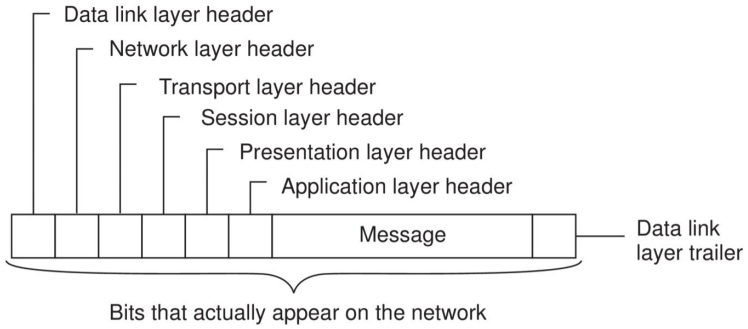
Este modelo foi designado de forma a permitir que sistemas abertos comuniquem. Um sistema aberto é aquele que está preparado para comunicar com qualquer outro sistema aberto, usando regras standard que ditam o formato, conteúdo, e significado das mensagens enviadas e recebidas.

Estas regras são formalizadas em protocolos de comunicação. De forma a permitir que um grupo de computadores comunique, na rede, devem todos concordar no protocolo a usar. Um protocolo é, por sua vez, um serviço de comunicação. Existem dois tipos:

1. Conexão orientada: antes de enviar informação, o emissor e recetor explicitamente estabelecem uma ligação e, possivelmente, negociam parâmetros específicos do protocolo a usar. Quando terminam, acabam com a ligação.
2. Sem conexão: não é preciso nenhum setup, de antemão. O emissor transmite a primeira mensagem apenas quando se encontra pronto.



No modelo OSI, a comunicação está dividida por 7 camadas, sendo que cada uma oferece um ou mais serviços de comunicação à camada que se encontra acima. Ou seja, cada camada fornece uma interface (conjunto de operações que, em conjunto, definem o serviço que a camada está preparada para oferecer) à camada acima.



Quando um processo quer comunicar com outro, constrói uma mensagem e passa-a à camada de aplicação. Esta camada vai adicionar um cabeçalho à mensagem, passando a mensagem resultante através das restantes camadas.
Na camada de apresentação, é adicionado um cabeçalho próprio, sendo a mensagem resultante passada à camada de sessão, e assim adiante. Será a camada física que vai, efetivamente, transmitir a mensagem.

A mensagem, ao chegar à máquina que hospeda o processo recetor, é passada em direção ascendente, sendo que cada camada examina o seu próprio cabeçalho. Finalmente, quando a mensagem chega ao recetor, este poderá responder, usando o caminho inverso.

Camadas de baixo nível

1. Camada física: apenas envia bits (codifica-os).
2. Camada de ligação: codificação de uma série de bits num frame com controlo de erros e fluxo - calcula o *checksum* e verifica se está correto no emissor e recetor.
3. Camada de Rede: como pacotes são roteados numa rede de computadores.

Camada de transporte

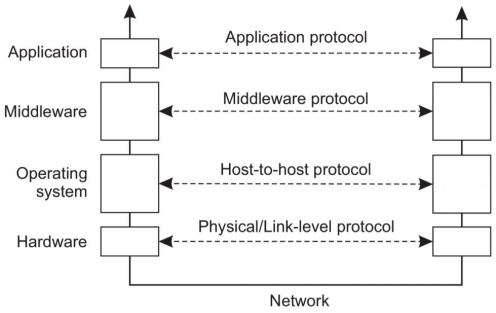
A camada de aplicação deve conseguir enviar uma mensagem para a camada de transporte com a expectativa de que vai ser enviada, sem perdas. Existem dois protocolos principais:

1. TCP: protocolo de transporte da internet.
2. UDP: protocolo de transporte sem ligação. É um IP com acréscimos mínimos.

Camada middleware

A middleware é uma aplicação que vive, logicamente, na camada de aplicação OSI, e que contém vários protocolos gerais que garantem, às suas próprias camadas, aplicações mais específicas.

O DNS é usado para procurar por um endereço de rede que tem um nome associado - este nome é o domain name.



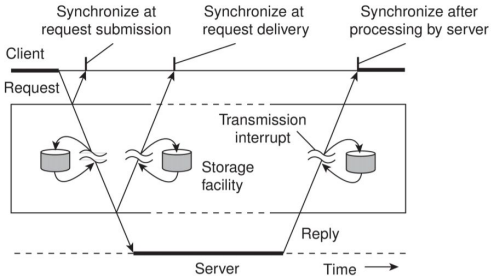
Esta aproximação leva a um esquema de camadas adaptado, onde as camadas de sessão e apresentação foram substituídas por uma única camada de *middleware*, que contém protocolos de aplicação independentes, que não pertencem a nenhuma camada.

Tipos de comunicação:

Persistente ou transiente:

- Persistente: uma mensagem que foi submetida para transmissão é guardada, pela comunicação de *middleware*, enquanto não é entregue, desde o emissor, até ao recetor.
- Transiente: uma mensagem é guardada pelo sistema de comunicação, apenas enquanto as aplicações de envio e receção estão a executar. Como se vê na figura acima, se a *middleware* não conseguir enviar a mensagem devido a uma interrupção na transmissão, ou porque o recetor não estava ativo, a mensagem vai ser descartada.

Assíncrona ou síncrona:



- Assíncrona: o emissor continua, imediatamente após ter submetido uma mensagem para transmissão. Isto significa que a mensagem é, imediatamente e temporariamente, armazenada pela *middleware* após transmissão.
- Síncrona: o emissor fica bloqueado até que o pedido seja conhecido, de forma a ser aceite. Existem 3 pontos onde a sincronização pode ocorrer:
    a. O emissor pode ficar bloqueado até que a *middleware* notifique que ficará responsável pela transmissão do pedido;
    b. O emissor pode sincronizar, até que o pedido seja entregue ao recetor esperado;
    c. O emissor espera até que o pedido tenha sido completamente processado, ou seja, até que o recetor envie uma resposta em retorno.

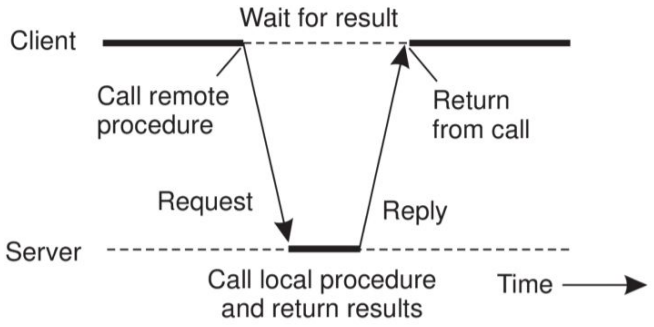
Exemplo: o modelo cliente/servidor é baseado no modelo de comunicações transiente síncrono.

Remote procedure call:

Quando um processo, numa máquina “A”, chama por um processo da máquina “B”, o processo chamador “A” é suspenso, e a execução do processo chamado ocorre em “B”. A informação pode ser transportada do *caller* para o *callee* nos parâmetros, e pode regressar no resultado do processo. Nada disto é visível ao programador.

O propósito é, então, permitir que programas chamem processos localizados noutras máquinas.

No entanto, existem alguns problemas. Devido ao facto dos processos que chamam/são chamados correrem em máquinas diferentes, eles executam em diferentes espaços de endereços, o que pode levar a que ambas as máquinas entrem em colapso, e cada falha pode levar a diferentes problemas.



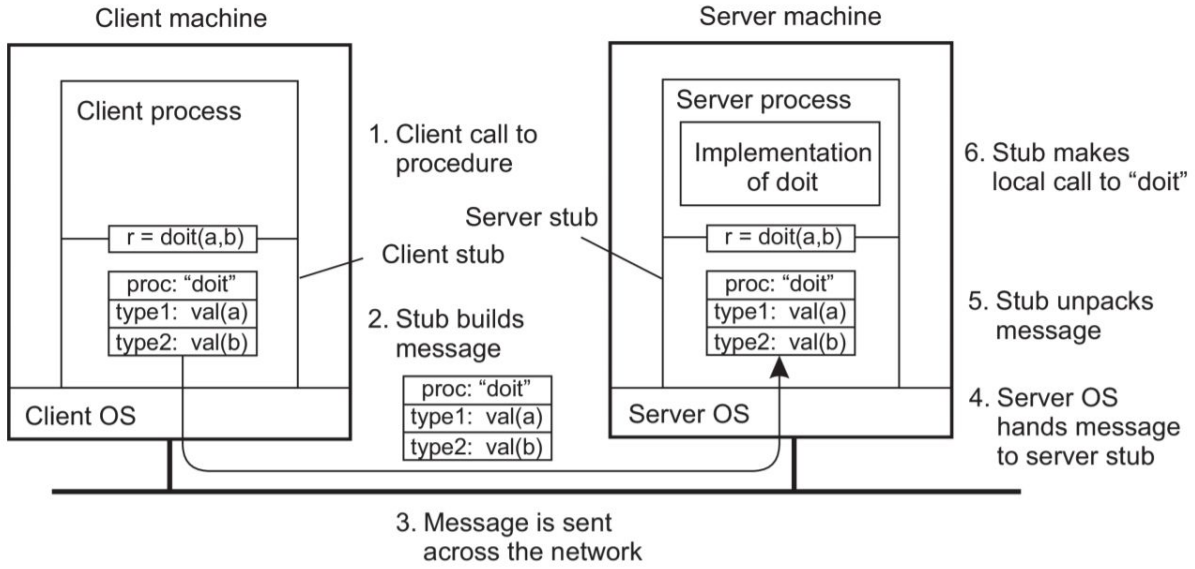
Uma RPC deve ser transparente - nenhum processo deve saber que o outro está a executar numa máquina diferente. A transparência é obtida de forma análoga - quando a operação *append* (adicionar informação a uma lista armazenada, tendo dois parâmetros - *append(data, dbList)*) é, realmente, um processo remoto, é oferecida uma diferente versão ao cliente - client stub. Este último, em vez de fazer uma operação de append, embala os parâmetros numa mensagem (wrap) e envia.

Quando uma mensagem chega ao servidor, o SO deste passa-a para o server stub - o lado do servidor equivalente ao *client stub*, ou seja, é uma parte de código que transforma pedidos vindos da rede em *procedure calls* locais. Para o servidor, é como se estivesse a ser chamado diretamente pelo cliente - faz o seu trabalho e, de seguida, envia o resultado para o *caller* (que, neste caso, será o server stub) normalmente.

Quando o *server stub* volta ao controlo, depois da chamada terminar, embala o resultado numa mensagem e chama *“send”* para retornar a mensagem para o cliente. Após isto, faz uma chamada a *“receive”* novamente, de forma a esperar pelo próximo pedido.

Quando o resultado da mensagem chega à máquina do cliente, o SO passa-a para a operação *“receive”* (que já havia sido chamada pelo client stub) e, consequentemente,o processo cliente é desbloqueado. O cliente trata da mensagem, e retorna-a, da forma habitual. Quando o *caller* toma controlo, seguindo a mensagem a que se faz *append*, tudo o

que sabe é que fez *append* de alguma informação para uma lista. Não faz ideia do controlo remoto que aconteceu na outra máquina.



Esta ignorância por parte do cliente é onde reside a beleza de todas estas operações.

RPC: passagem de parâmetros:

A função do *client stub* é, então, pegar nos seus parâmetros, embalá-los numa mensagem, e enviá-los para o *server stub*. Embalar parâmetros numa mensagem é chamado de parametre marshaling.

Na operação de *append*, deve ser assegurado que ambos os parâmetros são enviados pela rede, e corretamente interpretados pelo servidor. A questão é que, no fim, o servidor só vê uma série de *bytes*, que vêm da constituição original da mensagem, enviada pelo cliente. No entanto, não existe informação adicional sobre o que significam aqueles *bytes*, que normalmente acompanham a mensagem. Sendo assim, como vai a informação ser reconhecida pelo servidor? Além disto, é também preciso tratar de um outro problema - o alojamento de *bytes* na memória pode diferir entre as diferentes arquiteturas das máquinas.

A solução é transformar a informação a ser enviada para uma máquina num formato independente da rede, assegurando-se também que ambos os lados da comunicação esperam pelo mesmo tipo de mensagem, na transmissão.

Marshaling e *unmarshaling* trata-se de transformação para formatos neutros e forma uma parte essencial de RPCs.

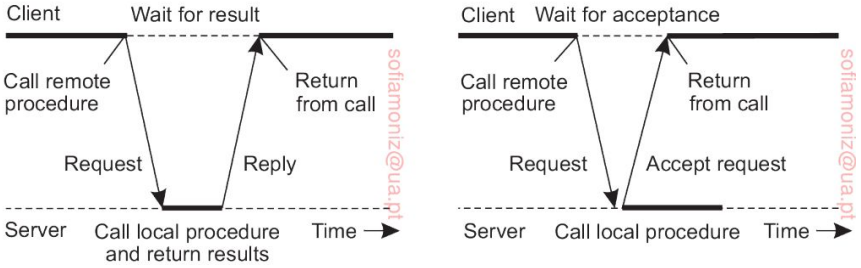
Chega-se assim a um problema: como são os ponteiros passados? A grande questão aqui é o facto de ponteiros só terem sentido se forem tratados localmente - eles referem-se a localizações de memória que só têm significado no processo de chamada.

Podemos, então, resolver o problema usando referências globais, que são significantes para o processo chamador e para o chamado. Um ponto importante é que ambos têm que

saber, exatamente, o que fazer quando uma referência global é passada. O processo chamado e o chamador devem ter exatamente a mesma noção das operações que podem ser realizadas. Além disso, ambos os processos devem também concordar com o que fazer quando um *file handle* é passado.

RPCs assíncronos:

Usados para evitar situações em que não há qualquer resultado a ser retornado ao cliente. Aqui, o servidor envia uma resposta imediata ao cliente, no momento em que o pedido de RPC é recebido, após o qual chama localmente o processo pedido. A resposta funciona como um aviso ao cliente, de que o servidor vai processar o RPC. O cliente, após receber esse aviso, vai continuar sem bloquear.

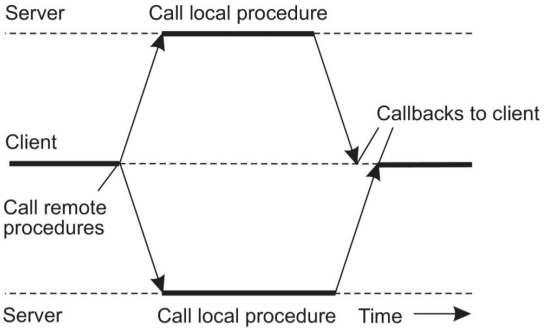


- Fig.a): Interação entre cliente e servidor num RPC tradicional
- Fig.b): Interação usando um RPC assíncrono

RPCs assíncronos são também úteis quando uma resposta vai ser retornada, mas o cliente não está preparado para esperar por esta, e não pode fazer nada nesse entretanto.

RPCs de multicast:

É mais uma alternativa aos RPC síncronos. Aqui, executam-se vários RPCs ao mesmo tempo - um RPC é enviado para um grupo de servidores.

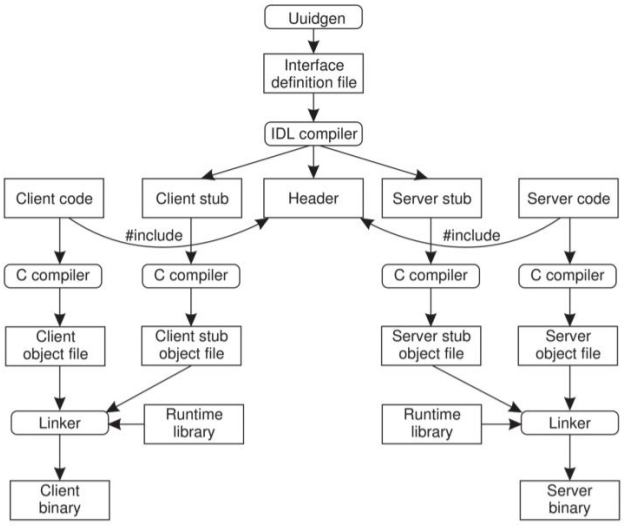


Neste exemplo, o cliente envia um pedido para dois servidores e estes processam o pedido, independentemente e em paralelo.

Existem dois problemas- o servidor pode não saber que o RPC vai ser enviado para mais que um servidor e é necessário considerar o que fazer com as respostas - o cliente vai processar todas as respostas, ou esperar por apenas uma?

RPC na prática - escrever um cliente e um servidor:

DCE: sistema de *middleware* nato, designado para executar enquanto existe uma camada de abstração entre o SO e aplicações distribuídas. A ideia é fazer com que o cliente possa ter uma coleção das máquinas existentes, adicionar o *software* de *DCE* e, seguidamente, ficar capaz de correr aplicações distribuídas, sem afetar as aplicações existentes. Os processos do utilizador agem como clientes, de forma a aceder a serviços remotos disponibilizados por processos servidores. Toda a comunicação entre clientes e servidores funciona através de RPCs.



Existe um IDL (interface definition language) file, que contém a informação necessária para fazer *marshal* dos parâmetros e, de forma correta, fazer *unmarshal*.

O cliente envia um identificador único, da interface em causa, na primeira mensagem RPC, e o servidor verifica se está correto. Assim, se o cliente tentar contactar um servidor errado, ou uma versão desatualizada do servidor correto, o servidor deteta, e o *binding* já não acontece.

O ficheiro IDL deve ser editado, de forma a ter os nomes dos procedimentos remotos e os seus parâmetros. Quando o ficheiro IDL está completo, o compilador IDL é chamado. O output desta compilação contém:

- Cabeçalho de ficheiro: contém o identificador único, que deve ser incluído tanto no código do cliente, como no do servidor.
- Client stub: contém os procedimentos responsáveis por coletar e embalar os parâmetros na mensagem que vai sair e, de seguida, chamar o sistema para a correr.
- Server stub: contém as *procedures calls* feitas pelo sistema, na máquina do servidor, quando chega uma mensagem.

O próximo passo será feito pela aplicação de escrita- escrever o código do cliente e do servidor.

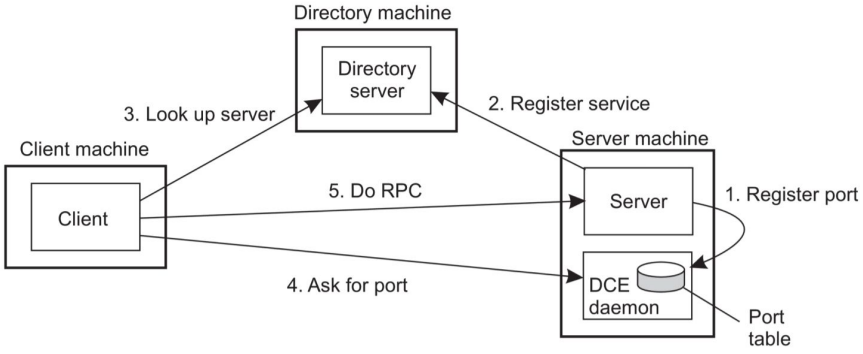
Ligação cliente-servidor:

De forma a permitir que um cliente chame um servidor, é necessário que o servidor tenha sido registado e preparado para receber chamadas. O registo de um servidor deve tornar possível, para um cliente, localizar o servidor e fazer *bind*.

Encontrar a localização de um servidor é feito em dois passos:

1. Localizar a máquina do servidor;
2. Localizar o servidor (o processo correto) na máquina.

O segundo passo é subtil. Comunica-se com o servidor, sendo que o cliente precisa de saber qual a porta do servidor por onde pode enviar mensagens. No DCE, é mantida uma tabela de (servidor,porta) em cada servidor - DCE daemon. Antes que a receção de pedidos fique disponível, o servidor deve perguntar ao SO por uma porta, que será registada no *DCE daemon*.

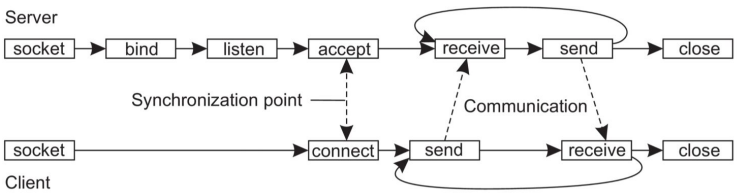


O servidor regista, também, o diretório, fornecendo-o ao endereço de rede da máquina do servidor, e o nome pelo qual o servidor pode ser procurado.

Sockets:

Ponto de extremidade de comunicação onde uma aplicação pode escrever informação que será enviada através da rede inerente, e para o qual a informação que chega pode ser lida. Uma *socket* forma uma abstração sobre a porta atual, que é usada pelo SO local para um protocolo de transporte específico.

Operation	Description
`socket`	Create a new communication end point
`bind`	Attach a local address to a socket
`listen`	Tell operating system what the maximum number of pending connection requests should be
`accept`	Block caller until a connection request arrives
`connect`	Actively attempt to establish a connection
`send`	Send some data over the connection
`receive`	Receive some data over the connection
`close`	Release the connection



Sockets com ZeroMQ:

Primeiramente, vai estabelecer-se uma ligação entre o emissor e o recetor, antes que a transmissão de mensagens possa começar. Uma *socket* pode estar vinculada a múltiplos endereços, permitindo que um servidor trate de mensagens de diferentes fontes, através de uma única interface.

Estas *sockets* suportam comunicação many-to-one, bem como one-to-many (*multicasting*). O essencial é que, aqui, a comunicação é assíncrona - o emissor vai continuar normalmente, depois de ter submetido uma mensagem ao subsistema de comunicação inerente.

O *ZeroMQ* estabelece um alto nível de abstração - acopla *sockets*. Uma *socket* de um tipo específico, usada para enviar mensagens, vai-se acoplar com a *socket* correspondente, que recebe o pedido. Cada par de *sockets* corresponde, então, a um padrão de comunicação.

Existem 3 padrões:

- Request-reply: o cliente usa uma *request socket*, esperando uma resposta apropriada. Espera-se, assim, que o servidor use uma *reply socket*. Este padrão simplifica problemas, pois evita a necessidade de chamar a operação *listen*, bem como *accept*.
- Publish-subscribe: os clientes subscrevem mensagens específicas que são publicadas pelos servidores - apenas as mensagens subscritas vão ser transmitidas. Este padrão permite mensagens *multicasting*, do servidor para vários clientes.
- Pipeline: um processo quer expulsar os seus resultados, assumindo que existem processos que os querem receber. A essência é que um processo de expulsão não se importa se outros processos recebem os resultados - o primeiro que ficar disponível irá fazê-lo, facilmente.

Queues:

Comunicação assíncrona persistente através de um *middleware* de filas (*queues*). As filas são *buffers* nos servidores de comunicação.

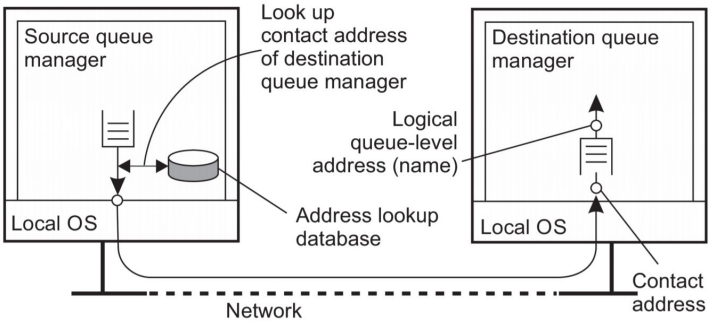
Várias aplicações podem partilhar uma única fila. A única garantia que o emissor tem é que a sua mensagem vai, eventualmente, ser inserida na fila do recetor. Não tem qualquer garantia quando, ou se, a mensagem vai, sequer, ser lida. Não há necessidade de o recetor estar a executar quando o emissor está a colocar a mensagem na fila, bem como o emissor não precisa de executar quando o recetor retira a mensagem da mesma.
O único aspeto importante, da perspectiva da *middleware*, é que a mensagem seja corretamente endereçada.

Operation	Description
`put`	Append a message to a specified queue
`get`	Block until the specified queue is nonempty, and remove the first message
`poll`	Check a specified queue for messages, and remove the first. Never block
`notify`	Install a handler to be called when a message is put into the specified queue

Modelo geral de uma queue:

- Gerenciada por um queue manager- processo separado ou implementado por uma biblioteca que está ligada à aplicação em questão.
- Uma aplicação apenas põe mensagens em *queues* locais. Desta forma, obter uma mensagem faz-se extraindo-a apenas da *queue* local.
- Se todos os *queue manager* estiverem ligados às suas respetivas aplicações, já não se poderá falar de um sistema de mensagens assíncrono persistente.

Se as aplicações apenas podem colocar mensagens em *queues* locais, então a mensagem terá, claramente, que transportar informação relativa ao destino da mesma. O *queue manager* deve assegurar-se que a mensagem chega ao destino devido.

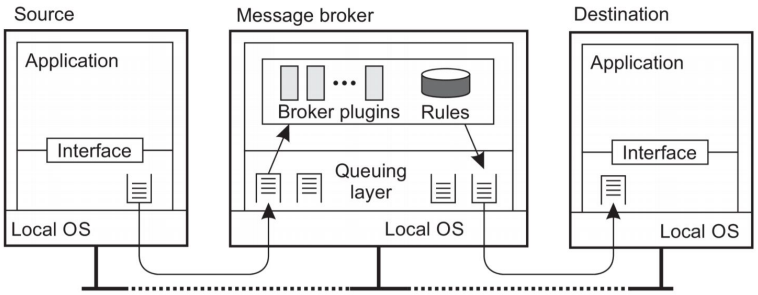


De forma a aumentar a transparência, é preferível que as *queues* tenham nomes lógicos e independentes da sua localização. Supondo que uma *queue* é implementada por um processo separado, usar nomes lógicos implica que cada nome irá ter um endereço de contacto associado, como um par *(host,port)*, sendo que o mapeamento nome-para-endereço fica facilmente disponível para um *queue manager*, como na imagem acima.

Outra questão é o facto de ser necessário considerar como é que o mapeamento nome-para-endereço fica disponível para um *queue manager*. Uma solução simples passa por implementar esse mapeamento como uma tabela de pesquisa, e copiar essa tabela para todos os *managers*. Isto leva a problemas de manutenção visto que, ao editar uma *queue*, todas as tabelas vão alterar.

Message broker:

Nós especiais, numa rede com sistema de *queue*, que tratam da comunicação. Agem como um *gateway*, ao nível da aplicação. O seu propósito principal é converter mensagens que chegam, de forma a que possam ser entendidas pela aplicação recetora.



É de notar que num sistema baseado em mensagens em *queues*, um *message broker* é apenas outra aplicação, como mostrado na figura acima. Não é, então, considerado uma parte integral de um sistema de *queues*.

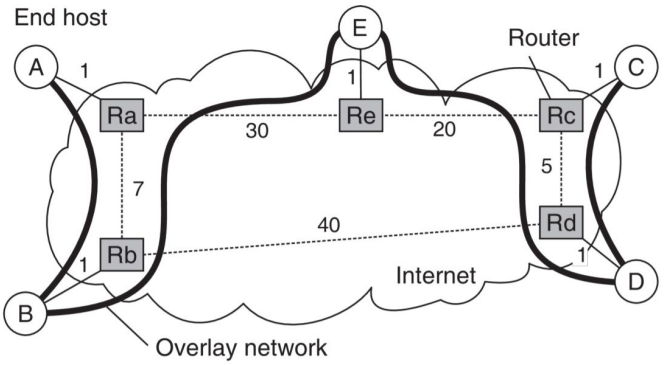
Um broker pode ter várias funcionalidades:

- Usado para converter mensagens para o formato esperado pelo destinatário;
- Pode funcionar como um *gateway* para aplicações, onde a informação do protocolo de mensagem de várias aplicações é codificado;
- Pode ter funcionalidades de roteamento, como no caso do modelo *publish-subscribe*, onde as aplicações enviam mensagens na forma de *publishing*, tendo o *broker* a responsabilidade de combinar aplicações, baseando-se nas mensagens que são trocadas.

Multicasting ao nível da aplicação:

Os nós estão organizados numa rede de sobreposição (*overlay*), que é usada para disseminar informação para os seus membros. É de notar que os *routers* de rede não estão aqui envolvidos e, como consequência, a comunicação entre nós nesta rede de sobreposição pode ter que atravessar vários elos físicos - rotear mensagens nesta overlay pode não ser o ideal, quando em comparação ao que se pode atingir com *routing* ao nível da rede.

Os nós devem organizar-se em árvore e, assim, irá existir um único caminho (*overlay*) entre cada par de nós. Uma alternativa é fazer com que os nós se organizem numa rede “de arame”, onde cada nó vai ter múltiplos vizinhos e, em geral, existem vários caminhos entre cada par de nós. No entanto, nesta segunda opção, se existir uma quebra de alguma ligação, ocorre uma oportunidade para disseminar informação, sem que seja necessário reorganizar toda a rede, imediatamente.



No entanto, existem alguns custos:

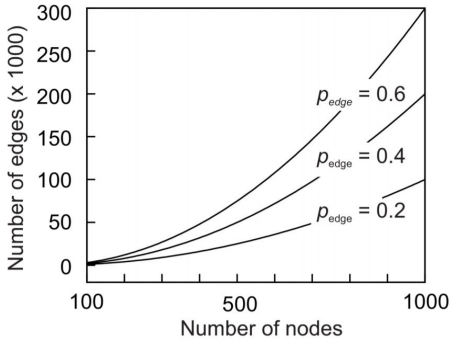
- Link stress: conta o número de vezes que um pacote passa pelo mesmo *link*. Se for superior a 1 significa que, apesar do pacote poder ser encaminhado por duas direções diferentes, parte dessas conexões pode corresponder ao mesmo *link*.
- Stretch: mede o rácio, em atrasos, entre dois nós da *overlay*, e também o atraso que os dois nós experienciam na rede inerente.

Flooding baseado em multicasting:

No geral, *multicasting* refere-se ao enviar de uma mensagem a um grupo específico de nós. Quando falamos em *multicasting*, pretende-se minimizar o número de nós intermediários, para onde não se pretende enviar a mensagem.

Para evitar essa ineficiência, deve ser construído uma rede *overlay* por cada grupo de *multicasting*.

Se supusermos que cada *overlay* corresponde a um grupo de *multicasting* e que, assim, precisaremos de fazer *broadcast* da mensagem, pode ser utilizado *flooding* - cada nó encaminha uma mensagem para cada nó, exceto para aquele que a enviou originalmente.



A ideia é simples: quando um nó faz *flooding* de uma mensagem, precisa de a encaminhar para um nó específico, sendo que o irá fazer com probabilidade *Pflood*. Ainda assim, existe um risco inerente- quanto mais baixo for *Pflood*, maior é a chance de nem todos os nós da rede serem alcançados.

Capítulo 5

Nomes, identificadores e endereços:

Um nome é uma string de bits ou caracteres usado para referir uma entidade, sendo esta última praticamente qualquer coisa, num sistema distribuído.

Para se poder operar sobre uma entidade, é necessário ter acesso à mesma, e tal será feito através de um ponto de acesso. O nome de um ponto de acesso é, por sua vez, um endereço. Uma entidade pode oferecer mais que um ponto de acesso.

Uma entidade pode, facilmente, mudar o seu ponto de acesso, ou um ponto de acesso pode ser atribuído a outra entidade. Se um endereço for usado para se referir a uma entidade, irá existir uma referência inválida quando tal acontece. É, então, preferível reconhecer um serviço por um nome independente do endereço do servidor a que está associado.

Da mesma forma, se uma entidade oferecer mais que um ponto de acesso, não é claro qual endereço deve ser usado como referência.

Assim, surgem os identificadores, que têm 3 principais propriedades:

1. Um identificador refere-se, no máximo, a uma entidade;
2. Cada entidade é, no máximo, referida por um identificador;
3. Um identificador refere-se sempre à mesma entidade (nunca é reusado).

Ao serem usados identificadores, torna-se mais fácil evitar a ambiguidade aquando da referência a uma entidade. Assim, se um endereço poder ser transferido para uma entidade diferente, não se pode usar esse endereço como identificador.

Broadcasting:

Localizar uma entidade é simples: uma mensagem, que contém o identificador de uma entidade, sofre *broadcast* para cada máquina, e cada máquina recebe um pedido para verificar se contém essa entidade. Apenas as máquinas que conseguem oferecer um ponto de acesso para aquela entidade enviam uma mensagem em resposta, que contém o endereço do ponto de acesso.

Este princípio é conhecido, na web, como ARP. Uma máquina faz *broadcast* de um pacote na rede local, perguntando quem é o dono de um dado IP. As máquinas que recebem o pedido verificam se contêm aquele endereço. Em caso positivo, é enviado uma mensagem de resposta.

No entanto, o *broadcasting* torna-se ineficiente quando a rede cresce- não escala para lá da rede local.

Forwarding pointers:

Quando uma entidade de move de “A” para “B” deixa, em “A”, uma referência para a sua nova localização, em “B”. A principal vantagem é que, assim que uma entidade pode ser localizada, um cliente pode procurar pelo endereço atual, seguindo uma rede de *forwarding pointers*.

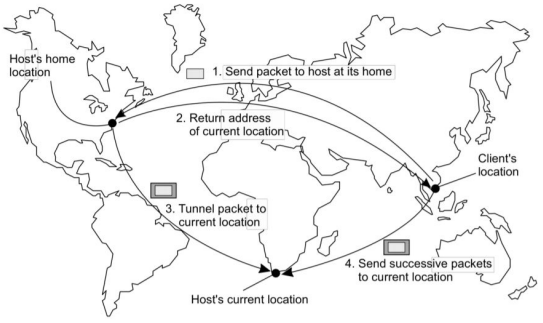
Mobile IP:

Home location é uma abordagem usada para suportar entidades mobile em larga escala, mantendo rastreio da localização atual de uma entidade. A *home location* é, normalmente, escolhida como sítio onde uma entidade foi criada.

Toda a comunicação, dirigida a um dado IP, é inicializada diretamente no *host* da entidade mobile - o home agent. Este está localizado numa rede local, correspondendo ao endereço de rede que contém o endereço IP do *mobile host*. Cada vez que um *mobile host* se move para outra rede, pede um endereço temporário, que pode ser usado para comunicação - care-of address, que está registado no *home agent*.

Quando o *home agent* recebe o pacote do *mobile host*, procura pela localização atual do *host*. Se o *host* estiver na rede local atual, o pacote é encaminhado. Se não, o pacote é tunelado para a localização atual do *host* - é empacotado como informação, num endereço IP, e enviado para o *care-of address*.

Um aspeto importante deste mecanismo é o facto de estar escondido das aplicações. Ou seja, o endereço IP associado a um *mobile host* pode ser usado por uma aplicação, sem mais delongas.



O *software* do lado do cliente, que constitui parte da comunicação independente da camada de aplicação, vai tratar do redirecionamento para o alvo da localização atual. Assim, na localização do alvo, a mensagem que foi tunelada vai ser desempacotada e tratada pela aplicação, no *mobile host*, como se estivesse a usar o endereço original. O cliente, primeiro, tem que contactar a *home*, que pode estar numa localização completamente diferente. Em consequência, a latência da comunicação aumenta.

Outra desvantagem é o facto de se usar uma *home location* fixa. Deve ser assegurado que existe sempre uma *home location* ou, então, será impossível localizar a entidade. Uma solução passa por registar a *home* num serviço de *naming* tradicional e deixar que o cliente

comece por procurar a localização da *home*. Como a *home location* pode ser considerada relativamente estável, a localização pode ser armazenada em cache logo a seguir a ter sido procurada.

Distributed Hash Tables (DHT):

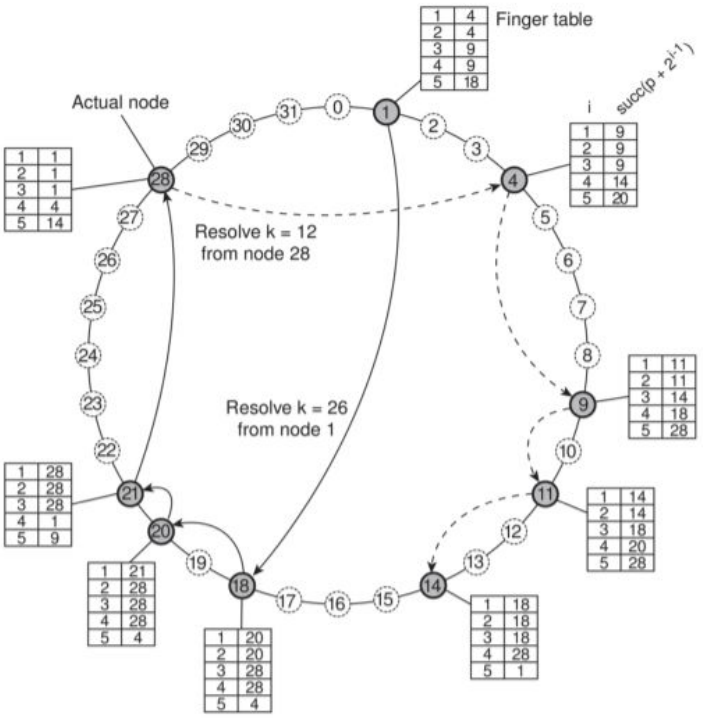
Pretende-se, agora, transformar um identificador para o endereço de uma entidade associada.

O sistema de cordas usa um identificador com *m-bits* para, de forma aleatória, atribuir identificadores aos nós, bem como chaves às entidades específicas. Uma entidade com uma chave, "*k*", fica sob a jurisdição do nó com o menor id, em que $id \geq k$. Este nó é o sucessor de "*k*", e é denotado por *succ(k)*.

O problema principal nos sistemas baseados em DHT é conseguir, de forma eficiente, resolucionar uma chave "*k*" a um endereço de *succ(k)*. Uma solução, não escalável, é fazer com que cada nó "*p*" se mantenha a par do sucessor *succ(p+1)*, bem como do seu precursor, *pred(p)*. Nesse caso, cada vez que um nó "*p*" recebe um pedido para resolucionar uma chave "*k*", vai encaminhar o pedido para um dos seus vizinhos, em que se aplique $pred(p) < k \leq p$ - o nó "*p*" deve devolver o seu próprio endereço, que iniciou a resolução da chave "*k*".

Ao invés desta solução linear, cada nó deve manter uma finger table, com máximo de "*m*" entradas.

Para procurar uma chave "*k*", o nó "*p*" vai, imediatamente, encaminhar o pedido para o nó "*q*", que tem index "*j*" na *finger table* de "*p*", onde $q = FT[j] <= k < FT[j+1]$.



Vamos supor que, na figura, queremos resolucionar k=26, partindo do nó 1.

1. Primeiro, o nó 1 vai procurar por k=26 na sua *finger table*. Descobre que este valor (k=26) é superior a $FT_1[5]$ sendo, assim, o pedido encaminhado para o nó $18=FT_1[5]$;
2. O nó 18 vai selecionar o nó 20, visto que $FT_{18}[2] <= k < FT_{18}[3]$;
3. Finalmente, o pedido é encaminhado do nó 20 para o nó 21, e daí para o nó 28, que é responsável por k=26. Por esta altura, o endereço do nó 28 é retornado para o nó 1, e a chave foi, assim, resolvida. Por razões similares, quando é pedido ao nó 28 para resolucionar a chave k=12, um pedido vai ser roteado, como se vê na linha a tracejado.

Geralmente, uma procura requer *O(log N)* passos, sendo que *N* é o número de nós do sistema.

Num sistema distribuído, a coleção dos nós participantes pode mudar a qualquer altura. Não só os nós juntam-se e abandonam de forma voluntária, como também se pode dar o caso de falharem.

Cada nó "*q*" vai, de forma regular, verificar se o seu precursor está vivo. Se não, a única coisa que "*q*" pode fazer é gravar este facto, definindo "*preq(q)*" como "*unknown*".

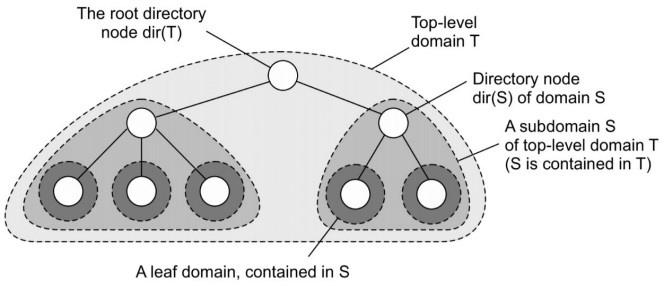
Explorar a proximidade de rede:

Existem 3 formas, baseadas em DHT, de manter o sistema a par da rede subjacente:

1. Atribuição de nós com base na topologia: atribuir identificadores de forma que dois nós, próximos, tenham identificadores que sejam também próximos. No entanto, isto pode levar a que nós, na mesma rede de empresa, tenham identificadores com um intervalo extremamente pequeno. Quando essa rede se tornar inalcançável, vai ter-se, de forma abrupta, uma falha na distribuição uniforme de identificadores.
2. Roteamento de proximidade: os nós mantêm uma lista alternativa para quem enviar pedidos. Por exemplo, em vez de um nó, no *Chord*, ter apenas um sucessor, vai manter-se a par de vários sucessores, e encaminhará o pedido para aquele que se encontrar mais perto.
3. Seleção do vizinho por proximidade: a ideia é otimizar as tabelas de roteamento, de forma a que o nó mais próximo seja o escolhido para vizinho. Esta seleção apenas funciona quando há mais nós por onde escolher.

Hierarchical Location Services (HLS):

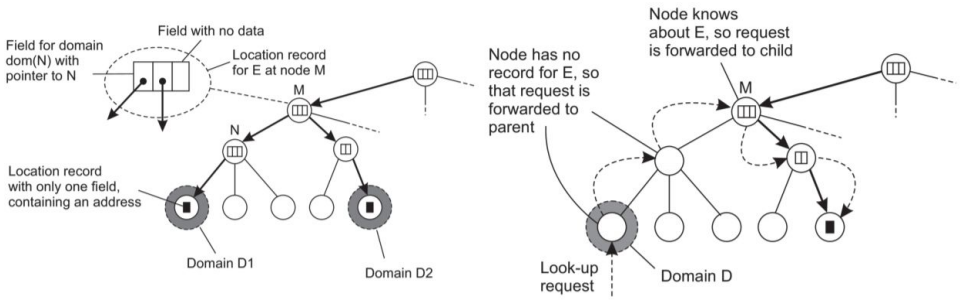
Num esquema hierárquico, uma rede está dividida numa coleção de domínios - existe um único domínio, no topo, que estende toda a rede. Cada domínio pode ser subdividido em vários, mais pequenos. Um domínio de nível mais baixo é chamado de leaf domain que, tipicamente, corresponde a uma rede local, num computador da rede ou célula num tel.mobile da rede. O pressuposto geral é que, num domínio mais pequeno, demora menos para passar uma mensagem, de nó em nó, do que num domínio maior.



Cada domínio "*D*" tem associado um nó diretório, dir(D), que mantém o rastreamento das entidades naquele domínio. Isto leva a uma árvore de nós diretórios e, no nível topo do domínio, existe um nó diretório, apelidado de root (directory) node, que sabe sobre todas as entidades (como é de notar na imagem acima).

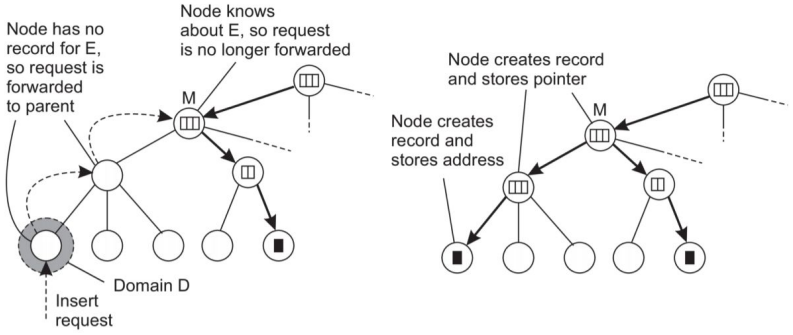
De forma a manter o rastro da localização de uma entidade, cada uma que se encontra localizada no domínio "*D*" é representada por um location record, no nó diretório *dir(D)*. Consequentemente, o *root node* vai ter o *location record* para cada entidade, onde cada *location record* guarda um ponteiro para o nó diretório no subdomínio do domínio abaixo, onde está localizada a entidade associada a esse *record*.

Organização da árvore e pesquisa:



- Fig.a): Uma entidade pode ter vários endereços, por exemplo, se for replicada. Se uma entidade contiver um endereço em D1 e D2, o nó diretório do domínio mais pequeno, contendo D1 e D2, vai conter 2 ponteiros, um para cada subdomínio que contém o endereço.
- Fig.b): Se o nó diretório não guardar o *location record* da entidade "E", esta não estará localizada em D. Consequentemente, o nó encaminha o pedido para o seu pai. É de notar que o pai representa um domínio maior do que o seu filho. Se o pai também não contiver *location record* para "E", o pedido de procura é encaminhado para um nível mais alto, e assim em diante.

Inserção:



Supondo que uma entidade "*E*" criou uma réplica, no *leaf domain* "*D*", onde precisa de inserir um endereço.

A inserção tem início no *leaf node dir(D)*, de "*D*", que, imediatamente, encaminha o pedido de inserção para o seu pai. O pai irá, por sua vez, encaminhar o pedido de inserção, até que este chegue ao nó diretório "*M*", que já tem armazenado o *location record* de "*E*".

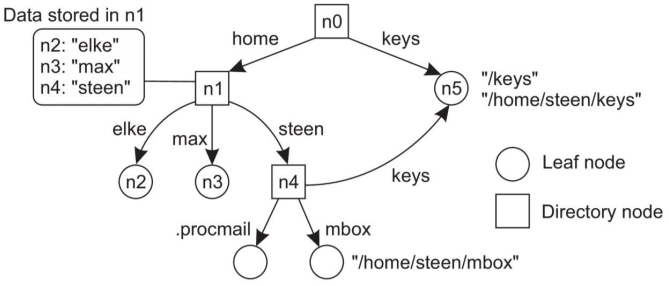
O nó "*M*" vai, então, guardar um ponteiro no *location record* de "*E*", que faz referência ao nó filho- onde o pedido de inserção teve origem. Nesta altura, o nó filho cria o *location record* para "E", que contém um ponteiro para o nó do próximo nível mais baixo, de onde veio o pedido. Este processo continua até se atingir o *leaf node*, onde começou a inserção. O *leaf node* vai, finalmente, criar um registo com o endereço da entidade, no *associated leaf domain*.

Uma alternativa passa por criar um *location record* antes de passar o pedido de inserção ao nó pai. Ou seja, a cadeia de ponteiros será, aqui, construída de baixo para cima. A vantagem, nesta abordagem, é o facto de se tornar possível fazer *lookups*. Consequentemente, se um nó pai ficar temporariamente inalcançável, o endereço pode ser procurado pelo domínio representado pelo nó atual.

Structured naming:

Os nomes estão comumente organizados em name spaces. Estes podem ser representados como grafos dirigidos com dois tipos de nós. Um leaf node representa uma entidade nomeada, e tem a propriedade de não ter arestas de saída. Normalmente, o *leaf node* guarda informação sobre a entidade que está a representar.

Em contraste ao *leaf node*, existe o directory node- tem um número de arestas de saída, todas com nome. Ele guarda uma tabela onde constam as arestas de saída, representadas como um par. Esta tabela tem o nome de directory table.



Na figura acima, o nó *n0* só contém arestas de saída, não contendo de entrada- nó root (raiz) do grafo. Cada caminho, num *naming graph*, pode ser referido como uma sequência de rótulos, correspondentes às arestas desse caminho. Uma sequência do tipo *N:[label 1 , label 2 , ..., label n]* tem o nome de path name. Se o primeiro nó num *path name* for a raíz do *naming graph*, é chamado de absolute path name. Caso contrário, tem o nome de relative path name.

Como os nomes estão sempre organizados num name space, um nome está sempre definido relativamente a um nó diretório.

Assim, a diferença entre nomes globais e locais pode ser confusa:

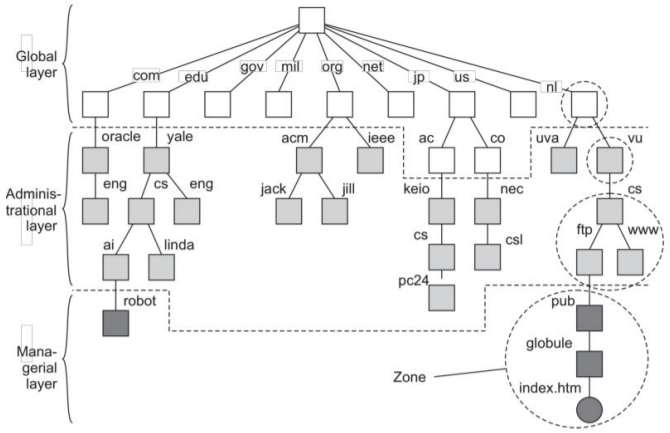
- Nome global: nome que denota a mesma entidade, não interessando onde é que este nome está a ser usado no sistema. Um nome global é sempre interpretado com respeito ao mesmo nó diretório.
- Nome local: a interpretação do nome depende do sítio onde o nome está a ser usado. É, essencialmente, um nome relativo, cujo diretório em que está contido é (implicitamente) conhecido.

Implementação de um name space:

Um name space permite que os utilizadores e processos adicionem, removam e procurem por nomes.

Assumindo-se que um *name space* tem apenas um nó raiz, é possível distinguir 3 camas:

- Camada global: formada pelos nós de mais alto nível, ou seja, o nó raiz e outros nós diretórios que estão logicamente perto da raiz (como os seus nós filhos). Os nós desta camada são caracterizados pela sua estabilidade, de maneira que as *directory tables* raramente mudam.
- Camada administrativa: formada pelos nós diretório que, em conjunto, são gerenciados por uma única organização. Os nós desta camada representam grupos de entidades que pertencem à mesma organização ou unidade administrativa.
- Camada gerencial: contém nós que têm tendência a mudar regularmente. Os nós desta camada são mantidos, não só pelos administradores do sistema, bem como pelos utilizadores individuais de um sistema distribuído.



O *name space* está dividido em partes que não colidem, chamadas zonas, no *DNS* (como na figura acima, que mostra um exemplo de separação de parto do name space do DNS). Uma zona é uma parte do *name space* que é implementada por um *name server* (sendo um *name server* uma aplicação que permite ter respostas a pedidos enviados a um serviço de diretório) separado.

Os *name servers,* em cada camada, são obrigados a conhecer diferentes requisitos. Se um *name server* falhar, uma grande parte do *name space* vai ficar inalcançável, visto que não pode acontecer *name resolution* com o servidor a falhar.

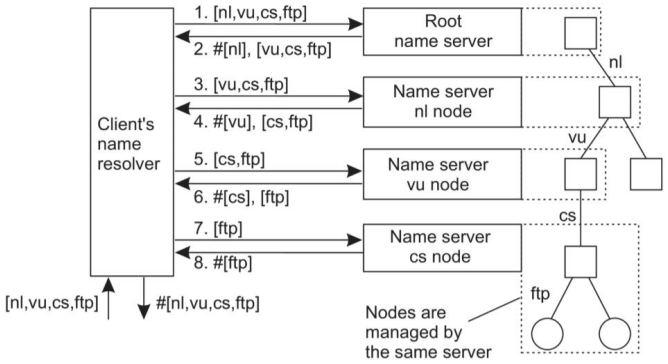
Issue	Global	Administrational	Managerial
Geographical scale	Worldwide	Organization	Department
Number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Client-side caching	Yes	Yes	Sometimes

Implementação de um name resolution:

Name resolution é quando, através do nome de um caminho, é possível encontrar informação guardada num nó, referido por esse nome. Saber como, e onde começar o *name resolution* é referido como closure mechanism - lida com a seleção inicial de um nó num *name space*, onde é suposto o *name resolution* começar.

De forma a explicar a implementação de um *name resolution* em larga escala, vai-se assumir que os *name servers* não são replicados e que as caches do lado do cliente não são usadas. Cada cliente tem, então, acesso ao name resolver local, que é responsável por garantir que a *name resolution* é realizada.

Name resolution iterativa:



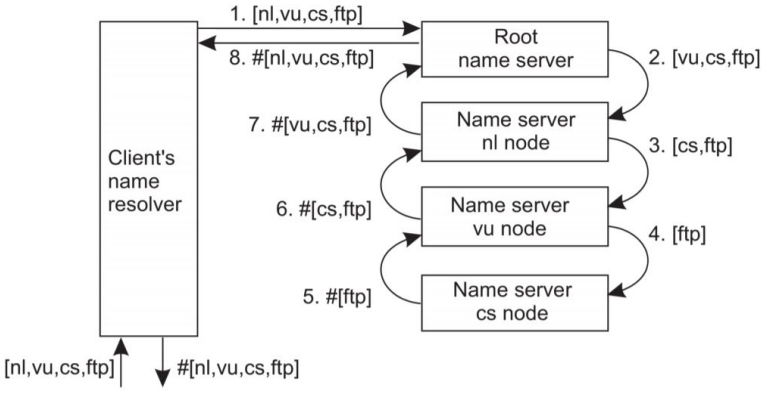
O *name resolver* passa o nome completo para o name server raiz. Assume-se que o endereço, onde o servidor raiz pode ser contactado, é bem conhecido. O servidor raiz vai resolucionar o *name path* enquanto conseguir, e retornar o resultado ao cliente.

No exemplo acima, ocorrem 5 passos principais:

1. O servidor raiz apenas pode resolucionar a *label nl*, para a qual vai retornar o endereço do name server associado;
2. O cliente vai passar o que sobra do *path name* para esse *name server*. Este servidor apenas pode resolucionar a *label vu*, retornando assim o endereço do *name server* associado, bem como o que sobra do *path name*;
3. O *name resolver* do cliente vai, então, contactar o próximo *name server*, retornando o endereço de FTP e o que sobre do *path name*;
4. O cliente contacta o servidor FTP, pedindo-lhe que resolva a última parte do *path name*;
5. O servidor FTP vai resolucionar as etiquetas e transferi-las para o ficheiro pedido.

Ps: quando se diz “resolucionar” pretende-se traduzir “resolve”, mas não há uma tradução 100% correta.

Name resolution recursiva:



Aqui, um name server passa o resultado ao próximo name server que encontra.

Seguindo o exemplo acima, mas com resolução recursiva, o servidor seguinte vai resolucionar o *path name* por completo e, eventualmente, devolver o ficheiro *index.html* ao servidor raiz que vai, em resposta, passar o ficheiro para o *name resolver* do cliente.

O problema, nesta abordagem, é o facto de exigir um maior desempenho em cada *name server* - cada um destes tem que tratar, por completo, da resolução de um *path name*.

Ainda assim, tem algumas vantagens:

- Fazer cache dos resultados é mais efetivo;
- Os custos de comunicação podem ser reduzidos.

Permite, ainda, que cada *name server* aprenda, de forma gradual, o endereço de cada name server responsável por implementar nós de mais baixo nível.

DNS name space:

Está organizado de forma hierárquica, como uma *rooted tree*. Como cada nó do *DNS name space* tem exatamente uma aresta de entrada, a etiqueta associada a essa aresta é também usada como nome para esse nó. Uma *subtree* é chamada de domain, e o *path name* para o seu nó raiz é chamado de domain name.

O conteúdo de um nó é formado por uma coleção de resource records, estando os principais na figura abaixo:

SOA	Zone	Informação sobre a zona representada
A	Host	Endereço IP do host que o nó representa
MX	Domain	Servidor de eMail para o nó
SRV	Domain	Servidor que lida com um dado serviço
NS	Zone	Servidor de nomes para a zona representada
CNAME	Node	Apontador simbólico
PTR	Host	Nome canónico de um host
HINFO	Host	Informação sobre o host
TXT	Any Kind	Qualquer Informação considerada útil

Aqui, um nó representa várias entidades ao mesmo tempo.

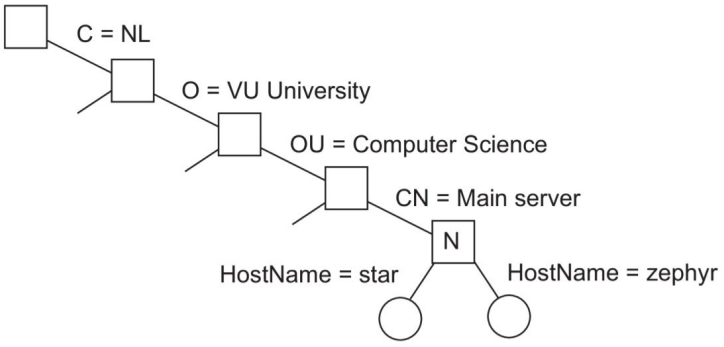
Attribute-based naming - LDAP:

Uma abordagem comum, para combater os serviços de diretórios distribuídos, é combinar *structured naming* com *attribute-based naming*.

O mais comum é usar um LDAP - lightweight directory access protocol. Este serviço consiste num número de registos, também referidos como directory entries. Um *directory entry* é comparável a um *resource record* no DNS. Cada registo contém uma coleção de pares, onde cada atributo tem um tipo associado.

A coleção de todas as entradas de diretórios num diretório LDAP é chamado de directory information base (DIB). Aqui, cada registo tem um nome único, de forma a que possa ser procurado. Cada atributo de nome é chamado de relative distinguished name (RDN).

O uso de nomes globais únicos leva a uma coleção hierárquica de entradas de diretórios - directory information tree (DIT) que, essencialmente, forma um *naming graph*, onde cada nó representa uma entrada de diretório.



Um nó, num *LDAP naming graph*, pode representar, simultaneamente, um diretório (no sentido tradicional), bem como um *LDAP record*. Esta distinção é suportada por duas diferentes operações de procura:

- Read: usada para ler cada registo, dado o seu path name no DIT;
- List: usada para listar os nomes das arestas de saída de um dado nó no DIT.