Os 5 Princípios da Programação Distribuída

1. **Distribuir com Moderação**
2. **Localizar (juntar) partes relacionadas**
3. **Usar Interfaces com granularidade compacta em vez de fina**
4. **Usar preferencialmente *stateless objects* em vez de *Stateful objects***
5. **Desenvolver para uma Interface e não para uma implementação**

Princípio 1. - Distribuir com Moderação

Este princípio é baseado num facto simples e inegável.

"invocar um método num objecto de um processo diferente é centenas de vezes mais lento que invocá-lo no mesmo processo. Mover o objecto para outra máquina numa rede e chamá-lo a partir de outra máquina pode ser outras 10 vezes mais lento"

Então quando distribuir ?

A resposta banal é: somente quando tiver que ser !

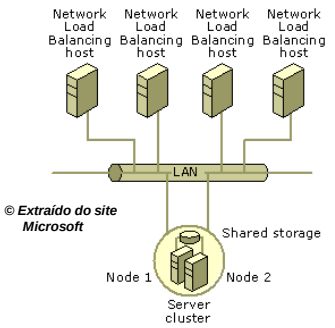
Algumas razões para Distribuir:

A. Sistemas de Gestão de Base de Dados (SGBD)

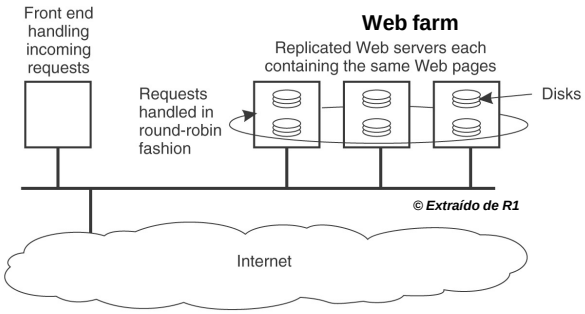
- **Executam-se num servidor dedicado, sendo software complexo, caro e que requerem hardware potente e por isso caro. Assim, não é fácil distribuir muitas cópias dos SGBD;**
- **Os SGBD contêm dados relacionados e partilhados por várias aplicações, sendo isto possível colocando vários servidores aplicacionais acedendo á mesma Base de Dados em vez de possuírem a sua cópia local;**
- **Os próprios SGBD são desenhados para serem executados num nível físico separado, expondo uma interface designada SQL (*Structured Query Language*)**

B. Balanceamento de Carga (distribuição horizontal)

- **Um servidor pode ser separado em partes logicamente equivalentes, garantindo que cada parte opera independentemente, incluindo réplicas dos dados ou partilhando um dispositivo de armazenamento.**



© Extraído do site Microsoft



© Extraído de R1

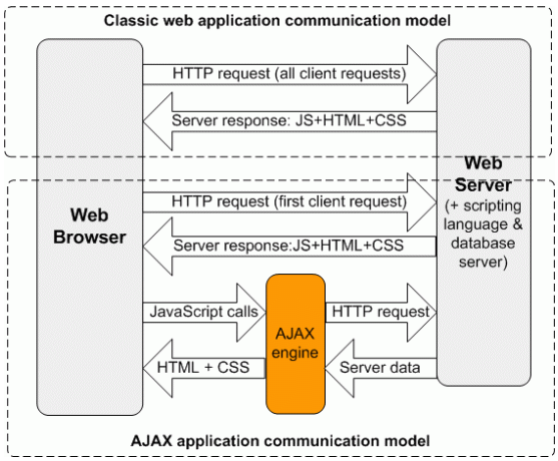
C. Distribuir a lógica de apresentação

(de decisão mais complexa)

De acordo com o principio de "distribuir moderadamente", as últimas tendências, indicam que se deve executar toda a lógica num servidor e simplesmente enviar HTML para os clientes em Web Browser. A consequência é ter problemas de desempenho, dado que cada interacção do utilizador tem de "viajar" para o servidor para que este possa gerar uma resposta.

Antes da proliferação da Web era mais comum executar toda a apresentação e até mesmo parte da lógica de negócio do lado do cliente, proporcionando a interacção com o utilizador mais rápida (menos "viagens" para o servidor), mas levantando um problema complexo de como manter actualizadas todas as máquinas de todos os clientes, por exemplo com uma nova versão.

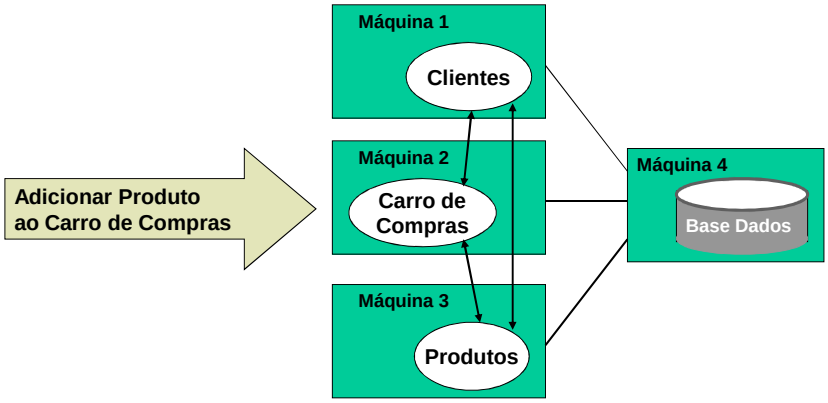
Distribuição no desenho de aplicações Web, com AJAX (Asynchronous JAvaScript and XML)



http://www.interaktonline.com/files/art/ajax/AJAX%20-%20Asynchronously%20Moving%20Forward.pdf

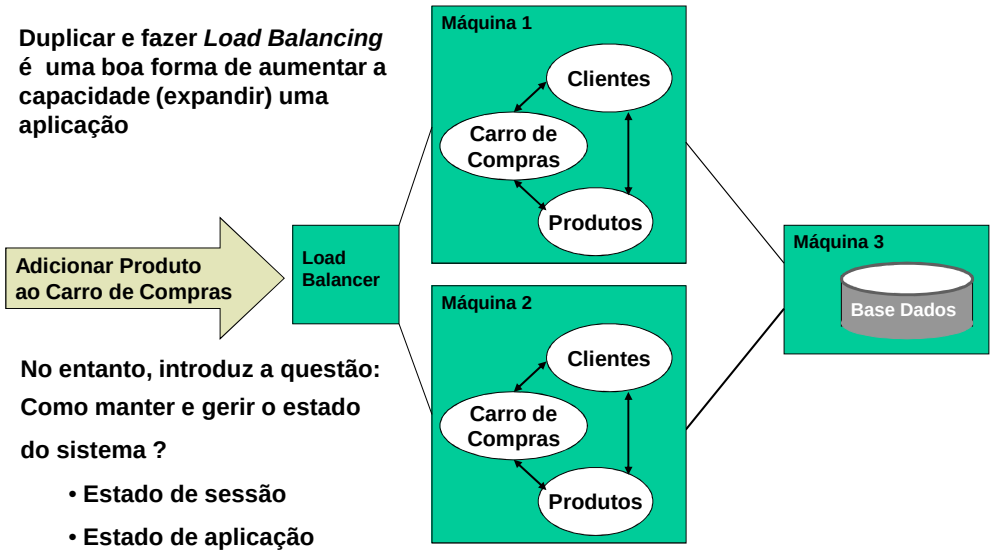
Princípio 2. – Localizar (juntar) partes relacionadas

Como NÃO desenhar uma aplicação distribuída de comércio electrónico



Distribuição Horizontal

Duplicar e fazer *Load Balancing* é uma boa forma de aumentar a capacidade (expandir) uma aplicação



No entanto, introduz a questão:

Como manter e gerir o estado do sistema ?

- **Estado de sessão**
- **Estado de aplicação**

Princípio 3) - Usar Interfaces com granularidade compacta em vez de fina

```
public class Cliente {
     public string FirstName;
     public string LastName;
     public string Email;
     // ... outros atributos

     public void Create(){} // Criar o cliente
     public void Save(){} // Salvar alterações
}

public class GestaoClientes {
     // ...
     public void SomeMethod() {
         Cliente cli = new Cliente();
         cli.Create();
         cli.FirstName="João"; cli.LastName="Silva";
         cli.Email="jsilva@yyy.pt";
          // ... actualiza outros atributos
         cli.Save();
     }
}
```

chatty **interface**
ou
fine-grained **interface**

Se a gestão de clientes se executar numa máquina e o objecto do tipo `Cliente` noutra, qualquer alteração de estado do objecto (atributo ou chamada a métodos) implica comunicação, o que se poderá traduzir em sérios problemas de desempenho

A solução passa por ter uma granularidade maior na Interface com o objecto remoto, neste caso `Cliente`, através de parâmetros com todos os atributos do cliente.
Neste caso é passado um objecto *serializable*

```
[Serializable]
public class DadosCliente {
    public string FirstName;
    public string LastName;
    public string Email;
    // ... outros atributos
}

public class Cliente {
    public void Create(DadosCliente dcli) {}
    public void Save(DadosCliente dcli) {}
}
```

chunky **interface**
ou
course-grained **interface**

Padrão: *Data Transfer Objects - DTO*

Princípio 4) - Usar *Stateless objects* em vez de *Stateful objects*

- Se o princípio 3 contradiz os puristas do *object-oriented*, este deixa-os furiosos, pois viola o principio do encapsulamento;
- No entanto para podermos tirar partido do balanceamento de carga devemos gerir o estado dos objectos com algum cuidado;

Stateless Object

Objecto que pode ser criado e destruído entre chamadas a métodos. Isto implica implementar uma classe que não pode utilizar valores dos atributos entre chamadas a métodos, logo são objectos com *chunky* interfaces.

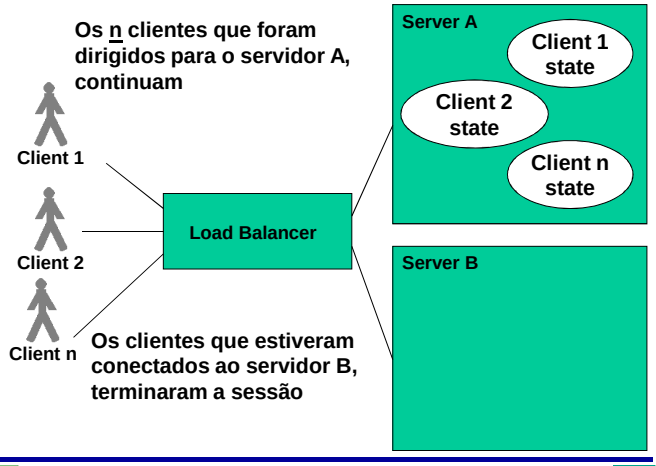
Stateful Object

Objecto que mantém estado, por exemplo valores de atributos, entre chamadas a métodos. Os objectos *Stateful*, afectam negativamente a expansibilidade por duas razões:

1. O objecto tem um tempo de vida, durante o qual pode acumular e consumir recursos escassos, provocando esperas noutros objectos, mesmo que o objecto que tem os recursos não esteja a ser utilizado;
2. Minimizam a possibilidade de replicar os objectos por vários servidores por questões de balanceamento de carga

Falta de Expansibilidade (*scalability*) com objectos *Stateful*

Num sistema sujeito a grande carga (clientes nos dois servidores), pode acontecer que todos os clientes do servidor B já terminaram a sessão, ficando assim um servidor totalmente *idle* e os utilizadores que ficaram ligados ao servidor A, continuam a estar sujeitos a eventual falta de desempenho.



Princípio 5) - Desenvolver para uma Interface e não para uma implementação

- **O código dos objectos remotos deverá poder ser alterado (por exemplo optimizado) sem implicações no lado dos clientes;**
- **Por outro lado uma evolução nas funcionalidades do lado servidor não deverá comprometer clientes anteriores;**
- **Evitar distribuir (*deployment*) por milhares de computadores clientes, versões de software só porque se alterou um pormenor no lado do servidor;**
- **Desde os primeiros modelos (Sun RPC, DCE-RPC, COM/DCOM, CORBA) que o conceito de interface tem um papel muito importante, tendo sido definidas linguagens de especificações de interfaces (IDL – Interface Definition Language);**
- **A definição de interfaces, para além de esconder detalhes de implementação, deverá também proporcionar versionamento, facilitar a interoperabilidade entre sistemas heterogéneos e permitir que seja possível invocar serviços remotos conhecendo unicamente a interface.**

Expansibilidade (*Scalability*) versus Desempenho

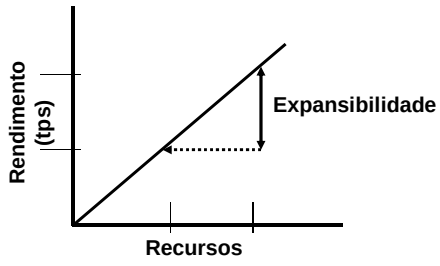
Apesar do conceito de expansibilidade de um sistema estar relacionado com o conceito de desempenho, eles não significam a mesma coisa.

Desempenho/*Performance* – é uma medida de rapidez (tempo de resposta) de um sistema.

Expansibilidade/*Scalability* – é uma medida de quanto aumenta o desempenho, quando adicionamos recursos a um sistema, tais como CPUs, memória ou computadores.

Rendimento/*Throughput* – refere-se à quantidade de trabalho que o sistema pode realizar num determinado período de tempo. Usualmente é expresso em transacções por segundo (tps).

Expansibilidade/*Scalability* – refere-se à mudança no rendimento como resultado de adicionar mais recursos



Dois tipos de Expansibilidade

Expansibilidade Vertical (*scaling up*)

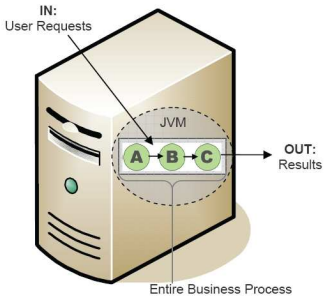
- Quando se substitui hardware lento por hardware mais rápido, por exemplo mudar de CPU de 500 MHz para 2 GHz, colocar mais memória, mais disco etc.
- Esta é a única forma de expandir um sistema por questões de desempenho, se o mesmo não foi desenvolvido tendo em conta os cinco princípios atrás enunciados. No entanto, para além de ser uma solução com custo elevado, continua a expor um único ponto de falha.

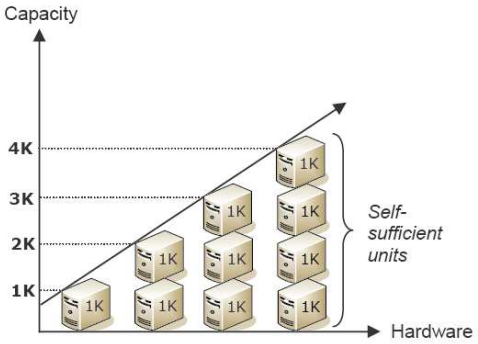
Expansibilidade Horizontal (*scaling out*)

- Quando se adiciona um servidor com balanceamento de carga para executar a mesma aplicação. Para além de proteger investimentos, proporciona também tolerância a falhas se um dos servidores falhar;
  - Possível se a aplicação suportar os 5 princípios da programação distribuída, principalmente os princípios 2 e 4

Expansibilidade Linear com Unidades Autónomas

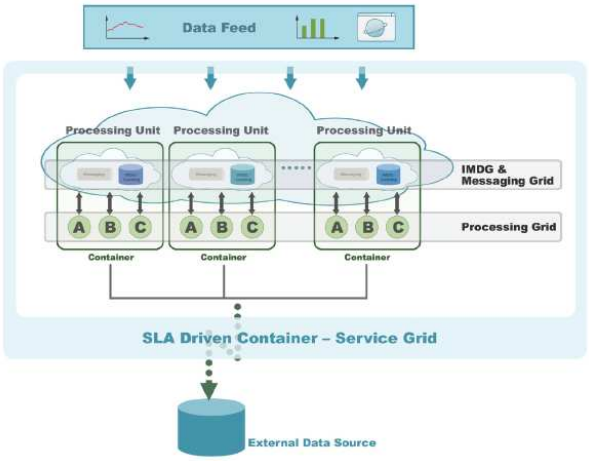
- Ocorre quando cada nova unidade hardware contribui sempre com a mesma capacidade adicional.





- Toda a camada de Negócio na mesma Unidade Autónoma;
- Implica desenvolver aplicações que implementem Processos de Negocio completos para um determinado número de Pedidos/Clientes (*aqueles que são suportados por uma única Unidade Autónoma*)

GigaSpaces eXtreme Application Platform (XAP)



http://www.gigaspaces.com/WhitePapers

Muitas vezes usam-se técnicas para aumentar o desempenho que diminuem a expansibilidade.

Um exemplo recorrente:

- **Para aumentar o desempenho, o desenvolvimento de aplicações Web com ASP (Active Server Pages) da Microsoft, é usada uma técnica de guardar (cache) informações no objecto de sessão do lado do servidor Web, para evitar, por exemplo, acessos à Base de Dados.**
- **Se a capacidade máxima de carga do servidor Web for atingida a reacção normal é adicionar outro servidor Web.**
- **Infelizmente tal situação poderá não ser possível, pois foi violado o princípio de usar preferencialmente objectos sem estado e portanto não podemos fazer expansibilidade Horizontal.**

Aspectos fundamentais a reter

- **Identificar as partes do Sistema Distribuído;**
- **Identificar a interface de cada parte;**
- **Definir o modelo de interacção entre a partes, tendo em mente que as interacções resultam em comunicação;**
- **Definir o modelo de coordenação e ordenação de eventos;**
- **Definir o modelo de falhas;**
- **Definir o modelo de expansibilidade;**
- **Definir a necessidade e modelo de Segurança, isto é, identificar o inimigo (permissões de acesso e canais de comunicação seguros);**
- **Qualidade do serviço (QoS)**