Caracterização da Computação Distribuída

Características distintivas das arquiteturas e do desenvolvimento de software de sistemas distribuídos:

- **Potencialidades e Dificuldades**

Porquê Computação Distribuída ?

Endereçar e encontrar soluções para os seguintes aspetos:

- **Escalabilidade**
- **Elasticidade**
- **Concorrência**
- **Tolerância a falhas**
- **interfaces e interoperabilidade**
- **Transparência**
- **Replicação de dados e serviços**
- **Segurança**
- **Qualidade de Serviço (*QoS - Quality of Service*)**

Algumas definições

Workload - Throughput

- **Quantidade de processamento (trabalho) que um computador/sistema é capaz de realizar num determinado tempo. Pode ser medido pelo número de utilizadores concorrentes, número de pedidos por segundo, número de mensagens etc.**

Speedup

- **Número que mede o desempenho relativo de dois sistemas para processar o mesmo problema. Tecnicamente mede a melhoria da velocidade de execução de uma tarefa executada em dois sistemas similares mas com configurações ou recursos diferentes.**

Amdahl's law (Gene Amdahl 1967)

- **Teoricamente o *speedup* de um sistema aumenta com o aumento dos recursos. No entanto, o valor de *speedup* é sempre limitado por alguma parte do sistema (tarefa) que não beneficia do aumento de recursos. A lei de *Amdahl* é muitas vezes usada na computação paralela para calcular teoricamente o *speedup* de um programa quando se usam múltiplos processadores.**

Um exemplo simples de *Speedup*

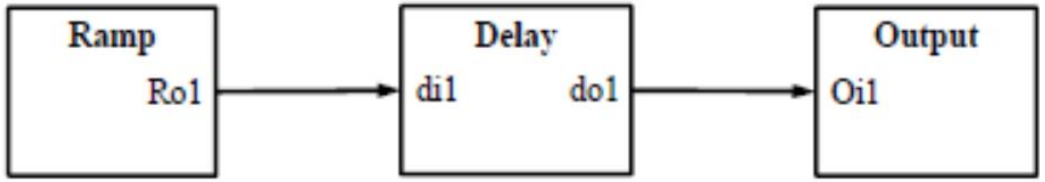


Figure 6.14: Pipeline workflow with an overloaded activity

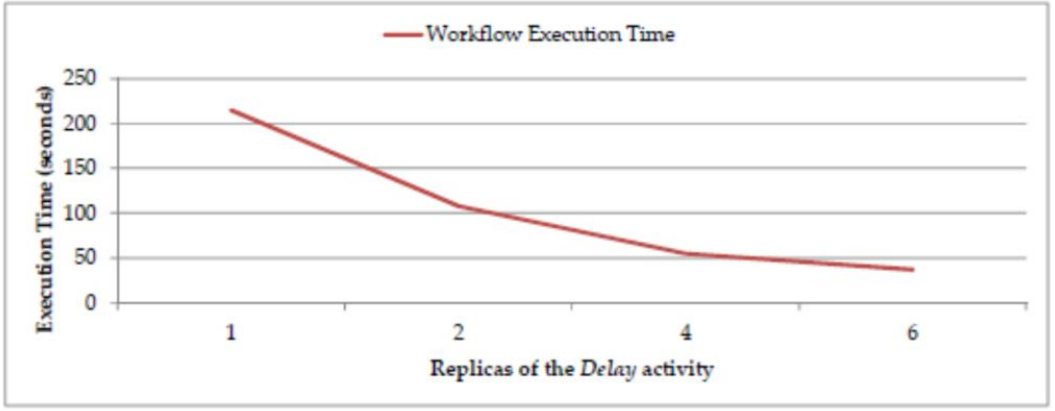


Figure 6.17: Execution times with load balancing pattern in the *Delay* activity

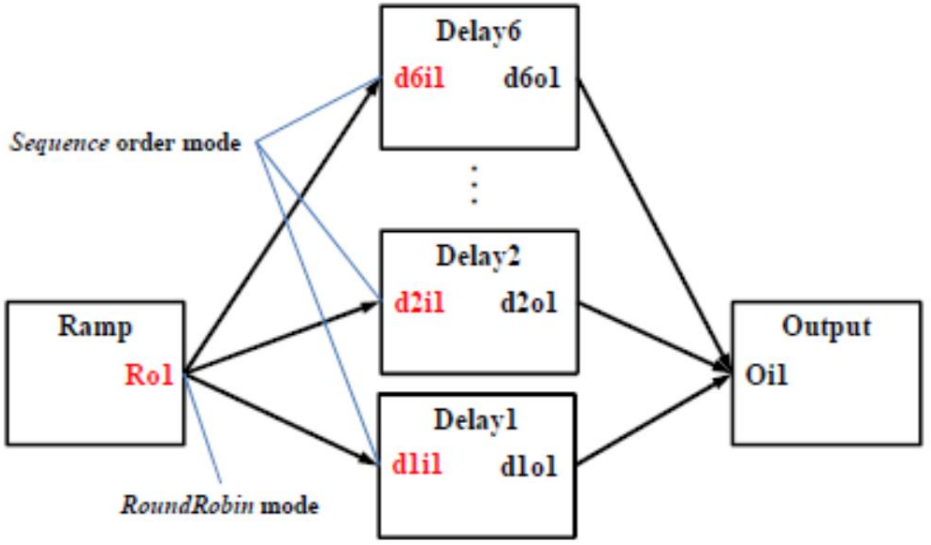


Figure 6.15: Workflow with a load balancing pattern

$$Speedup = \frac{ExecTime(1\ replica)}{ExecTime(6\ replicas)} = \frac{215}{37} = 5.8$$

Note que o *speedup* não é 6!

From: "A Model for Scientific Workflows with Parallel and Distributed Computing", Luis Assunção, FCT-UNL, 2016, *(https://run.unl.pt/handle/10362/19975)*

Exemplo da lei de *Amdahl*

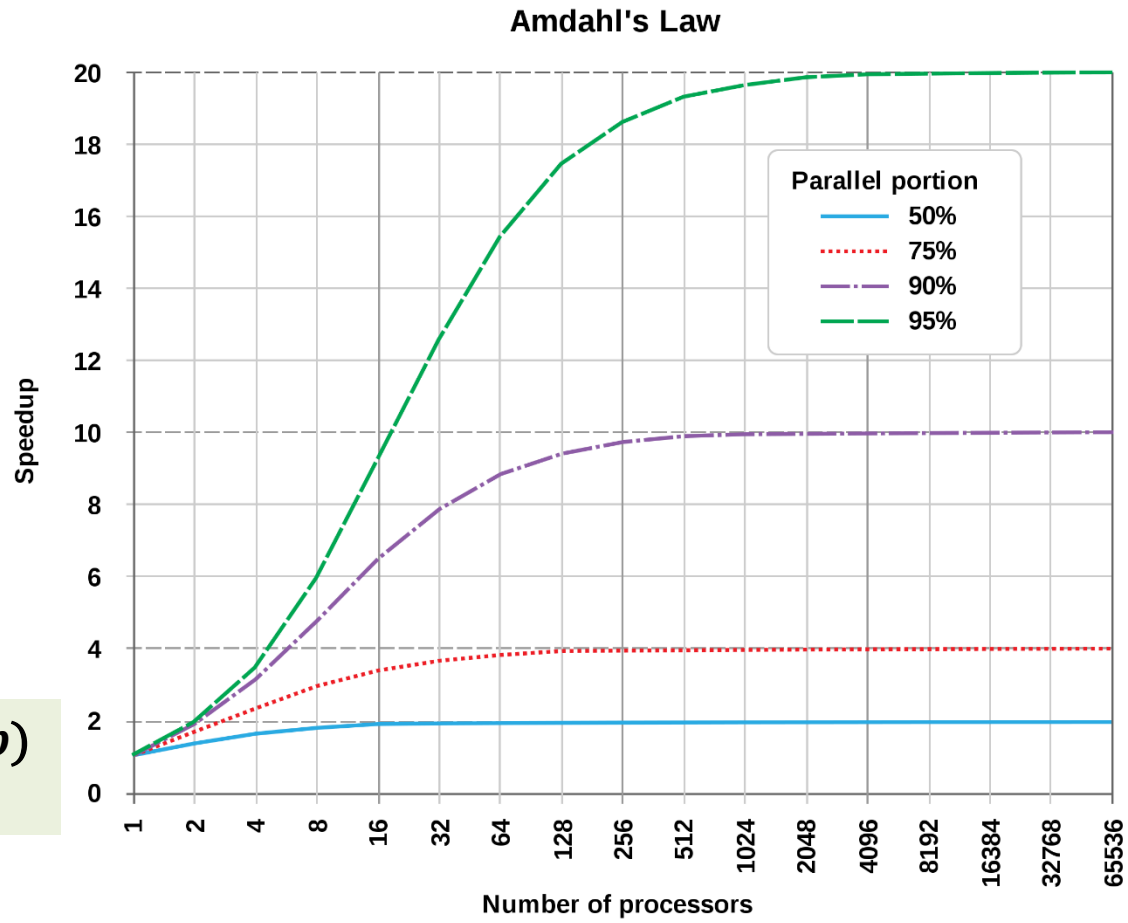
- **Programa que demora 20 horas num único processador**
  - **95% (19 horas) do programa pode ser paralelizado (p)**
  - **5% (1 hora) não pode ser paralelizado**

O *Speedup* é sempre limitado pela componente que não pode ser paralelizada $(1-p)$

Neste caso o speedup é limitado a 20 vezes

$$S \leq \frac{1}{(1-p)} = \frac{1}{(1-0.95)} = 20$$

Se a percentagem de paralelização (p) diminuir o *speedup* decresce



From: Distributed Computing in Java 9, Raja Malleswara, Rao Pattamsetti, Packt, 2017

Availability (disponibilidade)

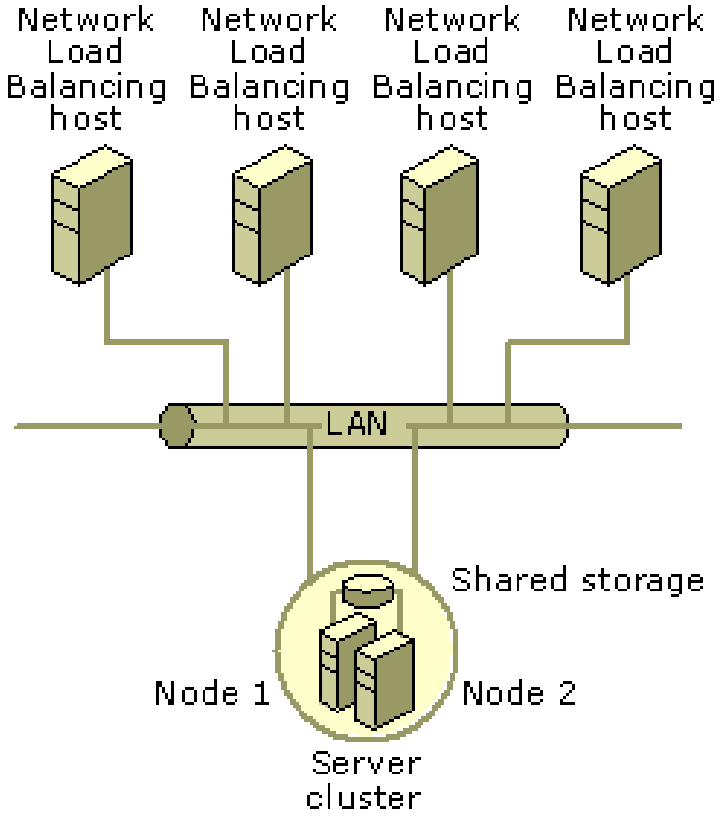
- ***Availability*** **– razão em percentagem do tempo em que um sistema funciona (*Uptime*) versus o tempo em que o sistema deveria funcionar;**

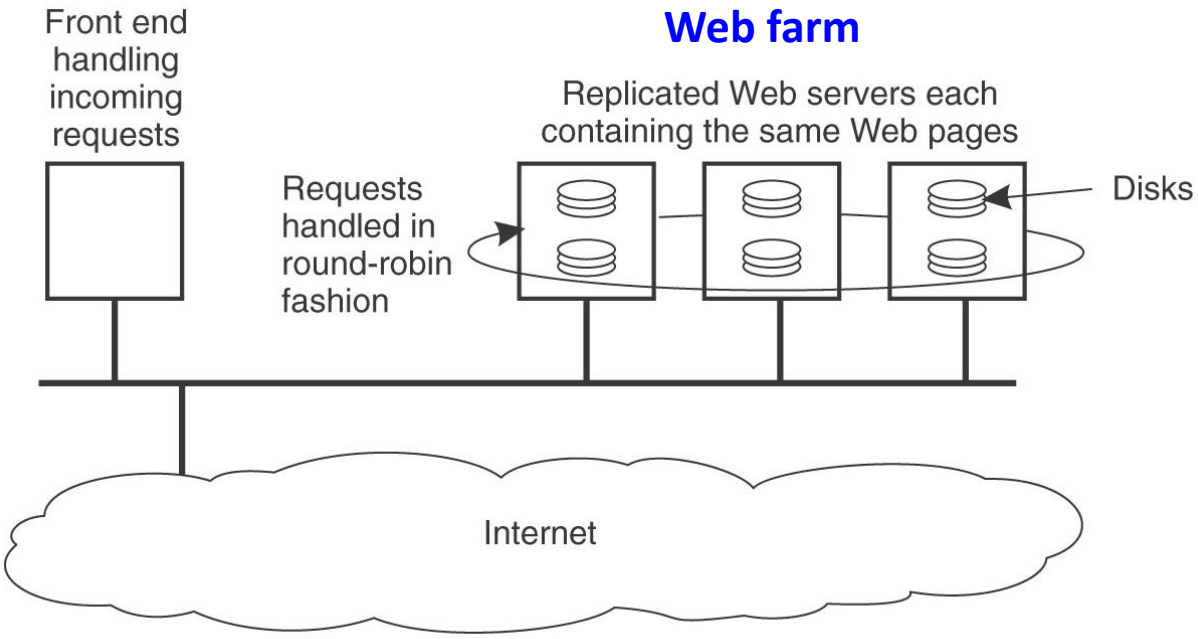
$$\frac{Uptime}{Uptime + Downtime}$$

- ***"Five nines"*** **ou 99.999% é normalmente o objetivo de ambientes críticos, isto é, 5,26 minutos de *downtime* por ano ou cerca de 26 segundos de *downtime* por mês;**
  - ✓ **Para ter *"Five nines"* a intervenção humana para recuperação de falhas é irrealista, pelo que as infraestruturas de alta disponibilidade têm de ser capazes de recuperar de falhas automaticamente sem intervenção humana.**

Balanceamento de Carga (escalabilidade por distribuição horizontal)

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





Escalabilidade

- **Um sistema é escalável (*scalable*) se ao aumentar os recursos (ex: número de processadores) ou perante o aumento do *workload* (ex: número de utilizadores, número de pedidos, número de transações por segundo) permanece efetivo, isto é, disponível, com desempenho e tempos de resposta aceitáveis.**

- **A escalabilidade levanta os seguintes desafios:**
  - **Custo controlado de recursos físicos (computadores, memória principal, memória secundária, periféricos, …)**
  - **Avaliação antecipada da evolução do sistema em termos de volumes de informação, engarrafamentos de mensagens, saturação do espaço de nomes, etc. Por exemplo, a previsão à partida da saturação de endereços IPv4 (32 bits) para IPv6 (128 bits);**
  - **Controlo da perda de desempenho, definindo quais os indicadores e métricas associadas:**
    - **% de CPU e memória**
    - **Número de pedidos ou transações por segundo**
    - **Volume de dados**
    - **Tempos de resposta, latências,**
    - **. . .**

Escalabilidade versus Desempenho

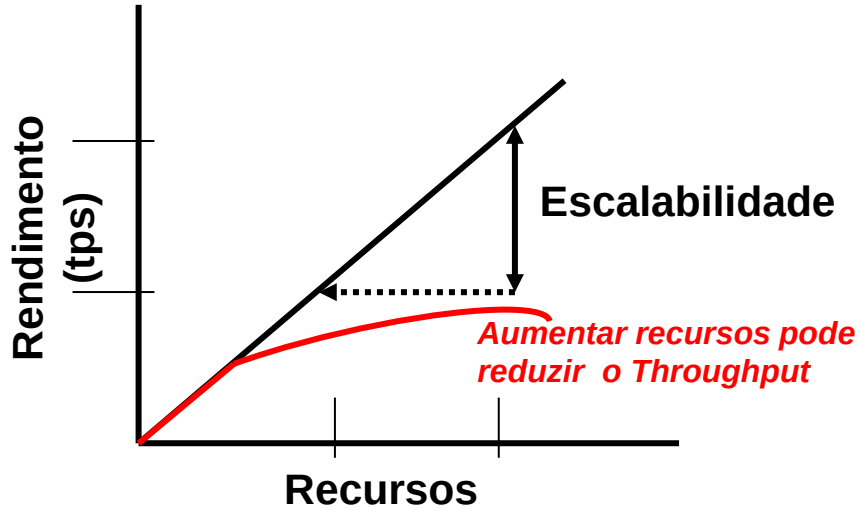
Apesar do conceito de escalabilidade de um sistema estar relacionado com o conceito de desempenho, eles não significam a mesma coisa.

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

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

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).

Escalabilidade/*Scalability* – refere-se à mudança no rendimento (suportar aumento de carga) como resultado de adicionar mais recursos



Tipos de Escalabilidade

Escalabilidade Vertical (*scaling up*)

- Quando se substitui hardware lento por hardware mais rápido, por exemplo mudar de CPU de 500 MHz para 2 GHz, mais processadores, 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 alguns princípios da computação distribuída. No entanto, para além de ser uma solução com custo elevado, continua a expor um único ponto de falha.

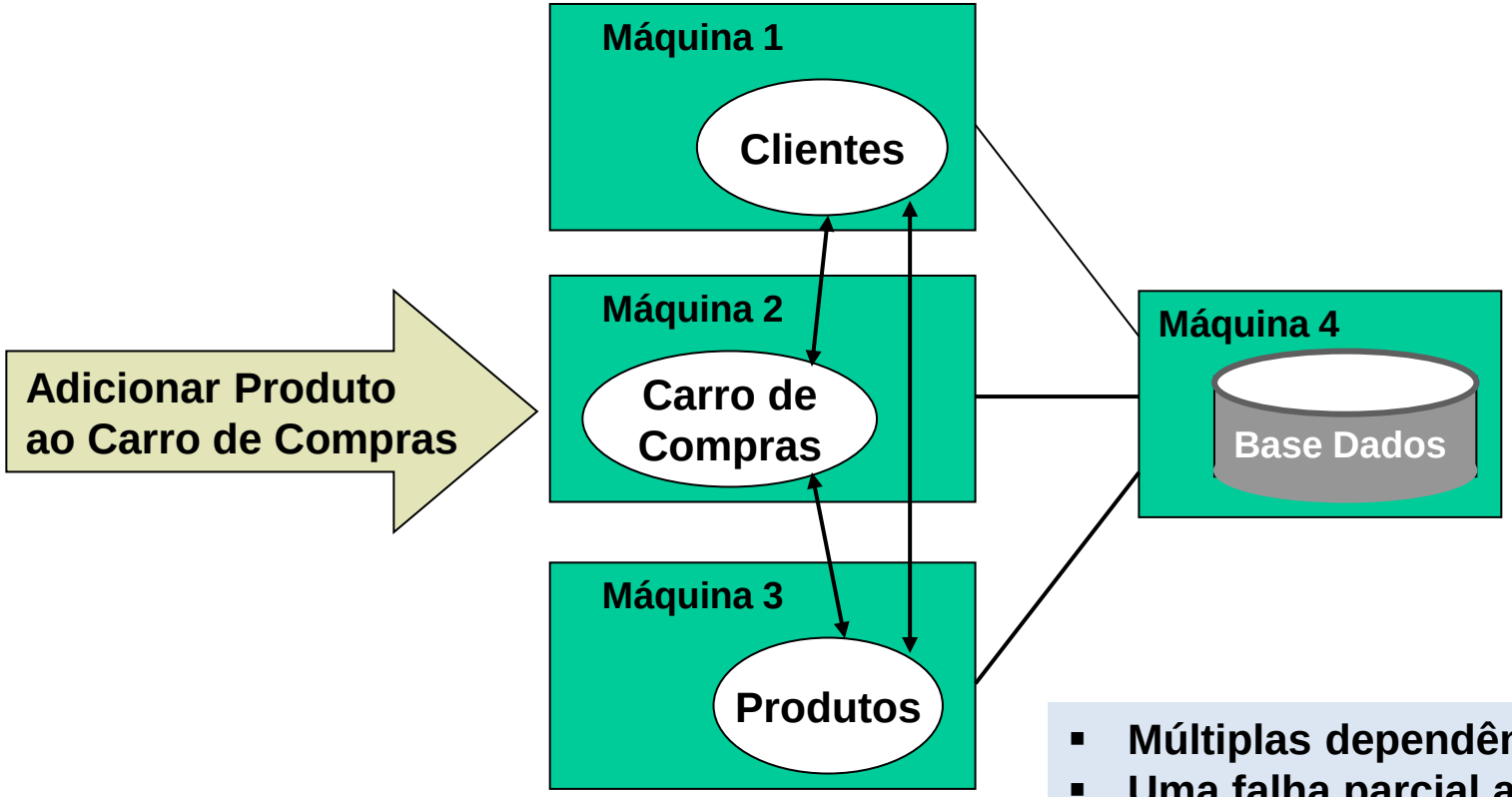
Escalabilidade Horizontal (*scaling out*)

- Quando se adicionam réplicas (por exemplo um servidor) para executar a mesma aplicação usando técnicas de equilíbrio de carga (*load balancing*). Para além de menores custos, adiciona tolerância a falhas se uma das réplicas falhar;
  - Possível se o desenvolvimento das aplicações suportar o bom princípio da programação distribuída de usar componentes *stateless*;
  - Pode conduzir a situações de sobredimensionamento (custos) para resolver *workloads* de pico.

Localizar (juntar) partes relacionadas

- ✓ **Assumindo que existiam previamente dois serviços independentes (Clientes e Produtos)**

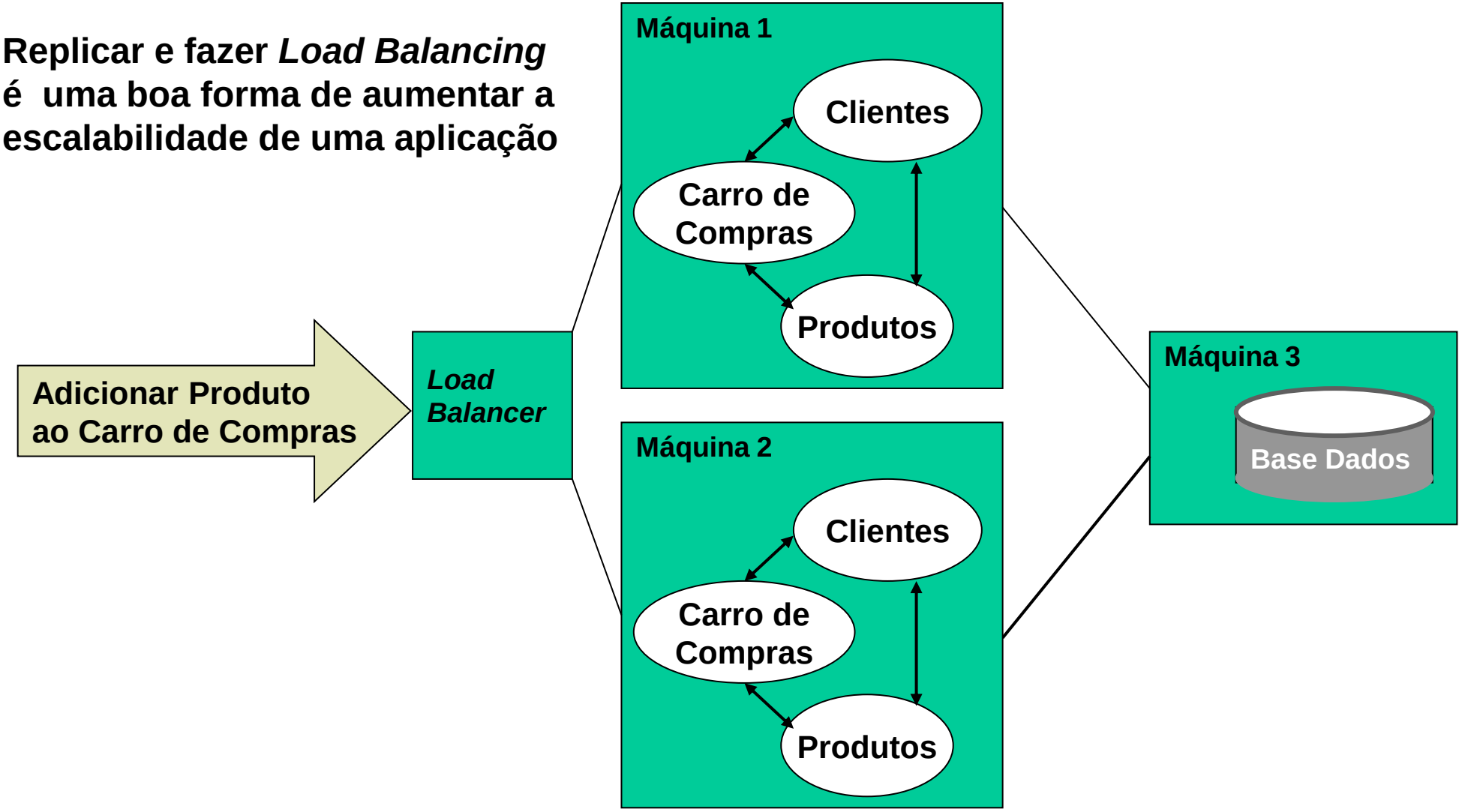
- ❖ **Como <u>NÃO</u> desenhar uma aplicação distribuída**



- **Múltiplas dependências**
- **Uma falha parcial afeta todo o sistema**
- **Não existe distribuição de carga**

Distribuição Horizontal

Replicar e fazer *Load Balancing* é uma boa forma de aumentar a escalabilidade de uma aplicação



Limitações à escalabilidade

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

Um exemplo recorrente:

- **Para aumentar o desempenho, no desenvolvimento de aplicações Web é usada uma técnica de guardar em memória (cache) informações num 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 por escalabilidade horizontal.**

- **Infelizmente tal situação poderá não ser viável, pois foi violado o princípio de usar preferencialmente objectos sem estado (*Stateless*) e portanto não podemos fazer escalabilidade horizontal.**

Usar *Stateless objects* em vez de *Stateful objects*

- **Para tirar partido do balanceamento de carga deve gerir-se o estado dos objetos com algum cuidado;**

Stateless Object

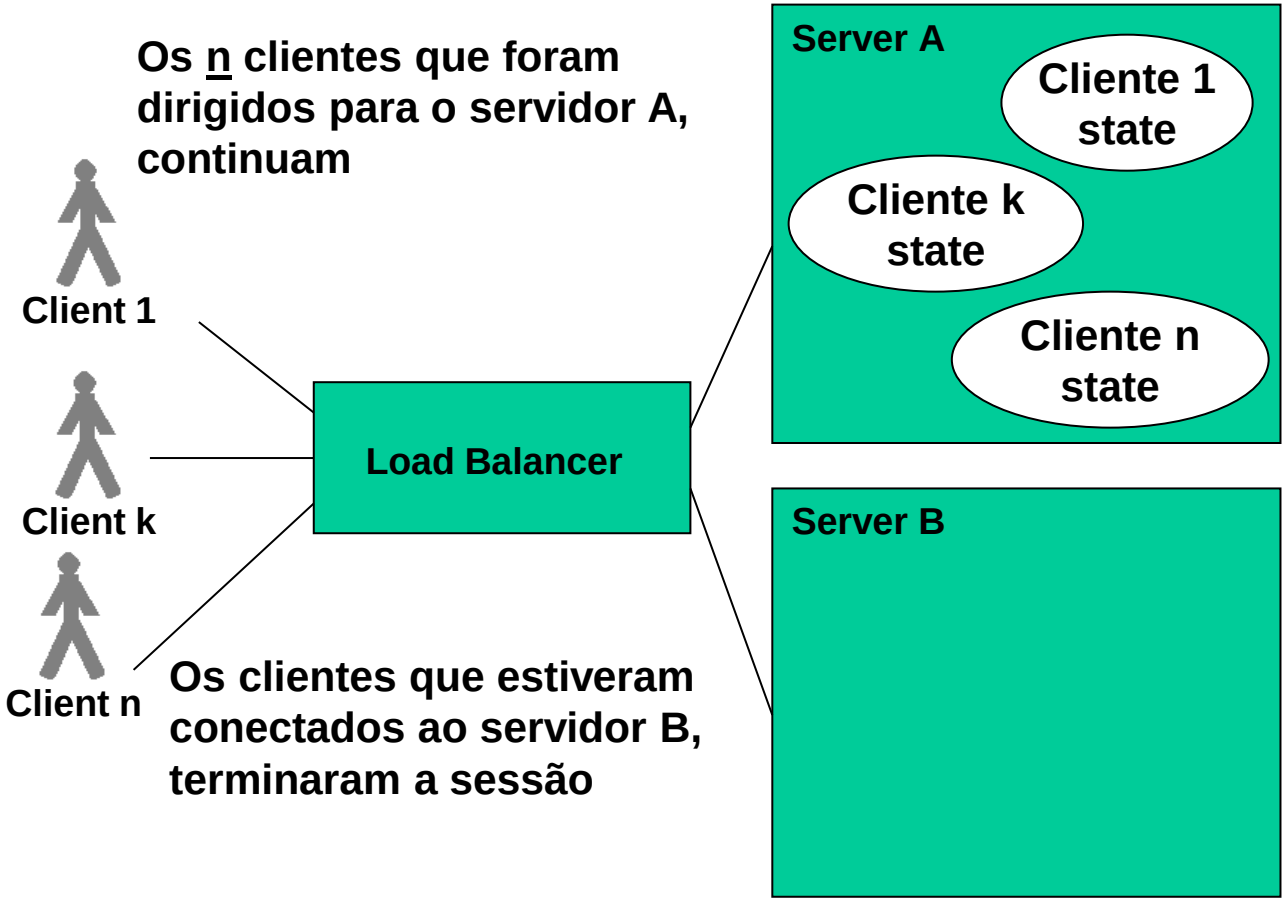
Objeto que pode ser criado e destruído entre chamadas a métodos. Implica implementar classes que não podem utilizar variáveis de instância ou atributos estáticos entre chamadas.

Stateful Object

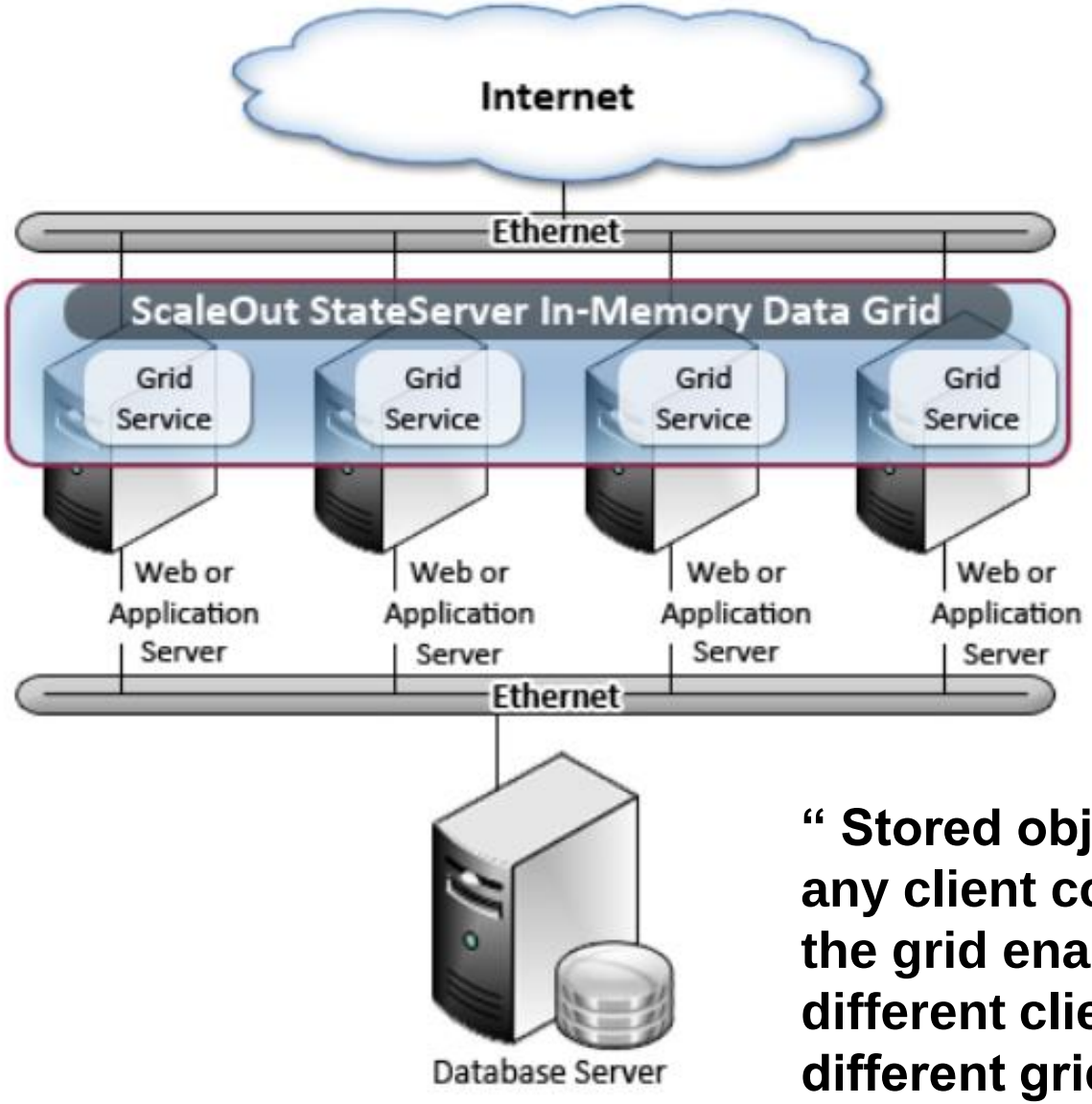
- ✓ **Objeto que mantém estado (valores dos atributos), entre chamadas a métodos;**
- ✓ **Afetam negativamente a escalabilidade por duas razões:**
  1. **O objeto terá de ter um tempo de vida longo, durante o qual pode acumular e consumir recursos escassos, mesmo que esses recursos não estejam a ser utilizados;**
  2. **Minimizam a possibilidade de replicar os objectos por vários servidores por questões de balanceamento de carga ou tolerância a falhas.**

Falta de escalabilidade com objetos *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.



Um exemplo de solução possível

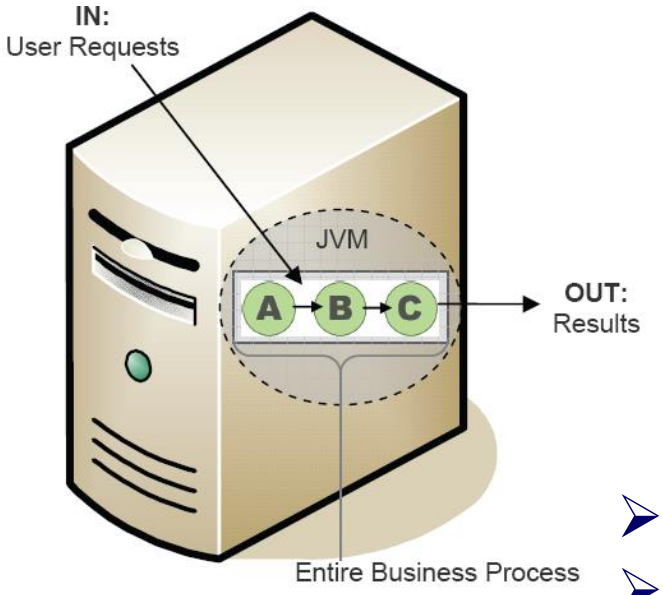


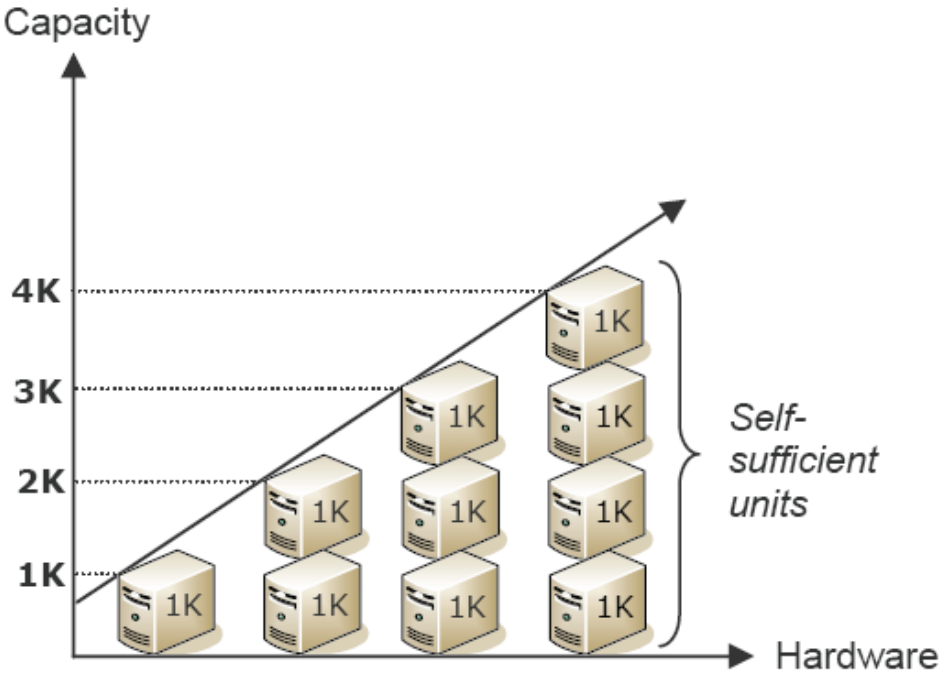
www.scaleoutsoftware.com

“ Stored objects can be accessed from any client connected to the data grid, and the grid enables simultaneous access by different clients to data objects stored on different grid servers “

Escalabilidade Linear com Unidades Autónomas

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





- **Processo de Negócio na mesma Unidade Autónoma;**
- **Implica desenvolver aplicações que implementem processos de negócio completos para um determinado número de Pedidos/Clientes, que são suportados por uma única Unidade Autónoma**

https://docs.gigaspaces.com/latest/overview/the-in-memory-data-grid.html

Elasticidade (*auto scalling*)

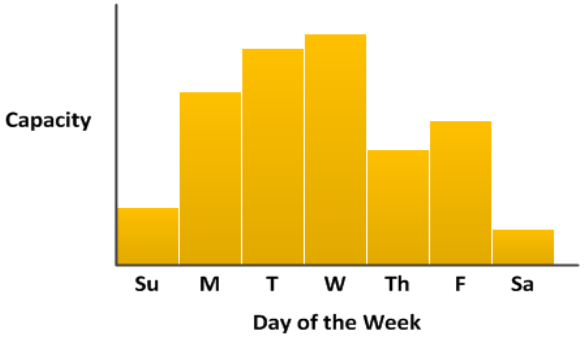
- **Elasticidade, consiste na habilidade de uma infra-estrutura, rapidamente aumentar ou diminuir os recursos mantendo estável o desempenho, a segurança, a gestão e a compatibilidade de protocolos.**

- **A escalabilidade está relacionada com a capacidade de um sistema suportar o aumento da carga de trabalho com aumento de recursos. O conceito de elasticidade permite libertar os recursos quando estes já não são necessários tendo assim um sinónimo de adaptabilidade.**

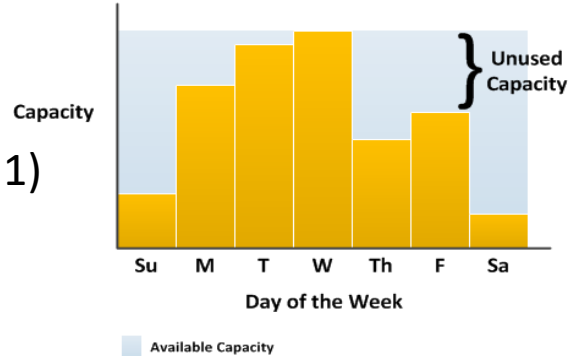
- **Em ambientes Cloud, elasticidade pode também implicar a habilidade de alterar, sem intervenção humana (em real time), os *Service Level Agreements (SLAs)* no sentido que os clientes pagam unicamente os recursos que usam.**

Capacidade versus necessidades

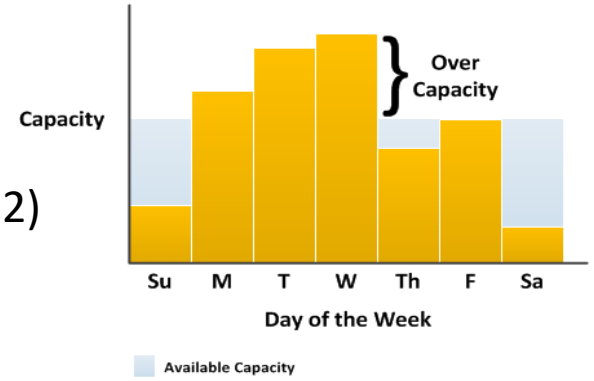
Capacidade necessária ao longo de uma semana



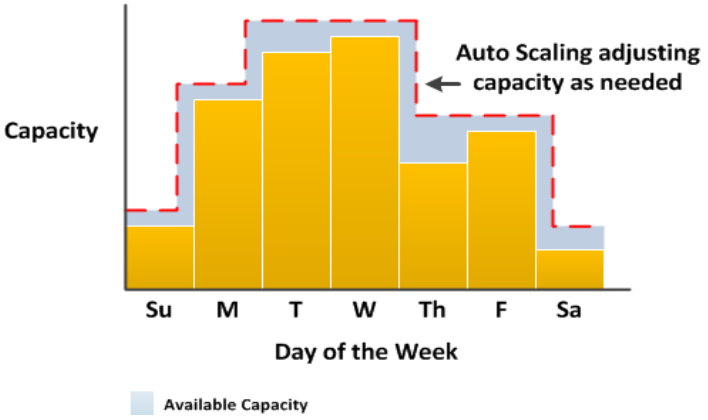
1)



2)



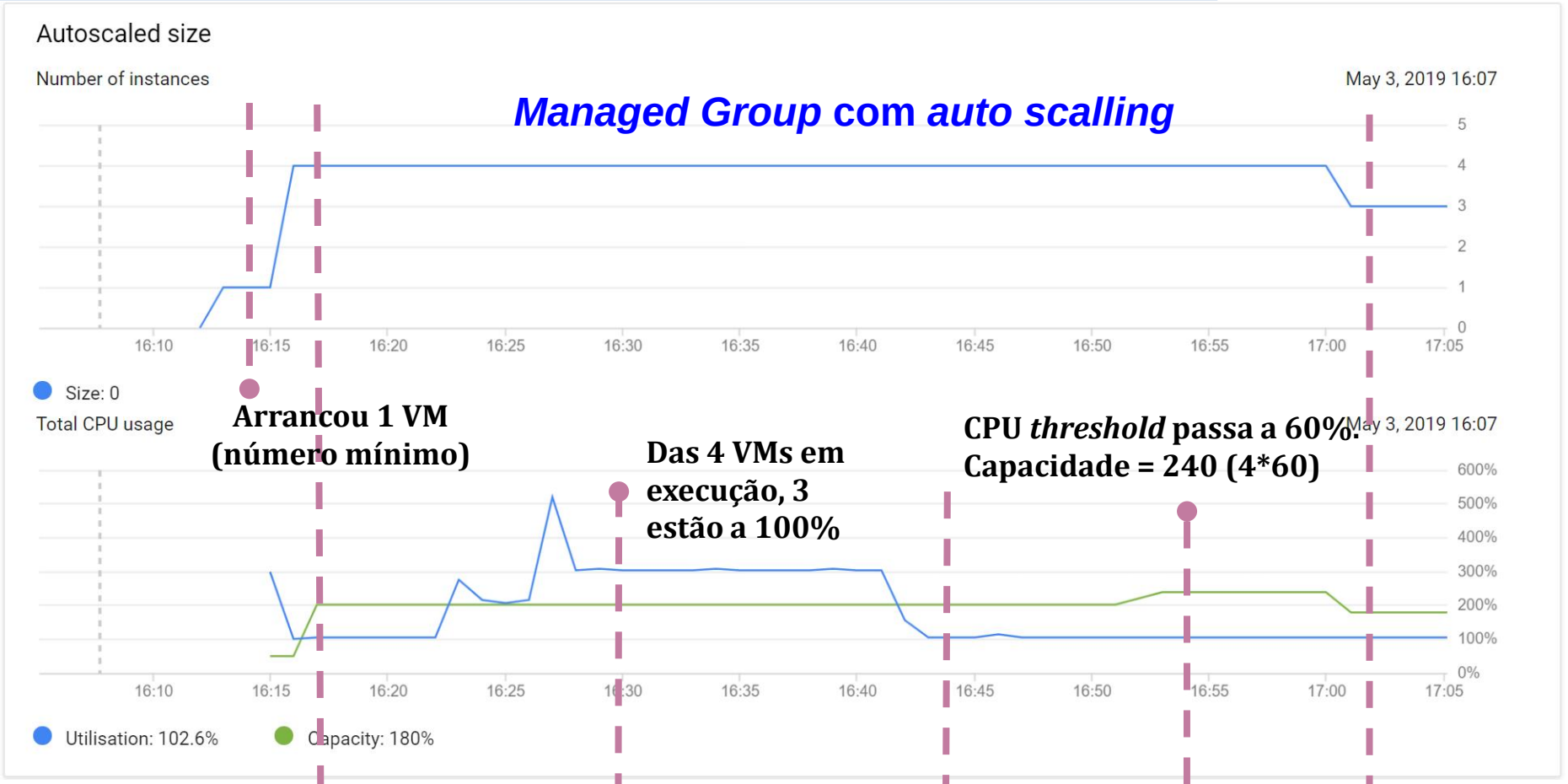
Elasticidade:
Adaptar a capacidade em função das necessidades



https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-benefits.html

- **Expande ou reduz recursos dinamicamente, baseado em condições:**
  - **Aumento de utilização de CPU de VMs;**
  - **Num determinado dia e hora;**
  - **Número de utilizadores concorrentes**

O caso *Google Cloud Platform* - grupos de instâncias de VMs



Arrancou 1 VM (número mínimo)

Das 4 VMs em execução, 3 estão a 100%

CPU *threshold* passa a 60%. Capacidade = 240 (4*60)

A intensa utilização inicial (utilização acima da capacidade) desencadeou a criação de novas instâncias (+3)

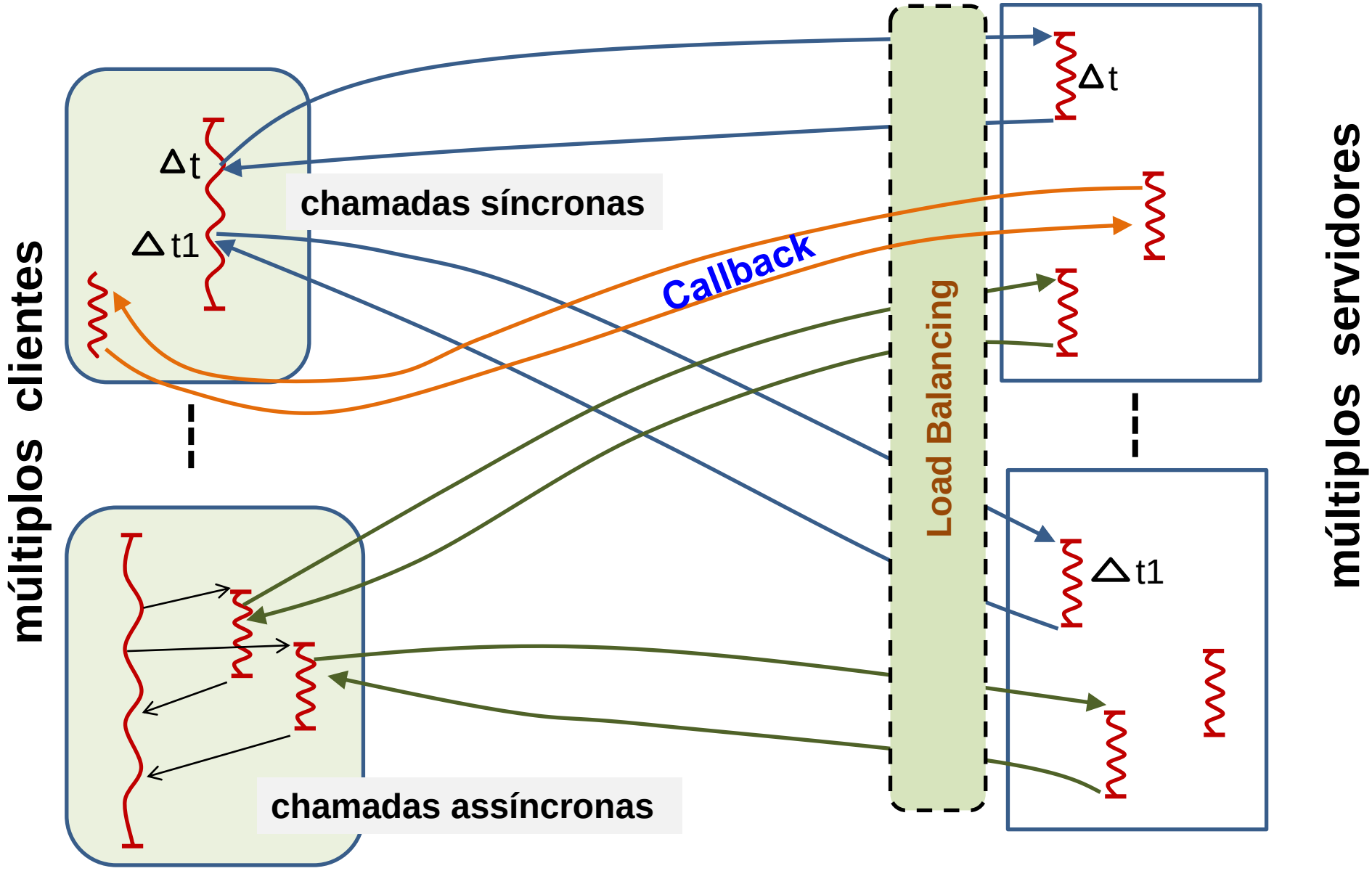
Apenas 1 VM a 100%

Grupo é reduzido para 3 instâncias Capacidade = 180 (3*60)

Concorrência

- **Processos executados em concorrência**
  - **Um processador (core) versus múltiplos processadores (cores)**
- **Processos executados em paralelo**
  - **Estação de trabalho com N processadores (N > 1) - Multi-processador**
  - **Múltiplos servidores (VMs) distribuídos**
- **Servidores concorrentes na resposta a clientes (múltiplas *Threads* para atender os pedidos no servidor)**
  - **Implica Controlo da Concorrência**
- **Mecanismos de sincronização, exclusão mútua, gestão de filas de espera (prioridades), Eleição; Consensos; Chamadas assíncronas, etc.**
- **Transações Distribuídas**

Concorrência a vários níveis



Tolerância a falhas

- As falhas num sistema distribuído são parciais, isto é, alguns componentes falham, enquanto outras continuam em função. Detetar e tratar falhas parciais é extremamente difícil;

- Técnicas para lidar com falhas:
  - Redundância do hardware (*clustering de servidores*, RAID (*Redundant Array of Independent Disks)*, etc.)
  - Recuperação de erros por parte do software: retransmissão de mensagens e replicação de dados em diferentes localizações;
  - Um utilizador deverá poder continuar a sua atividade numa estação de trabalho alternativa (por exemplo usando técnicas de *Virtual Desktop*);
  - Replicação de recursos e servidores;
  - Reposição de estados anteriores após uma falha (*rolled back*);
  - Grupos de serviços replicados na implementação de aplicações;
  - A disponibilidade (*Availability*) é normalmente maior num sistema distribuído, com maior facilidade em disponibilizar recursos alternativos, potenciado pelas tecnologias de virtualização

Falhas Bizantinas

- **Falhas arbitrárias usadas para descrever os piores cenários de falhas, onde diferentes observadores observam diferentes comportamentos num sistema**

  "Byzantine generals problem"

  Consideremos unicamente 3 generais A, B e C e em que A é traidor;
  - ✓ **A diz a B → *Atacar***
  - ✓ **A diz a C → *Retirar***
  - ✓ **B diz a C → O general A mandou *Atacar***

  C não pode concluir quem é traidor. Se A ou B ?

  ***Lamport, Shostack, and Peace*, provaram [1] que o problema só tem solução se,**

  $$N \geq 3t + 1$$

  N – número de generais
  t – número de generais traidores.

  isto é, na presença de falhas é necessário recorrer a algoritmos distribuídos de geração de consensos por maioria.

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

Falhas Bizantinas e os sistemas distribuídos

- **Um sistema é tolerante a falhas bizantinas, se continua a funcionar mesmo quando algumas partes do sistema não funcionam ou não obedecem a protocolos estabelecidos;**
  - **Ambientes aeroespaciais, sistemas de controlo de tráfego aéreo;**
  - **Sistemas colaborativos (múltiplas organizações), onde não exista uma autoridade central: *Bitcoin*, *blockchain*;**

- **Na prática, devido à complexidade, não é comum suportar falhas bizantinas. Recorre-se a técnicas de ter alguma autoridade que uma vez eleita decide se alguma ação pode ou não prosseguir;**

- **Os algoritmos de suporte a falhas bizantinas requerem normalmente consensos de maioria ou mesmo de 2/3 dos participantes:**
  - **Usando uma aproximação do tipo "generais bizantinos" para detetar bugs de software, requeria 4 implementações diferentes e garantir que apenas uma tivesse bugs!**

Desenvolver para uma Interface e não para uma implementação

interface ≡ contrato

- **O código dos serviços 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);**
- **Veremos em Google RPC que as interface de um serviço é descrita na linguagem de script *Protobuf*;**
- **A definição de interfaces, para além de esconder detalhes de implementação, deverá facilitar a interoperabilidade entre sistemas heterogéneos e permitir que seja possível invocar serviços remotos conhecendo unicamente a interface.**

Usar Interfaces com granularidade compacta em vez de fina

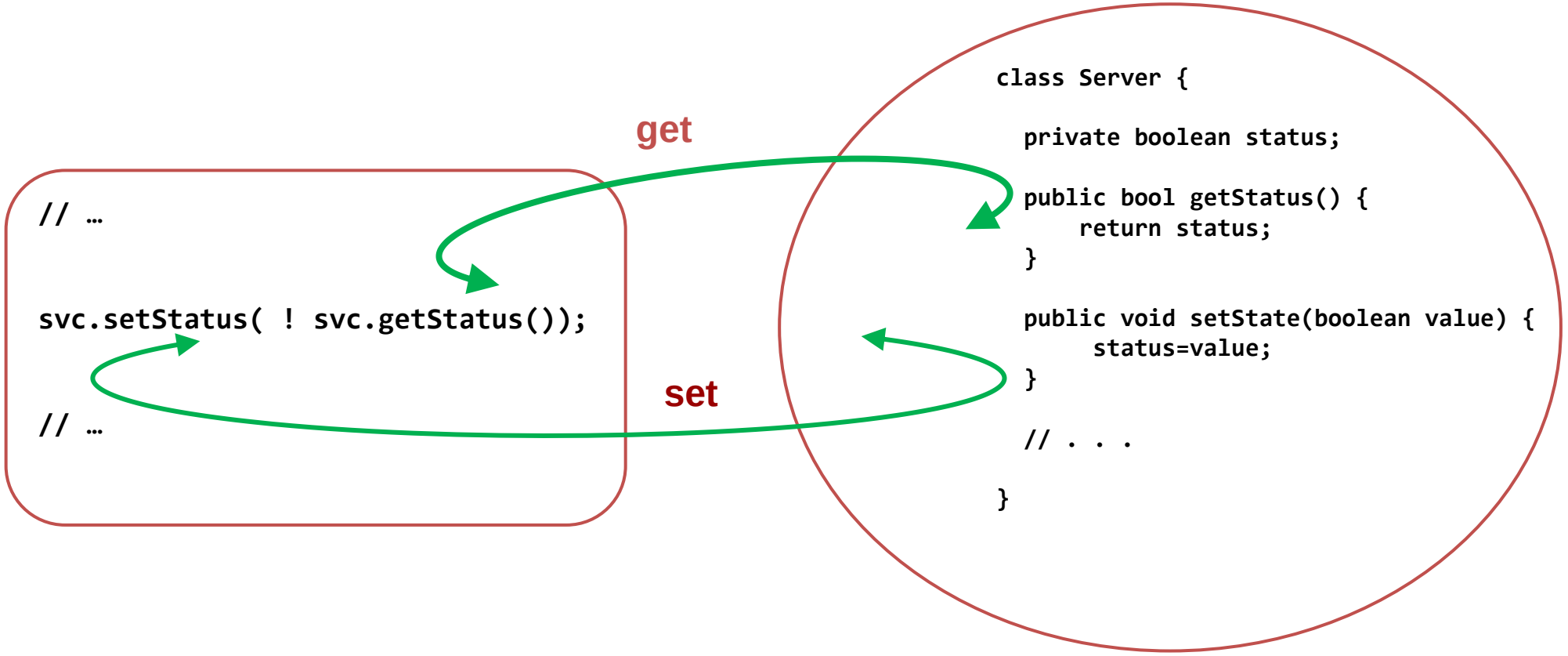
```
public class Cliente {
      public string FirstName;
      public string LastName;
      public string Email;
      // ... outros atributos

      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` estiver instanciado noutra, qualquer alteração de estado do objecto (atributo ou chamada a métodos) implica comunicação, o que se poderá traduzir em sérios problemas de desempenho

Um caso inocente !

Uma simples linha de código pode envolver vários acessos remotos



Padrão: *Data Transfer Objects* - DTO

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

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.

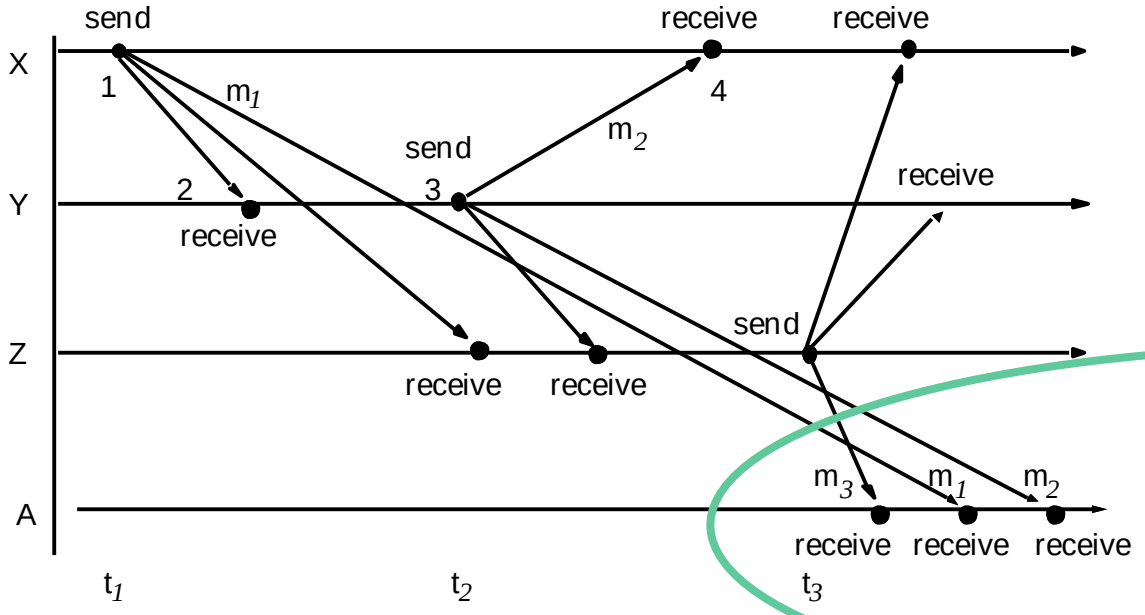
Classe serializable

```
[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) {}
}
```

```
import java.io.Serializable;

public class DadosCliente implements Serializable
{
   //. . .
}
```

Coordenação do trabalho em cada parte e coordenação global

A coordenação entre as partes, a existência de estados globais e a ordenação de eventos levantam grandes desafios e complexidade na programação de sistemas distribuídos;



- **m1- mensagem de convocação de uma reunião**
- **m2 e m3 - respostas.**

- **Resposta de Z para uma reunião ?**
- **Marcação de uma reunião por X**
- **Resposta de Y para a reunião**

© Extraído de B1

Necessidade de capturar dependências causais

Eficiência e desafios das comunicações entre as partes

- ✓ **A comunicação entre as partes introduz a falta de eficiência e desempenho nos sistemas distribuídos, embora a evolução dos últimos anos (larguras de banda cada vez maiores) tenha atenuado o problema**
- ✓ **A comunicação de dados tem sempre associada uma latência**
- ✓ **A comunicação apresenta desafios de segurança**
- ✓ **A existência de comunicações seguras, introduz *overhead*.**
- ✓ **O problema de endereçamentos *Network Address Translation* (NAT), também conhecido como *masquerading*, introduz dificuldades de conectividade**

Replicação de dados e serviços

- **A replicação distribuída em múltiplos nós (computadores ou VMs) tem vantagens:**
  - ✓ **Aumento de desempenho em termos de *throughput***
  - ✓ **Diminuir a latência ao ter os dados geograficamente perto dos utilizadores**
  - ✓ **Aumento da tolerância a falhas**

- **Mas a replicação introduz a necessidade de resolver os seguintes desafios:**
  - ✓ **Consistência (modelos de consistência eventual)**
  - ✓ **Tratamento dos conflitos na escrita de dados. Um ou mais *leaders* reencaminham as escritas para os seguidores (*followers*)**
  - ✓ **Coordenação, sincronização, eleição e consenso (*agreement*) sobre a ordem das ações envolvidas**

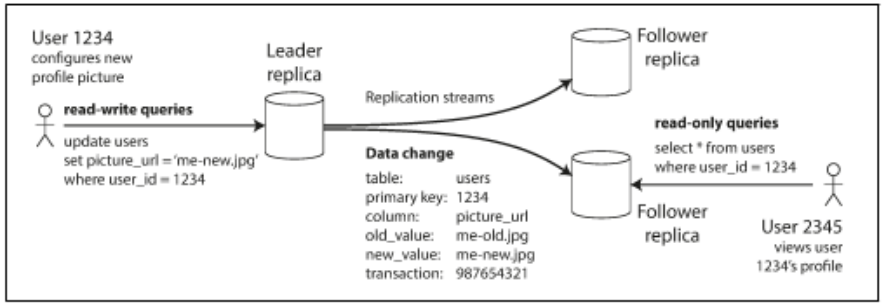


Figure 5-1. Leader-based (master–slave) replication.

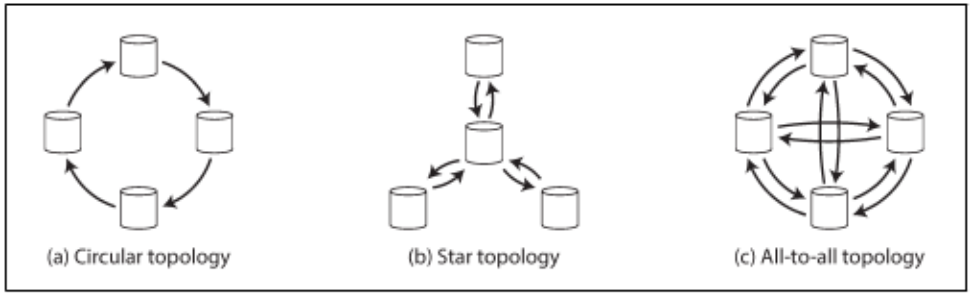
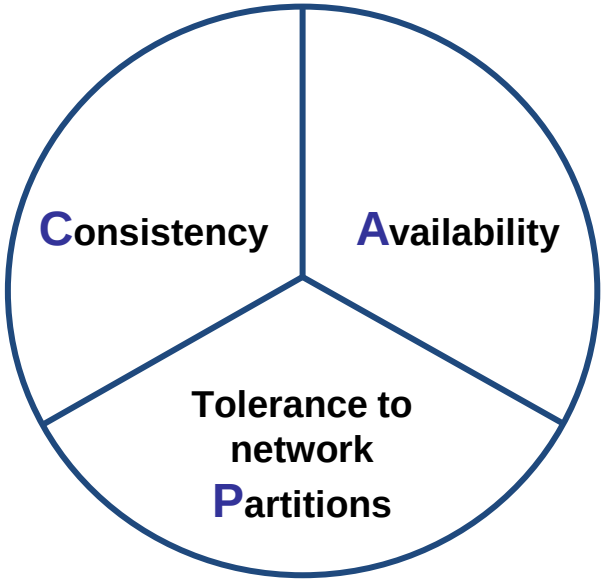


Figure 5-8. Three example topologies in which multi-leader replication can be set up.

***From: [B3] Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, Martin Kleppmann, 2017**

CAP Theorem



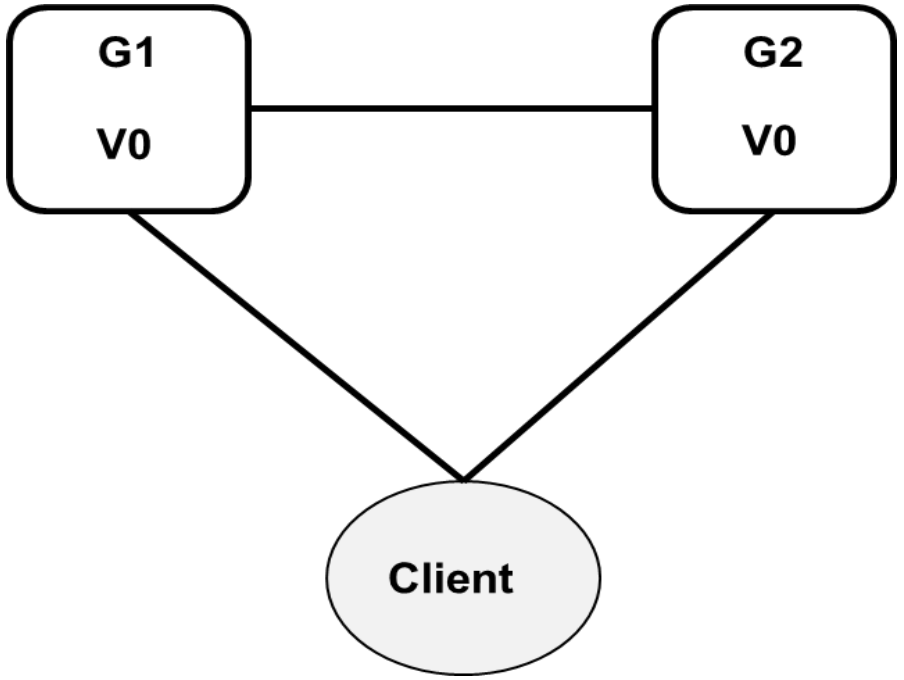
"Num sistema distribuído que partilhe dados com réplicas só podemos ter 2 das 3 propriedades"

Eric A. Brewer - UC Berkeley, 2000

- ***Consistency*** **– Cada leitura observa a última escrita. O sistema oferece um estado consistente para todos os observadores.**
- ***Availability*** **– O sistema continua a funcionar (eventual degradação na qualidade de serviço) na presença de falhas ou de falta de conectividade de alguns nós.**
- ***Partitions Tolerance*** **- O sistema continua a funcionar apesar de haver atraso ou falha na entrega de mensagens.**

Teorema CAP num sistema distribuído

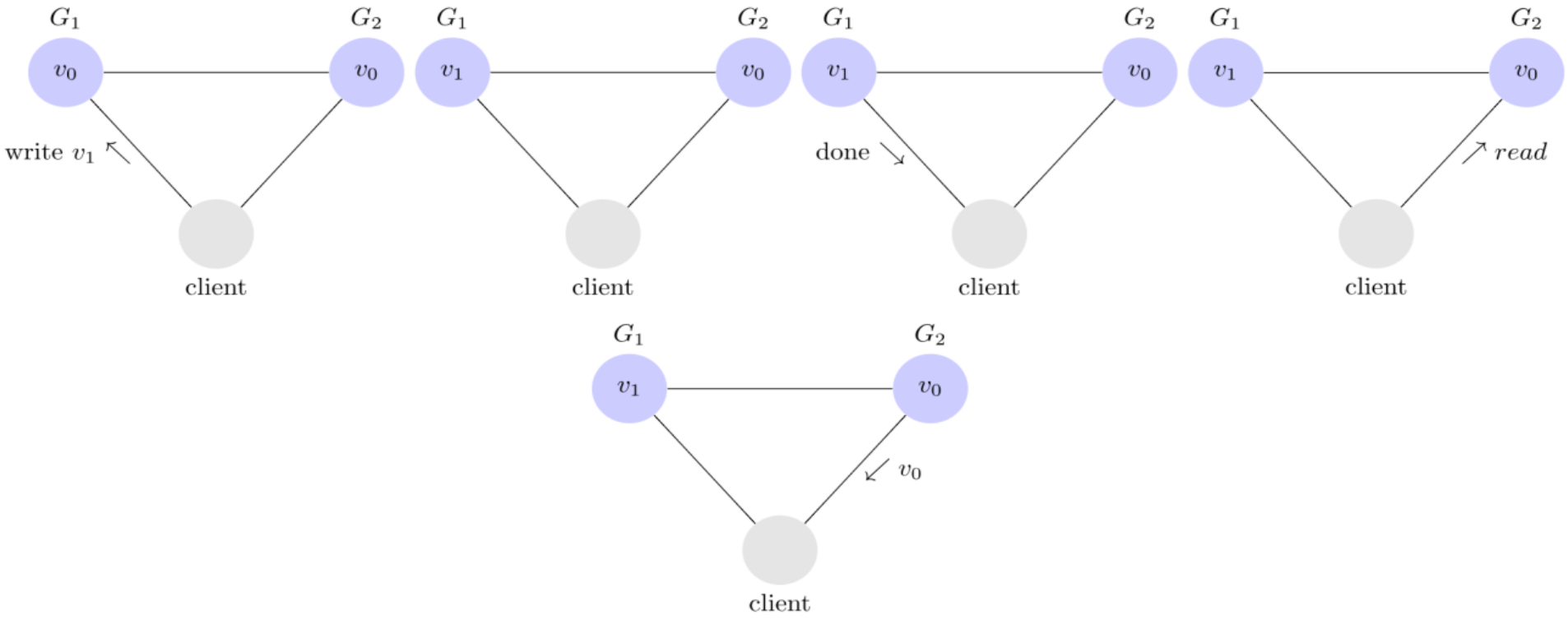
- **Um sistema com dois servidores G1 e G2 que comunicam entre si**
- **O cliente pode comunicar com os dois servidores**
- **Os servidores têm um mesmo objeto com o valor V0**



* Baseado em: *https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/*

Consistência

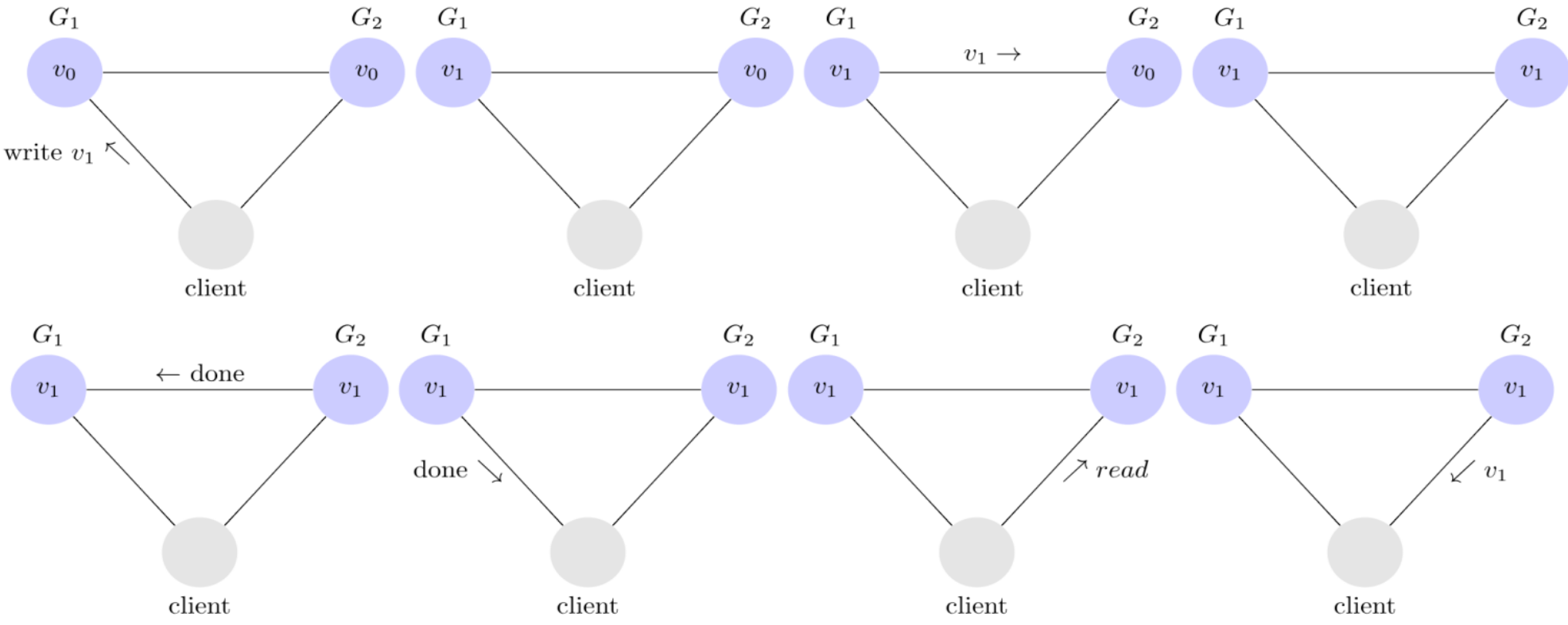
- **Uma operação READ devolve sempre o valor da última operação WRITE**



Inconsistente

Consistência

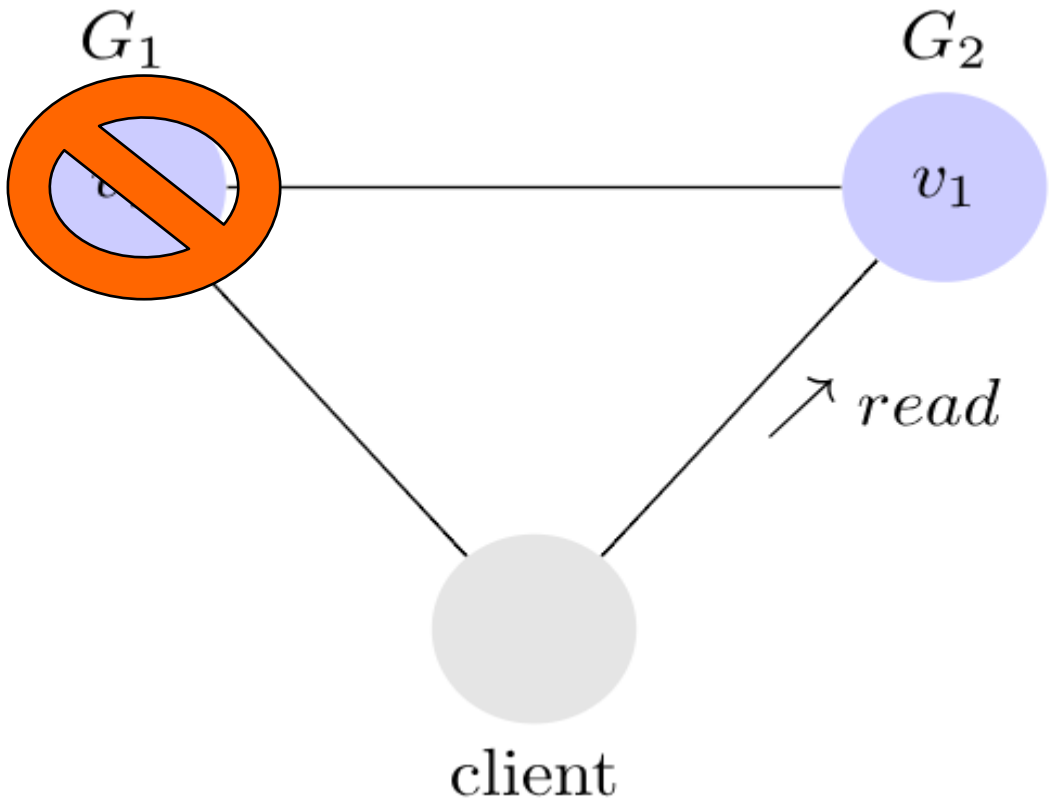
- **Uma operação READ devolve sempre o valor da última operação WRITE**



Consistente

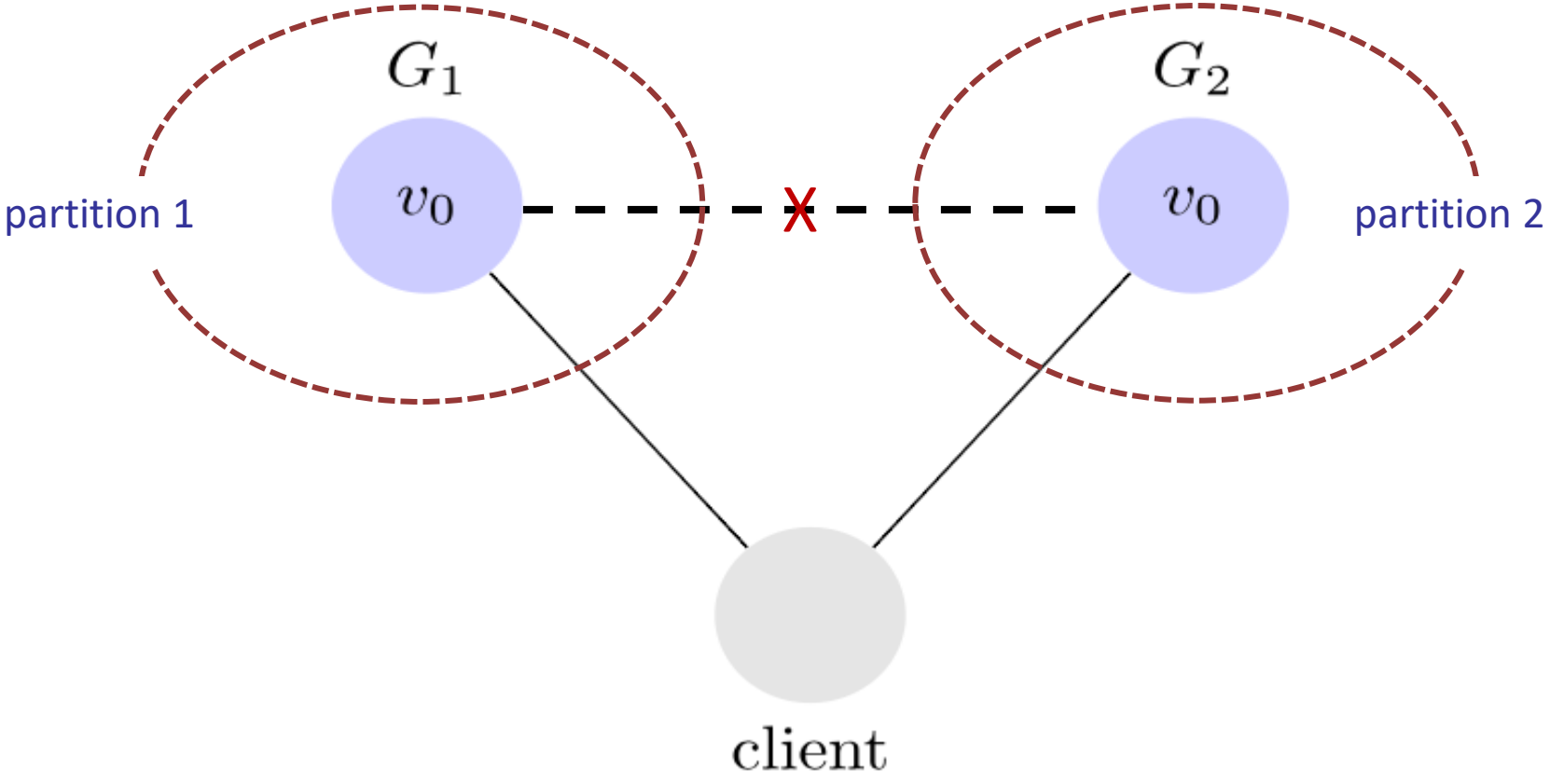
Disponibilidade

- **Todos os pedidos têm uma resposta, isto é um dos servidores disponíveis deve responder aos pedidos do cliente**



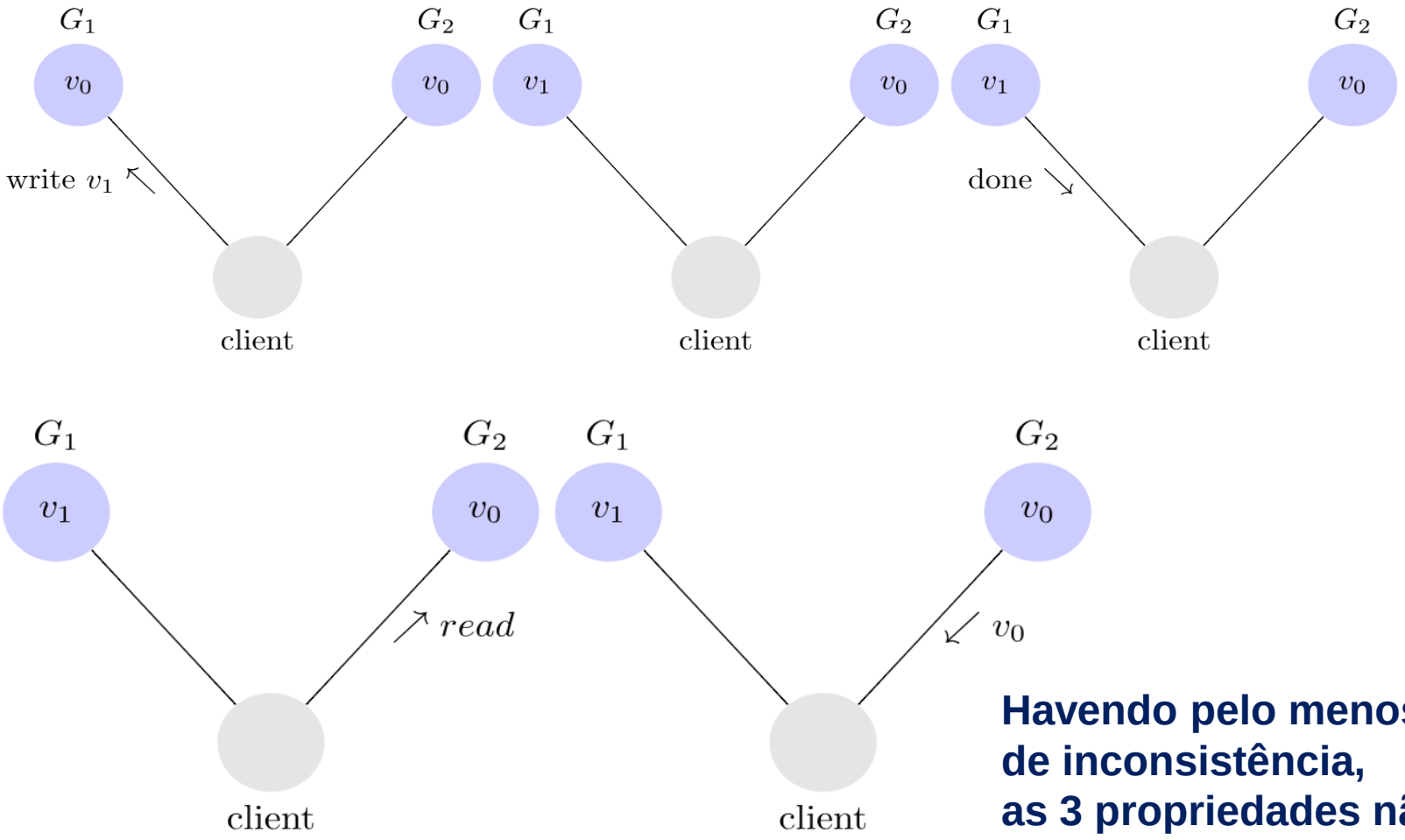
Tolerância à partição

- **Tolerância à perda arbitrária de mensagens enviadas entre os nós do sistema (2 servidores)**



“Prova” do CAP

- **Por absurdo, vamos admitir que um sistema tem as 3 propriedades**



Havendo pelo menos 1 cenário de inconsistência, as 3 propriedades não podem ser *sempre* garantidas

Teorema CAP e o desenho de aplicações

- **Os arquitetos de aplicações distribuídas, necessitam definir um compromisso maximizando combinações de consistência (*Consistency*) e disponibilidade (*Availability*), assumindo que existem partições (*Partitions*) e técnicas para lidar com elas;**

- **Por exemplo, os sistemas de armazenamento de dados NoSQL, baseiam-se em técnicas adequadas para trabalhar com múltiplas partições (centenas ou milhares de computadores ligados em rede), privilegiando a disponibilidade**

  - **Semântica BASE (*Basically Available, Soft state, Eventually consistent*).**

Transparência (i)

- **Ao acesso: O acesso a recursos locais e remotos usa operações idênticas. Esconde diferenças entre representação de dados e de convenções de nomeação de recursos (ex: acesso a ficheiros);**

- **À localização: Permite que recursos em diferentes locais, sejam acedidos do mesmo modo sem conhecimento da sua localização, por exemplo endereços IPs. A nomeação de recursos (por exemplo, URLs com nomes de domínio) tem um papel importante na transparência à localização;**

- **À migração: Permite a deslocação de recursos ou processos sem afetar o modo de acesso aos mesmos;**

- **À concorrência: Permite que múltiplos processos concorrentes, usem recursos partilhados, sem interferência, isto é, múltiplos acessos concorrentes devem deixar o recurso num estado consistente. Por exemplo, um objecto/serviço servidor pode processar vários pedidos de vários clientes em concorrência.**

Transparência (ii)

- **Às réplicas: Uso de múltiplas instâncias (réplicas) de um recurso para melhorar o desempenho, segurança e a tolerância a falhas, sem conhecimento por parte do utilizador ou programador da existência de réplicas. A transparência às réplicas implica transparência à localização, isto é todas as réplicas devem ter o mesmo nome. Exemplos: Bases de Dados distribuídas; Mirroring Web Sites**

- **Às falhas: Disfarce de falhas, permitindo a conclusão de tarefas perante falhas de hardware ou de componentes de software. Mascarar falhas é um dos aspetos mais complexo, ou mesmo impossível, em sistemas distribuídos;**

- **Ao desempenho: Permite a reconfiguração de um sistema de modo a acompanhar as variações de carga. Por exemplo, Serviços na Cloud de *Auto-scaling*;**

- **À escalabilidade: Permite a escalabilidade do sistema sem alteração da estrutura do sistema nem dos algoritmos das aplicações**

Existem conflitos (*trade-off*) entre altos níveis de transparência e o desempenho

Segurança

- **Privacidade - (confidencialidade) - protecção contra acessos não autorizados;**
- **Integridade - Protecção contra alterações e corrupção dos dados;**
- **Autenticação – Garantia de que os interlocutores são quem dizem ser;**
- **Autorização – Os utilizadores autenticados podem ter permissões diferentes para as diferentes actividades;**
- **Disponibilidade – Protecção contra interferências no acesso ao serviço;**

- **Os desafios:**
  - **Protecção contra ataques de negação do serviço (*denial of service*), por exemplo, "bombardeamento" de um serviço com pedidos, impossibilitando outros utilizadores de o usar;**
  - **Segurança de código móvel;**
  - **Roubo de identidades;**

Qualidade de Serviço

- **Garantir que os utilizadores têm acesso às funcionalidades de um serviço dentro de limites definidos para determinados indicadores, por exemplo, o tempo de transmissão na distribuição de conteúdos multimédia;**
- **As principais características, não funcionais, envolvidas que afectam a QoS são a fiabilidade (reliability), segurança, desempenho e a capacidade de reconfigurar o sistema (adaptability).**
- **A disponibilidade em tempo apropriado, dos recursos computacionais e de comunicação necessários;**

❖ **Na Cloud a qualidade de serviço (QoS) é importante:**

- **Tempos de execução e de resposta;**
- **Variações de carga (*workloads*) versus alocação e atribuição dinâmica de recursos, por exemplo VMs;**
- **Possibilidade de monitorizar e definir restrições que garantam os níveis de serviços SLA (*Service Level Agreements*) adequados**

As 8 falácias dos sistemas distribuídos

1. **A rede é confiável**
   - **as aplicações precisam de tratar erros e repetir chamadas**
2. **A latência é zero**
   - **as aplicações precisam de minimizar pedidos**
3. **A largura de banda é infinita**
   - **as aplicações devem usar o menor *payload* possível para o problema**
4. **A rede é segura**
   - **As aplicações têm de garantir segurança na comunicação ponto a ponto**
5. **A topologia não muda**
   - **Mudanças afetam latência, largura de banda e destinos**
6. **Existe um administrador**
   - **Interação com diferentes sistemas e políticas de gestão**
7. **Custo de transporte é zero**
   - **Na Cloud existe muitas vezes um custo monetário para mover dados**
8. **A rede é homogénea**
   - **Afeta a fiabilidade, latência e largura de banda**

A ler: https://www.simpleorientedarchitecture.com/8-fallacies-of-distributed-systems/

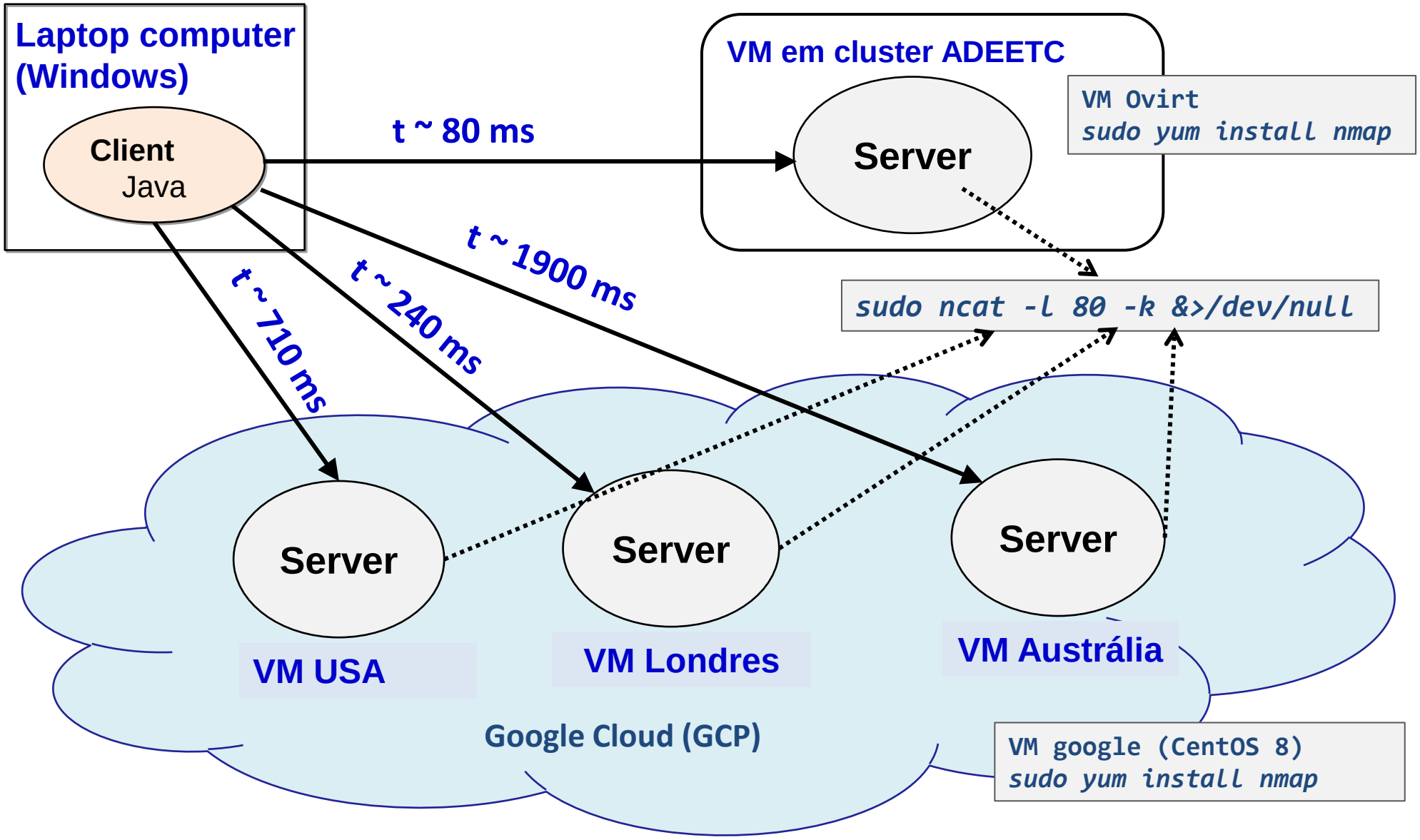
Latência e largura de banda

- ***Latency*** **– Tempo para transferir dados de um ponto para outro.**
- ***Bandwidth*** **– Quantidade de dados que podemos transferir num determinado tempo.**

Latency vs. Bandwidth – Developers vs. Einstein by *Ingo Rammer*

"But I think that it's really interesting to see that the end-to-end bandwidth increased by 1468 times within the last 11 years while the latency (the time a single ping takes) has only been improved tenfold. If this wouldn't be enough, there is even a natural cap on latency. ***The minimum round-trip time between two points of this earth is determined by the maximum speed of information transmission: the speed of light. At roughly 300,000 kilometers per second*** *(3.6 * 10E12 teraangstrom per fortnight),* ***it will always take at least 30 milliseconds to send a ping from Europe to the US and back****, even if the processing would be done in real time."*

Exemplo: Latência & Cloud - *File Upload* (1 MB)



Aspetos fundamentais a reter

- **No desenvolvimento de sistemas distribuídos deve considerar-se:**
  - **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 escalabilidade;**
  - **Definir a necessidade e modelo de Segurança, isto é, identificar o inimigo (permissões de acesso e canais de comunicação seguros);**
  - **Qualidade do serviço (QoS)**
  - **Não esquecer as 8 falácias, em todas as decisões de arquitetura**