

# Consistência e Replicação



*"I thought cloning myself would really save me time around the house,  
but it turns out he's just as lazy as I am."*

# Razões para replicar

- Aumentar a fiabilidade do Sistema
  - Continuidade de operação após uma falha
  - Proteção contra corrupção dos dados
- Performance
  - Cobrir áreas geográficas distantes → e.g.: Netflix tem servidores espalhados pelo Mundo!
  - Atender números elevados de solicitações
- Porque não replicar? ☺*Depende do Projeto!*
  - Eventual falta de consistência entre replicas

As coisas têm de continuar mesmo quando um nó falha!

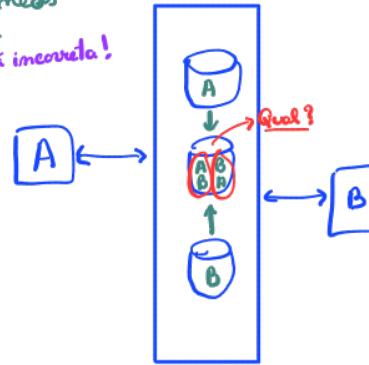
→ Base de dados distribuída resolue! (processos de coordenação resolvem a corrupção)

Será que queremos replicar?  
→ RabbitMQ é fácil  
→ Kafka é difícil, mas...

# Performance e Escalabilidade

## Problemas:

- Escritos simultâneos
- Leitura e Escrita  
Leitura que depois está incorreta!



- Se A e B escreverem em BD's diferentes, pode não ser escrito pela ordem correta ...
- Duas escritas ao mesmo tempo podem dar erro, precisamos de Exclusão Mútua.

Sincronização com réplicas

- Para manter consistência entre réplicas, é geralmente necessário garantir que as operações conflituantes são feitas pela mesma ordem.

Read-Read e Write-Read não geram problemas!

- Operações que geram conflitos:

- Read-write: operação de leitura e escrita concorrentes
- Write-write: duas operações de escrita

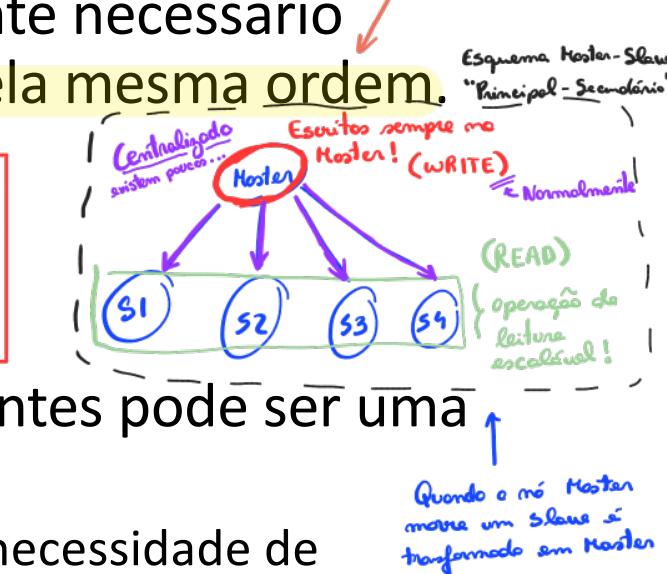
- Garantir uma ordem global em operações conflituantes pode ser uma operação onerosa, o que baixa a escalabilidade.
  - Solução: diminuir requisitos de consistência para que a necessidade de sincronismo global seja evitada.

Ou Coordenação!

→ Diferentes versões têm impacto em diferentes requisitos

Problema: 2 falhos em simultâneo

• Não existem soluções perfeitas! (Depende do Caso de Uso)



## Mecanismos de consistência:

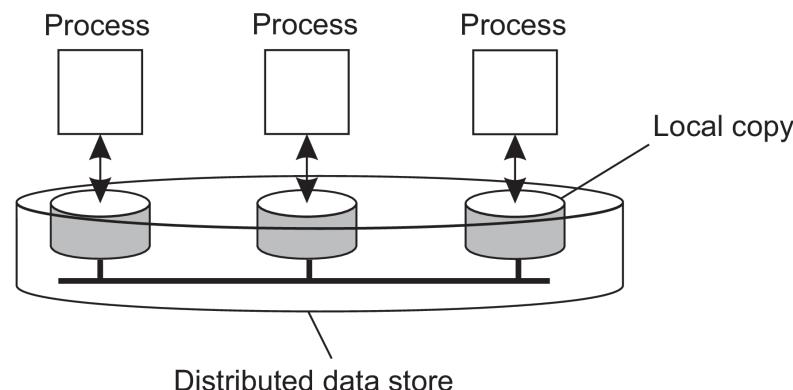
centrado nos dados  
armazenados!  
↓

# Modelos de consistência Data-centric

### • Modelo de consistência

- Um contrato entre uma data store (distribuída) e processos, em que a data store especifica precisamente qual o resultado de uma operação de leitura e escrita em concorrência.

→ Read-Write, mas os processos sabem { Como eles resolvem  
qual a ordem usada por todos a concorrência  
eles... internamente / Desempate!



# Consistência contínua

Depende SEMPRE  
do caso de uso!

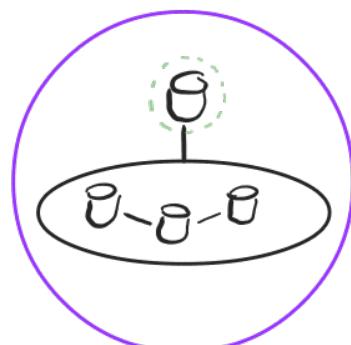
- Podemos falar do grau de consistência:

- Replicas podem diferir no seu valor numérico { ver se um campo tem valor 10 → e o outro na outra DB é 12  
*Nos nem tudo é número!* (Distância entre strings)}
- Replicas podem diferir na sua imutabilidade relativa { Pode não ser a cópia exata do Master... se for uma app bem distribuída sincronizam-se periodicamente (as bases de dados são distribuídos por modelo de negócio)}
- Podem existir diferenças no número e ordem de operações de actualização (update) { Agora na sincronização:  
 $A:0 \rightarrow 1$   
 $A:1 \rightarrow 2$   
ou  
 $A:0 \rightarrow 1$   
 $A:1 \rightarrow 2$   
*Pode ignorar a ordem e o número de operações*

- Conit

- Unidade de consistência => especifica a **unidade de dados** sobre a qual deve ser medida a consistência

Unidade de  
consistência



As BD normalmente analisam  
a informação interna (não sabe  
concretamente!)

text1  
text2

Quanto diferem?

→ Uma réplica deveria ser  
igual ao master...  
→ Mas podemos aceitar coisas...



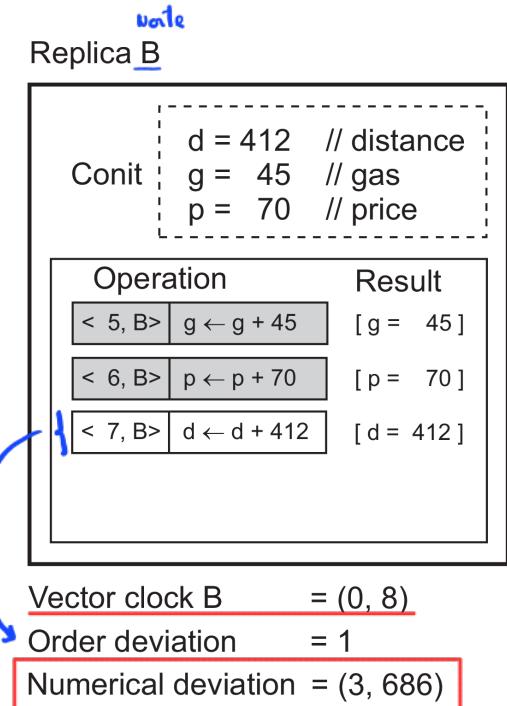
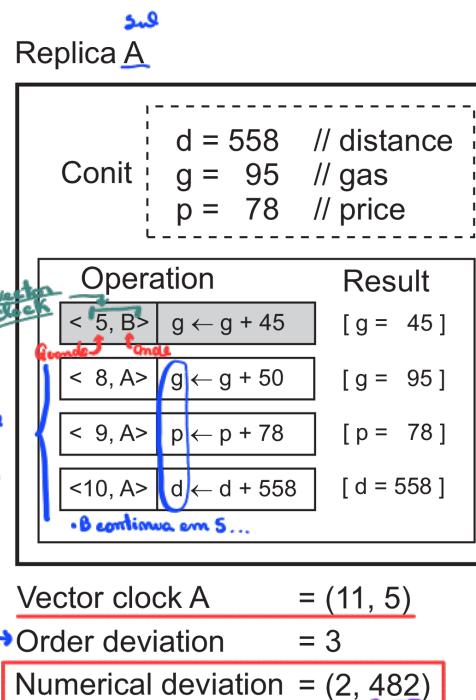
## Exemplo: Conit

- Cada réplica tem um vector clock
- B envia a A operação a cinzento
  - A executou permanentemente essa operação
- Conit:
  - A tem 3 operações pendentes:
    - Desvio de **ordem** = 3
  - A perdeu 2 operações de B:
    - Max diff é  $70 + 412 = (2, 482)$
    - Desvio **numérico** de 482

Depende do nosso compromisso!

$(\dots, \dots)$   
n-dimension  
(n més)

Desvio de  
ordem de 3  
operações



! Somente TUDO, som  
saber o que é...

• Queremos saber o desvio de dados!

### MEDIDA da DIFERENÇA NUMÉRICA

• Mas se fizessem  $\oplus \ominus$  o Conit podia  
se monitorar...

• Liso o Conit dirige:

! Porém a escrita! E fazer sincronização!

LOOK! fazer até um threshold  
de Conit e voltar... Não  
precisamos de 100%

# Consistência Sequencial

*Não podemos ter causalidade! //*

- Definição

- O resultado de qualquer execução é o mesmo se todos operações em todos processos forem executadas na mesma ordem sequencial, e as operações de cada processo individual aparecerem na sequência na ordem especificada pelo programa.

- (a) Armazenamento consistente sequencialmente.

- (b) Armazenamento não consistente sequencialmente

a)

P1:	W(x)a
P2:	W(x)b
P3:	R(x)b
P4:	R(x)b R(x)a

*Não é a ordem! É a sequência!*

↓

P1:	W(x)a
P2:	W(x)b
P3:	R(x)b
P4:	R(x)a R(x)b

*Hos, foram executados  
pela mesma ordem!*

*Não me interessa a ordem,  
quero apenas que seja  
IGUAL*

*Foram executados  
por ordens diferentes!*

*Precisamos de um relógio lógico!*

# Consistência causal

Se houverem dependências  $a \rightarrow b$ ,  
por exemplo incrementos!

- Escritas que estão relacionadas casuisticamente, têm que ser vistas por todos processos com a mesma ordem. Escritas concorrentes podem ser vistas com ordens diferentes por diferentes processos.
- (a) Violão<sup>v</sup> de armazenamento casuístico. (b) Sequência correcta de eventos num armazenamento casuístico.

a)

P1:	W(x)a
P2:	R(x)a $\xrightarrow{a \rightarrow b}$ W(x)b
P3:	
P4:	

Efeito de causa violado

$a \rightarrow b$   
não posso ler a  
depois de b ...

b)

P1:	W(x)a				
P2:	<del>W(x)b</del> <i>Não hoce!</i>				
P3:					
P4:	<table border="1"><tbody><tr><td>R(x)b</td><td>R(x)a</td></tr><tr><td>R(x)a</td><td>R(x)b</td></tr></tbody></table> <i>Mensagem chegou fora de ordem!</i>	R(x)b	R(x)a	R(x)a	R(x)b
R(x)b	R(x)a				
R(x)a	R(x)b				

*Não existe causa!*

# Agrupar Operações

Consistência sequencial: ordem igual em todos os processos!

Consistência causal: se uma escrita foi causal então essa ordem deve prevalecer!

- Definição:

O direito a lock é sequencial

- Acesso a **locks** são consistentes sequencialmente
- Não são permitidos acessos a locks, antes que todas escritas prévias tenham sido completadas ?
- Nenhum acesso a dados é permitido até que todos acessos anteriores a **locks** tenham sido feitos.

- Ideia base

- Não interessa que as leituras e escritas de uma **série** de operações seja imediatamente conhecida por outros processos. Apenas queremos que o **efeito** da serie seja conhecido.

⇒ Consistência sequencial!

Não preciso da ordem, apenas do efeito final !!

## Agrupar Operações (2)

Pouca performance! //

P1: L(x) W(x)a L(y) W(y)b U(x) U(y)

---

Sistema de Consequência  
causal!

P2: L(x) R(x)a R(y) NIL

---

↓  
Não tem lock!  
dá NULL...

P3: L(y) R(y)b



Data-centric {  
• Consistência sequencial: a mesma seq. em todos  
• Consistência causal: a causa deve ser considerada

≠

(E)!

Data-Centric

# Modelos Client centric - Consistência Eventual

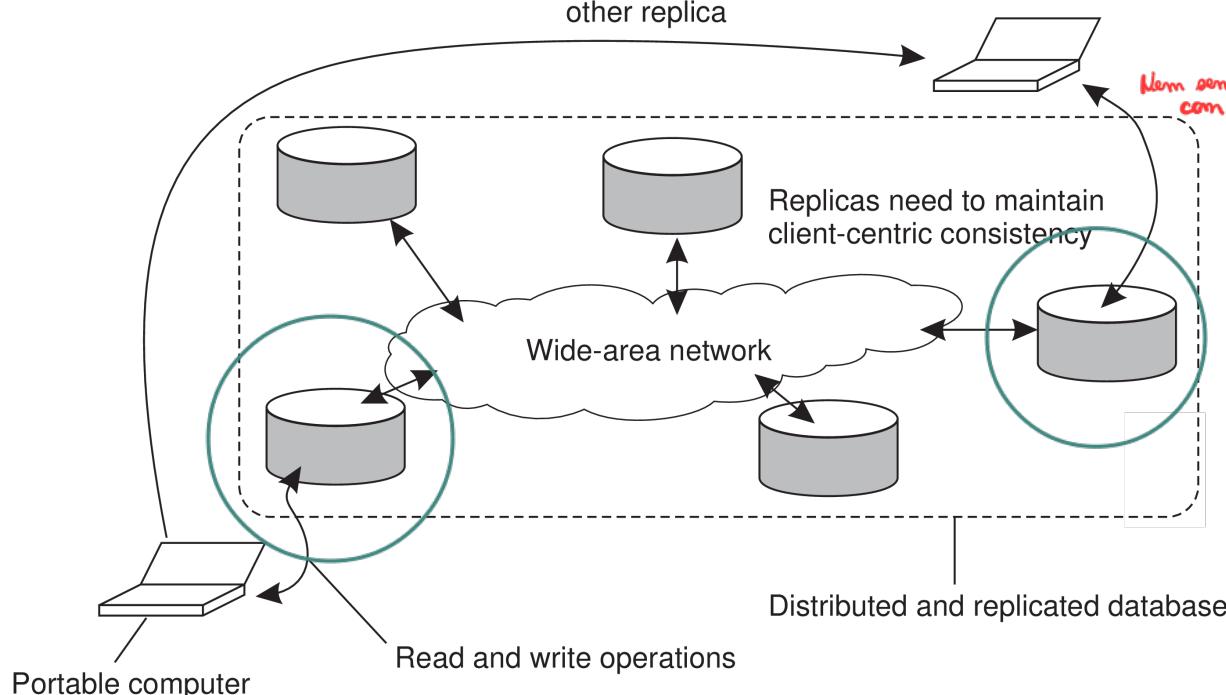
→ consistência em eventos!  
→ experiência é consistente!

Client moves to other location  
and (transparently) connects to  
other replica

As bases de dados podem  
não estar sincronizados!  
Mas a experiência é consistente

Nem sempre o cliente fala  
com o mesmo servidor

O feed do Instagram  
utiliza isto, vemosendo  
coisas diferentes enquanto  
não há consistência



Todos os operações para  
não corromper uma  
base de dados  
distribuída

*Client - Centric //*

## Leituras Monotónicas

- Definição ou x, ou um mais recente!
  - Se um processo ler o valor do dado x, qualquer operação de leitura sucessiva sobre x pelo mesmo processo retorna o mesmo valor ou mais recente.
- (a) leitura monotónica (b) leitura não monotónica

Server			
L1:	$W_1(x_1)$	$R_1(x_1)$	
L2:	$W_2(x_1; x_2)$	$R_1(x_2)$	

*em consequência*

Server			
L1:	$W_1(x_1)$	$R_1(x_1)$	
L2:	$W_2(x_1 x_2)$	$R_1(x_2)$	

*Escrita concorrente*      *"pipe" escrita não consequencial (não escrito relacionado com  $x_1$ )*

# Escritas Monotónicas

Não existem escritos simultâneos no mesmo processo //

- Definição
  - Uma operação de escrita por um processo num dado  $x$  é completada antes de qualquer operação de escrita sucessiva em  $x$  pelo mesmo processo.
- (a) escrita monotónica (b) data store que não disponibiliza consistência por escrita monotónica (c) sem consistência por escrita monotónica (d) Consistente pois  $W_1(x_1; x_3)$  apesar de  $x_1$  aparentemente reescrever  $x_2$ .

a) L1:  $W_1(x_1)$

---

L2:  $W_2(x_1; x_2)$        $W_1(x_2; x_3)$

b) L1:  $W_1(x_1)$

---

L2:  $W_2(x_1|x_2)$        $W_1(x_1|x_3)$

c) L1:  $W_1(x_1)$

---

L2:  $W_2(x_1|x_2)$        $W_1(x_2; x_3)$

*↑ paralelo*      *↑ conexão*

*Mas continua não monotónica*

d) L1:  $W_1(x_1)$

---

L2:  $W_2(x_1|x_2)$        $W_1(x_1; x_3)$

*↑ preferiu o antigo*

*Corrigi sobre algo que era consistente*

*Aqui ele pôs o fio consistente*

e.g.: Operação que destroi saldo, por mais que corriga...  
↳ Mas se eu voltar no valor antigo posso corrigir! \*

# Leitura após escrita

Write-Read

Read - your - writes

Sai no teste

- Definição
  - O efeito da operação de escrita por um processo nos dados x, será sempre visto pelas operações de leitura sucessivas em x pelo mesmo processo.
- (a) Leitura após escrita consistente (b) Não consistente

$$L1: W_1(x_1)$$

$$L2: W_2(x_1; x_2)$$

$$\underline{R_1(x_2)}$$

? Mesmo por  
outro processo

$$L1: W_1(x_1)$$

$$L2: \underline{W_2(x_1|x_2)}$$

$$\underline{\text{concorrência}}$$

$$R_1(x_2)$$

↑  
Podemos estar  
a ler algo  
inconsistente!

# Escrita após leitura

Read - Write

write-follows-reads

Sai no teste

- Definição
  - Numa operação de escrita por um processo sobre os dados  $x$  após uma operação de leitura prévia sobre  $x$  pelo mesmo processo, é garantido que ocorre sobre o mesmo valor de  $x$  que foi lido ou mais recente.
- (a) Escrita após leitura consistente (b) não consistente

L1:	$W_1(x_1)$	$R_2(x_1)$
L2:	$W_3(x_1;x_2)$	$\underline{W_2(x_2;x_3)}$

L1:	$W_1(x_1)$	$R_2(x_1)$
L2:	$W_3(x_1 x_2)$	$\underline{W_2(x_1 x_3)}$

# Localização de Replicas

Netflix faz muito isto...



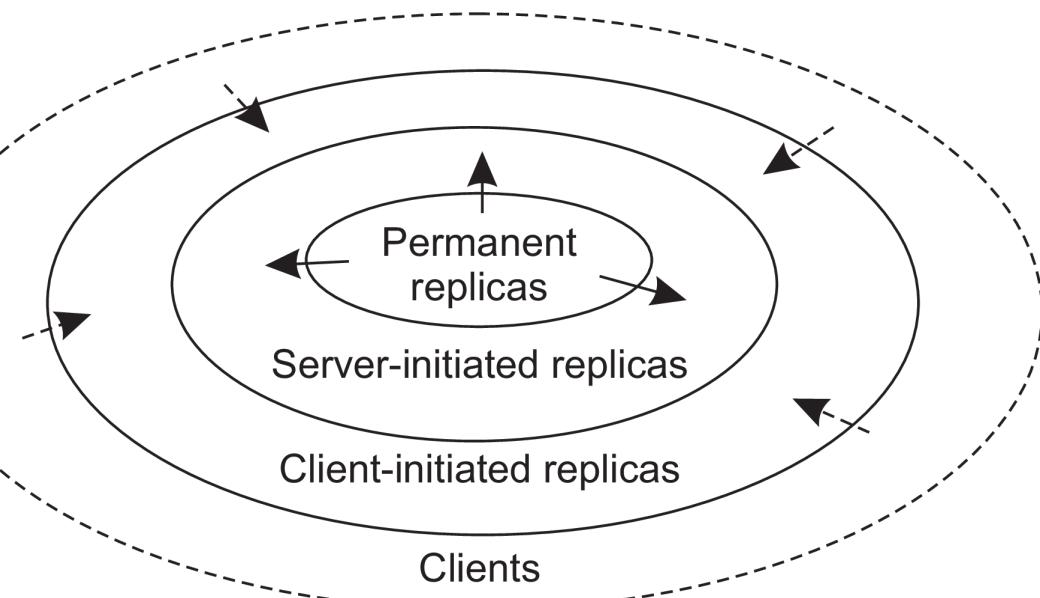
- Descobrir quais os melhores K locais de entre N possíveis localizações
  - Seleccionar a melhor localização de entre N localizações possíveis, para a qual a média da distância aos clientes é mínima. De seguida escolher o melhor servidor. Repetir o processo K vezes. (caro) *> Pode ser difícil definir antes... (Por exemplo em todos as localizações)*
  - Seleccionar os Ks maiores Sistema Autonomo (AS) e colocar um servidor no computador melhor ligado. (caro) *⇒ Em Lisboa... País do GigaPix*
  - Posicionar nós num espaço d-dimensional, onde a distância reflete a latência. Identificar as K regiões com densidade mais elevada e colocar um servidor em cada uma. (barato) *⇒ Tem de ser dinâmico!*

# Replicação de conteúdo

- Distinguir os diversos processos:

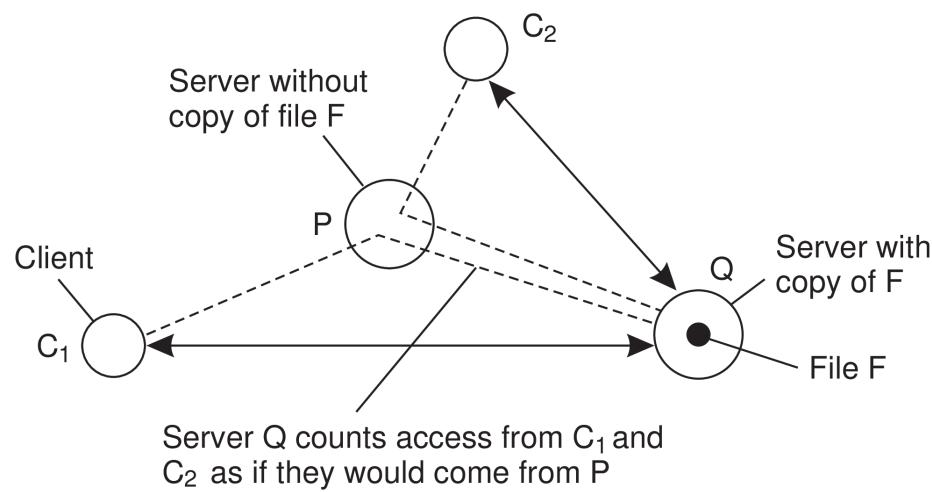
- Replicas permanentes:  
Processo/máquina tem sempre uma réplica
- Replicas iniciadas pelo servidor:  
Processo pode dinamicamente hospedar uma réplica a pedido do servidor.  
*No nosso PC guardamos websites para a próxima vez que precisam (Servidor poupe...)*
- Replicas iniciada pelo cliente:  
Processo pode dinamicamente hospedar uma réplica a pedido do cliente (client cache)

*//Mais normal!//*



# Replicas iniciadas pelo servidor

Primeiro acesso é lento... (acesso ao Datastore...)

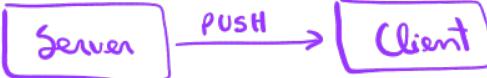


- Guardar número de acessos por ficheiro, agregar por servidor
- Número de acessos cai abaixo do limite D -> apagar ficheiro
- Número de acessos acima do limite R -> replicar ficheiro ↗ *Muito popular*
- Número entre D e R -> migrar ficheiro  
↗ *Se a série ficar mais popular ele envia para a zona + popular*

# Distribuição de Conteúdos

- Considerando apenas Cliente-Servidor:
  - Propagar apenas **notificação**/invalidação de uma actualização (caches)
  - Transferir **dados** de uma cópia para outra (DB's distribuídas): **replicação passiva**
  - Propagar a operação de **actualização** para outras cópias: **replicação activa**

# Distribuição de Conteúdos: Cliente/Servidor

- PUSH • Push updates: iniciado pelo servidor, actualização é propagada independentemente do facto do destino ter pedido ou não
- Cenários de Backup!*  
*PUSH para o Slave,*  
*Envio*
- 

- PULL • Pulling updates: iniciado pelo cliente, em que o cliente solicita o envio de actualizações
- Ela sente necessidade → Se for distribuído em peso a todos*
- É possível comutar dinamicamente entre os dois modos através de “leases”: Um contrato no qual o servidor promete fazer o push de updates para o cliente até o contrato expirar
- Back up de hora em hora → 5 em 5 minutos (vai mudando)*  
*Vai vendo ...*
- 
- Bom para o Projeto Final*

Leasing: o cliente tem um contrato e o servidor vai dando PUSH durante esse contrato, quando acaba ele faz PULL

# Comparação entre push e pull protocols para replicação

*Podemos ficar  
com a cache  
incompatível  
por muito tempo*

	Push-based	Pull-based
Estado no servidor	<p><i>vais pedindo atualizações... (PUSH)</i></p> <p><u>Lista de replicas nos clientes e caches</u></p>	-
Mensagens trocadas	Update (e possivelmente <i>fetch-update</i> se tiver sido uma invalidação)	<i>Poll e Update</i>
Tempo de resposta no cliente	Imediato (ou tempo de <i>fetch-update</i> ) 	<u>Tempo de Fetch-update</u> <i>tempo de juntar tudo e retornar</i>

# Distribuição de Conteúdos: Leases

- Tempo de uma "lease" deve ser dependente do comportamento do sistema
  - **Leases baseadas na idade:** De um objecto que não é alterado faz muito tempo, não é esperado que venha a sofrer alteração nos tempos próximos, pelo que o tempo de lease deve ser longo.  
*tempo de atualização pode ser longo!*
  - **Leases com base na frequência de renovação:** Clientes que pedem frequentemente um dado objecto, deverão ter um tempo de expiração mais longo.
  - **Leases baseadas no estado:** Quanto mais carga tiver o servidor, mais curtos devem ser os períodos de lease.  
*Quando tiveres decidido avisa... (esperemos a versão de código)*  
*Deve enviar atualizações mais rápidas!*

# Unicasting vs multicasting

(Broadcast)

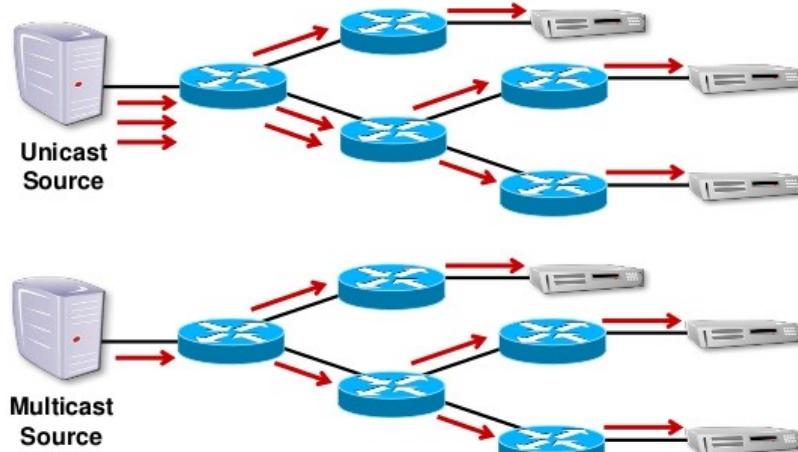
8 Melhor, a rede  
replica ...

Quando temos disponível um meio de físico  
de broadcast:

Podemos fazer *push* de forma eficiente,  
com o mesmo custo de ptpt

↳ Ponto a ponto

## Unicast vs. Multicast



© 2012 Cisco and/or its affiliates. All rights reserved.

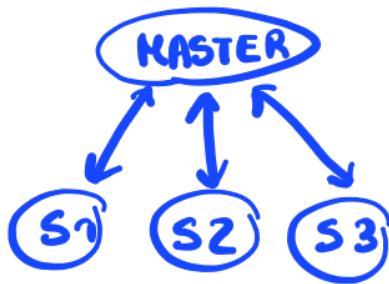
Cisco Public

12

# Consistência Continua

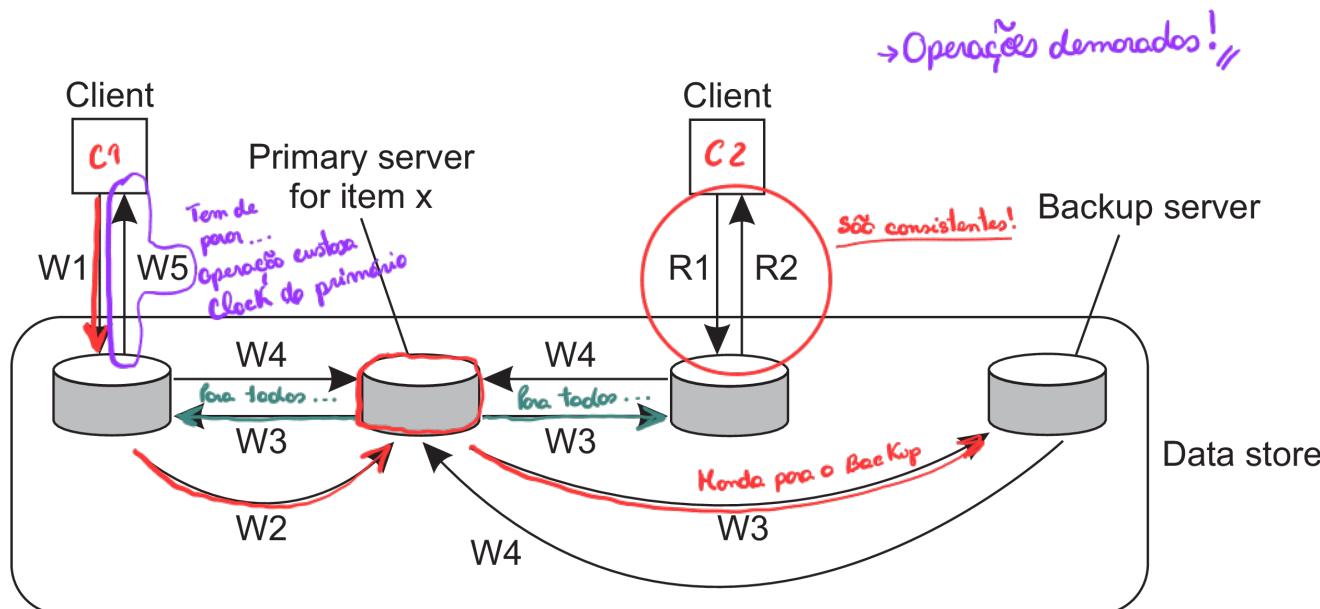
• Para base de dados de bancos!  
• Não existem correções!

- Limitar desvio numérico
  - Garantir que valores não se afastam de um intervalo
- Limitar desvio idade
  - Garantir que não há um atraso de atualizações maior que um intervalo
- Limitar desvio de ordens
  - Garantir um numero máximo de ordens em atraso



• Para garantirmos consistência contínua //

# Protocolos baseados num primário



W1. Write request

W2. Forward request to primary

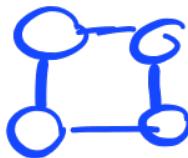
W3. Tell backups to update

W4. Acknowledge update

W5. Acknowledge write completed

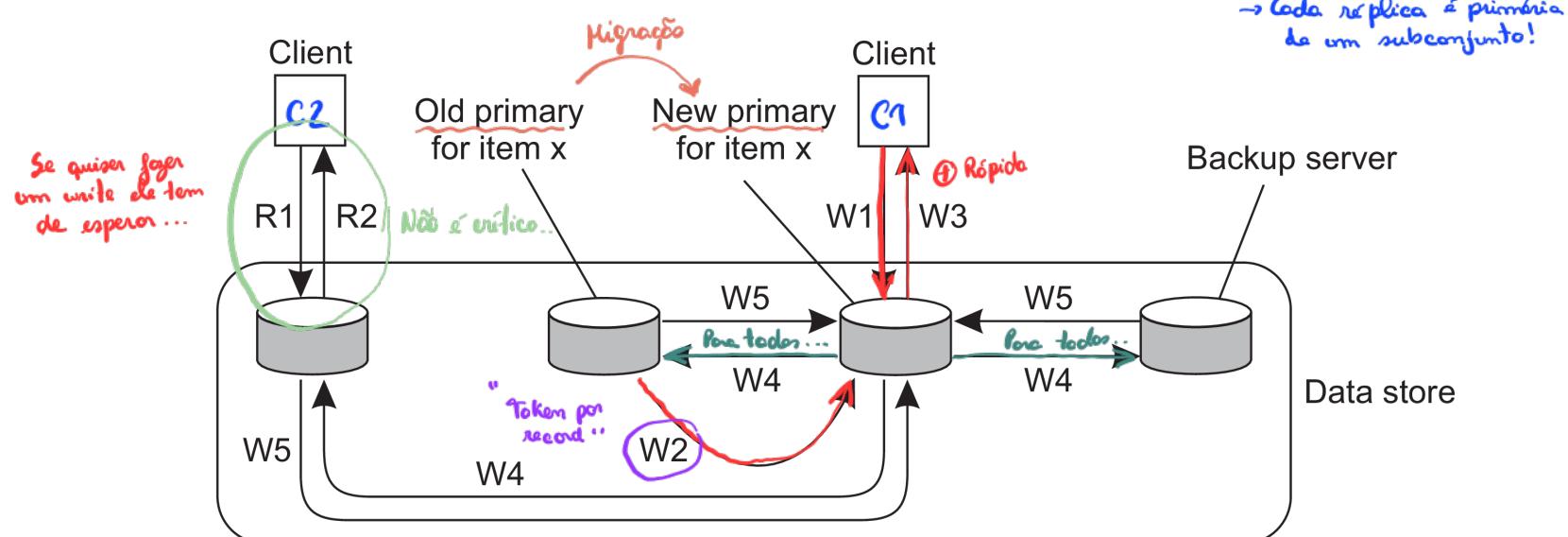
R1. Read request

R2. Response to read



Primário com escritas locais!  
→ Primário por Record...

# Protocolos baseados num primário com escritas locais



- W1. Write request
- W2. Move item x to new primary
- W3. Acknowledge write completed
- W4. Tell backups to update
- W5. Acknowledge update

- R1. Read request
- R2. Response to read

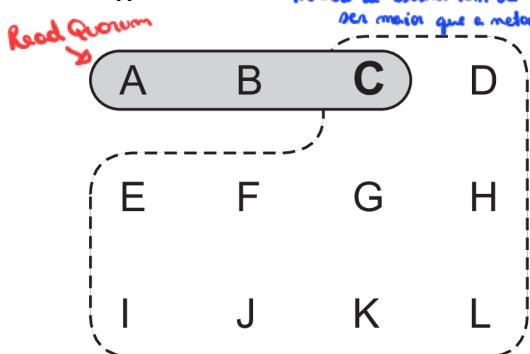
# Protocolos escrita replicada

DEFINIR Servidores de Leitura e Servidores de Escrita

## Read Quorum vs Write Quorum

Tem que obdecer as seguintes regras:

- Nóros
  - $N_R + N_W > N_{\text{Nós}}$ !
  - $N_W > N/2$  Número de escritores tem de ser maior que a metade dos nós

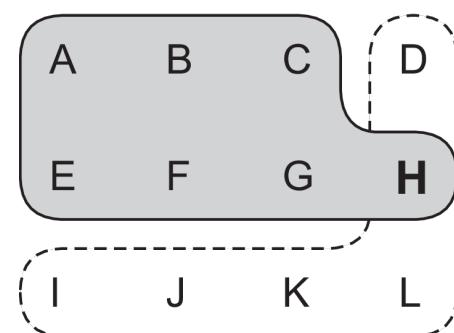


$$N_R = 3, \quad N_W = 10$$

(a) Escolha correcta de read e write  
(Read One Write All)

- Para ler um ficheiro com  $N$  réplicas o cliente deve juntar um Read-Quorum!
- Para escrever  $N_W$

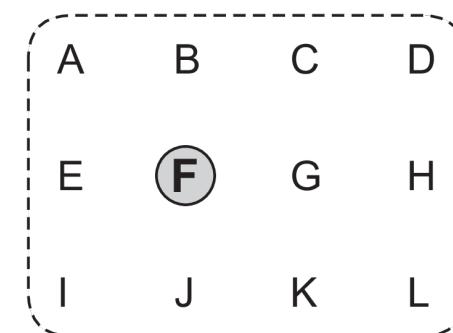
Se um W5 correr Mal!  
Base de dados inconsistentes...



$$N_R = 7, \quad N_W = 6$$

XX  $N_W > \frac{N}{2}$

Traz problemas de write-write



$$N_R = 1, \quad N_W = 12$$

Read one  
write all

↑  
Algoritmo de votação!