MCTA025-13 - SISTEMAS DISTRIBUÍDOS

REPLICAÇÃO E CONSISTÊNCIA

Emilio Francesquini

13 e 15 de agosto de 2018

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

DISCLAIMER

- Estes slides foram preparados para o curso de **Sistemas Distribuídos na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Estes slides foram adaptados daqueles originalmente preparados (e gentilmente cedidos) pelo professor **Daniel Cordeiro, da EACH-USP** que por sua vez foram baseados naqueles disponibilizados online pelos autores do livro "Distributed Systems", 3ª Edição em: `https://www.distributed-systems.net`.

- Introdução (do que se trata isso?)
- Consistência centrada nos dados
- Consistência centrada no cliente
- Gerenciamento de réplicas
- Protocolos de consistência

Problema principal

Para manter a consistência entre as réplicas, geralmente precisamos garantir que todas as operações **conflitantes** sejam realizadas na mesma ordem em todas as réplicas

Operações conflitantes

Terminologia da área de controle de transações:

read–write conflito onde uma operação de leitura e uma de escrita ocorrem de forma concorrente

write–write conflito com duas operações concorrentes de escrita

Problema

Garantir a ordem global de operações conflitantes pode ser muito custoso, diminuindo a escalabilidade. Solução: diminuir os requisitos de consistência e, com sorte, conseguir evitar sincronizações globais

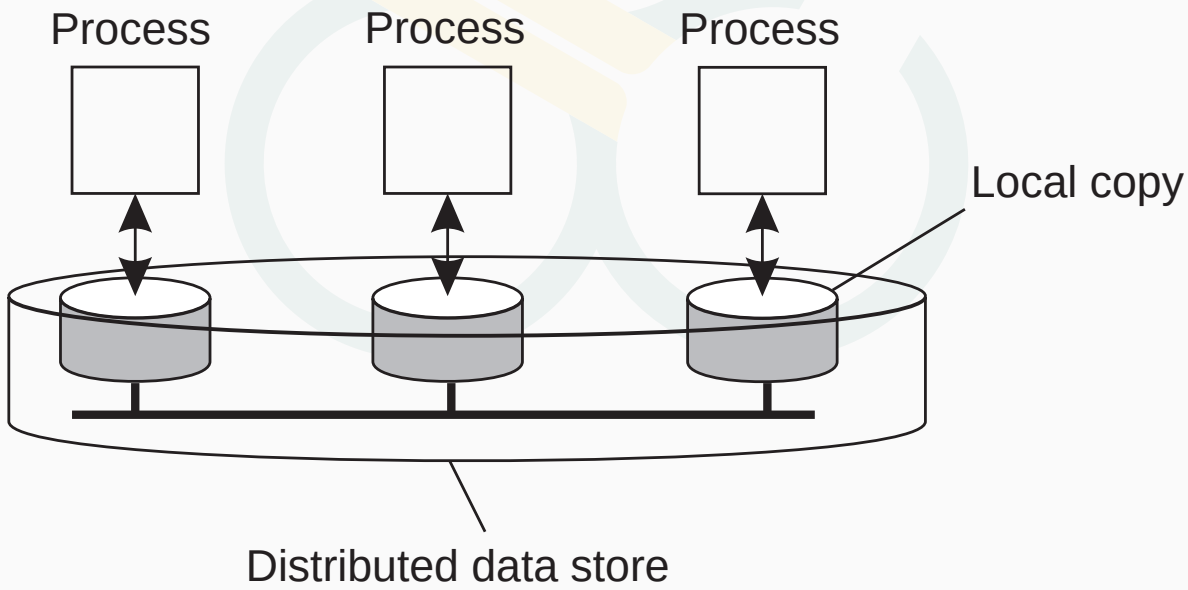
MODELOS DE CONSISTÊNCIA CENTRADOS EM DADOS

Modelo de consistência

É um contrato entre um *data store* (armazém de dados) distribuído e os processos, no qual o *data store* define precisamente o resultado de operações concorrentes de leitura e escrita

Importante:

Um *data store* é uma coleção de dispositivos de armazenamento distribuídos



Observação

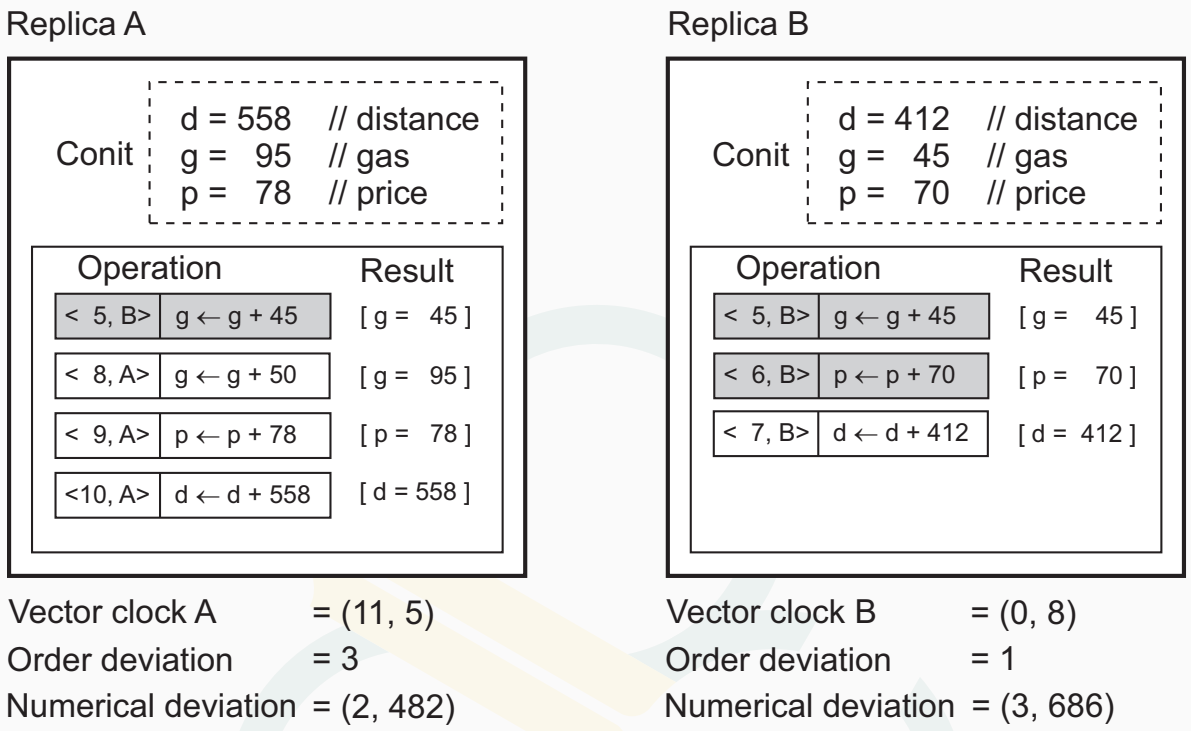
Podemos considerar diferentes **graus de consistência**:

- réplicas podem diferir em relação aos seus **valores numéricos**
- réplicas podem diferir em relação à **desatualização relativa**
- pode haver diferenças no número e na ordem das **operações de atualizações realizadas**

Conit: consistency unit

Especifica a unidade de dados sob a qual a consistência será medida

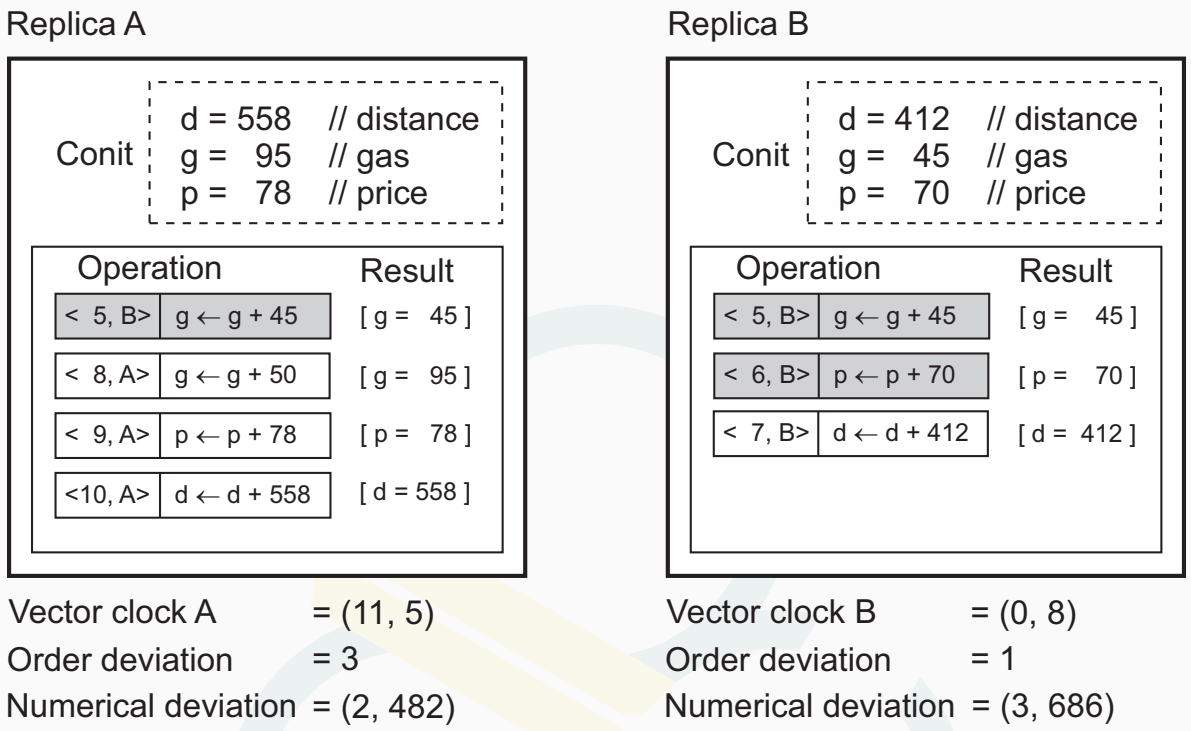
EXEMPLO: CONIT



Conit: variáveis d, g e p

- Cada réplica possui um **relógio vetorial**:
  (tempo conhecido @ A, tempo conhecido @ B)
- *B* envia à *A* a operação $[\langle 5, B\rangle: g \leftarrow g + 45]$; *A* faz com que a operação se torne permanente (não pode ser *rolled back*)

EXEMPLO: CONIT

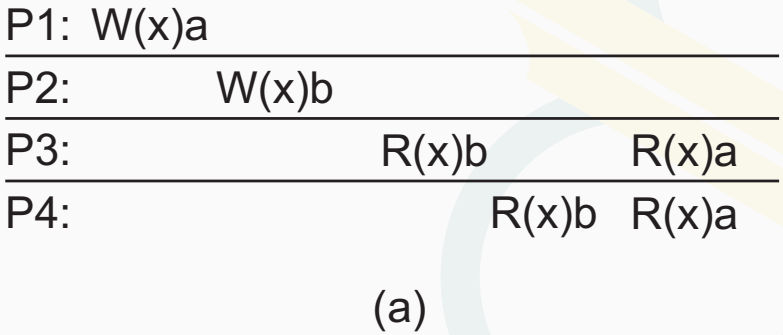


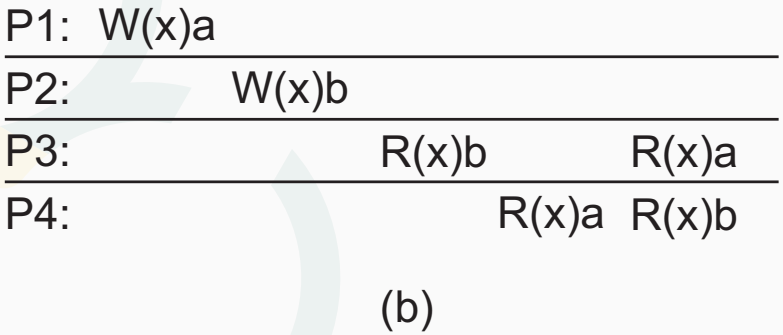
Conit: variáveis d, g e p

- *A* tem três operações **pendentes** (desvio de ordem = 3)
- *A* perdeu **duas** operações de *B*, resultando em uma diferença máxima de 70+412 unidades $\Rightarrow (2, 482)$ (desvio numérico)

Definição

O resultado de qualquer execução é o mesmo, como se as operações de todos os processos fossem executadas na mesma ordem sequencial e as operações de cada processo aparecer nessa sequência na ordem especificada pelo seu programa.





(a) é um *data store* com consistência sequencial; (b) não apresenta consistência sequencial

CONSISTÊNCIA SEQUENCIAL — EXEMPLO

Processo 1	Processo 2	Processo 3
`x = 1;`	`Y = 1;`	`Z = 1;`
`print(Y, Z);`	`print(X, Z);`	`print(X, Y);`

CONSISTÊNCIA SEQUENCIAL — EXEMPLO

Processo 1	Processo 2	Processo 3
`x = 1;`	`Y = 1;`	`Z = 1;`
`print(Y, Z);`	`print(X, Z);`	`print(X, Y);`

- **6!** combinações possíveis...

CONSISTÊNCIA SEQUENCIAL — EXEMPLO

Processo 1	Processo 2	Processo 3
`x = 1;`	`Y = 1;`	`Z = 1;`
`print(Y, Z);`	`print(X, Z);`	`print(X, Y);`

- **6!** combinações possíveis...
- ...mas são todas válidas?

CONSISTÊNCIA SEQUENCIAL — EXEMPLO

Execution 1

```
P1:  x ← 1;
P1:  print(y,z);
P2:  y ← 1;
P2:  print(x,z);
P3:  z ← 1;
P3:  print(x,y);
```

Prints: 001011
Signature: 0 0 1 0 1 1

(a)

Execution 2

```
P1:  x ← 1;
P2:  y ← 1;
P2:  print(x,z);
P1:  print(y,z);
P3:  z ← 1;
P3:  print(x,y);
```

Prints: 101011
Signature: 1 0 1 0 1 1

(b)

Execution 3

```
P2:  y ← 1;
P3:  z ← 1;
P3:  print(x,y);
P2:  print(x,z);
P1:  x ← 1;
P1:  print(y,z);
```

Prints: 010111
Signature: 1 1 0 1 0 1

(c)

Execution 4

```
P2:  y ← 1;
P1:  x ← 1;
P3:  z ← 1;
P2:  print(x,z);
P1:  print(y,z);
P3:  print(x,y);
```

Prints: 111111
Signature: 1 1 1 1 1 1

(d)

CONSISTÊNCIA SEQUENCIAL — EXEMPLO

Execution 1	Execution 2	Execution 3	Execution 4
P_1: x ← 1; P_1: print(y,z); P_2: y ← 1; P_2: print(x,z); P_3: z ← 1; P_3: print(x,y);	P_1: x ← 1; P_2: y ← 1; P_2: print(x,z); P_1: print(y,z); P_3: z ← 1; P_3: print(x,y);	P_2: y ← 1; P_3: z ← 1; P_3: print(x,y); P_2: print(x,z); P_1: x ← 1; P_1: print(y,z);	P_2: y ← 1; P_1: x ← 1; P_3: z ← 1; P_2: print(x,z); P_1: print(y,z); P_3: print(x,y);
Prints: 001011 *Signature:* 00 10 11	*Prints:* 101011 *Signature:* 10 10 11	*Prints:* 010111 *Signature:* 11 01 01	*Prints:* 111111 *Signature:* 11 11 11
(a)	(b)	(c)	(d)

- A assinatura 00 00 00 é válida?

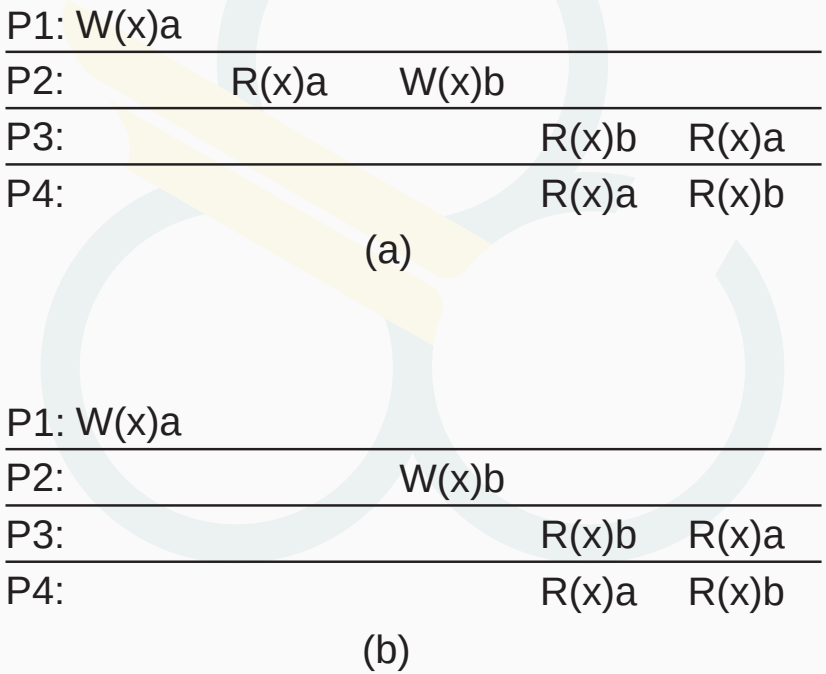
CONSISTÊNCIA SEQUENCIAL — EXEMPLO

Execution 1	Execution 2	Execution 3	Execution 4
P_1: x ← 1; P_1: print(y,z); P_2: y ← 1; P_2: print(x,z); P_3: z ← 1; P_3: print(x,y);	P_1: x ← 1; P_2: y ← 1; P_2: print(x,z); P_1: print(y,z); P_3: z ← 1; P_3: print(x,y);	P_2: y ← 1; P_3: z ← 1; P_3: print(x,y); P_2: print(x,z); P_1: x ← 1; P_1: print(y,z);	P_2: y ← 1; P_1: x ← 1; P_3: z ← 1; P_2: print(x,z); P_1: print(y,z); P_3: print(x,y);
Prints: 001011 *Signature:* 00 10 11	*Prints:* 101011 *Signature:* 10 10 11	*Prints:* 010111 *Signature:* 11 01 01	*Prints:* 111111 *Signature:* 11 11 11
(a)	(b)	(c)	(d)

- A assinatura `00 00 00` é válida?
- A assinatura `00 10 01` é válida?

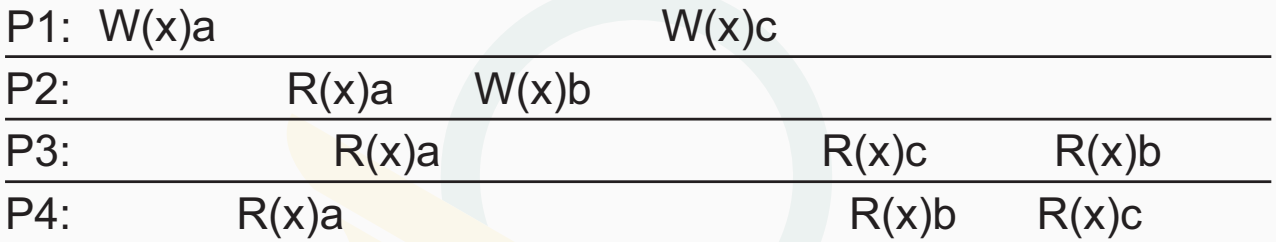
Definição

Operações de escrita que potencialmente possuem uma relação de causalidade devem ser vistas por todos os processos na mesma ordem. Escritas concorrentes podem ser vistas em uma ordem diferente por processos diferentes.



(a) uma violação da consistência causal; (b) uma sequência correta de eventos em um *data store* com consistência causal

COMPATIBILIDADE SEQUENCIAL-CAUSAL



- É **sequencialmente** consistente?
- É **causalmente** consistente?

Definição

- acessos às variáveis de sincronização (*locks*) possuem consistência sequencial
- o acesso às variáveis de sincronização não é permitido até que todas as escritas anteriores tenham terminado em todos os lugares
- nenhum acesso aos dados é permitido até que todos os acessos às variáveis de sincronização tenham sido feitos

Ideia básica:

Você não precisa se preocupar se as leituras e escritas de uma **série** de operações serão imediatamente do conhecimento de todos os processos. Você só quer que o **efeito** dessa série seja conhecido.

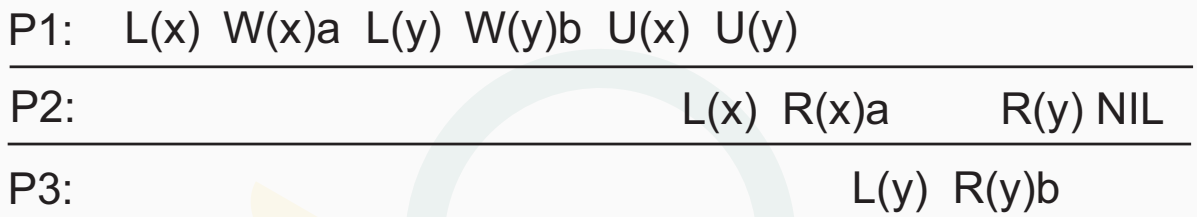


Figura: Um sequência de eventos que respeita a consistência de entrada.

Observação

Consistência de entrada implica a necessidade de proteger os dados com *locks* (implícitos ou não)

EVENTUAL CONSISTENCY

- Dependendo da aplicação não é problema que as **atualizações não sejam propagadas imediatamente**
  - Normalmente os clientes acessam a mesma réplica, então não há problemas de inconsistência (na visão do cliente)
- Permite uma implementação de **modelo de consistência com menos restrições e portanto mais eficiente** (PQ?)
- Exemplos:
  - Facebook
  - DNS
  - Páginas Web em Geral

EVENTUAL (EN) ≠ EVENTUAL (PT)

O mesmo vale para espanhol, francês, ...

- **eventual (en)** *adjective*: happening at some indefinite future time or after a series of occurrences; ultimate
- **eventual (pt)** *a2g*: 1. Que é incerto, podendo acontecer ou deixar de acontecer; CASUAL; FORTUITO. 2. Que ocorre de vez em quando; OCASIONAL: Tínhamos encontros eventuais. [Antôn.: frequente.]

NÃO ESCREVA "consistência eventual" na prova sob risco de causar revolta no seu professor.

- E você não quer o seu professor revoltado enquanto ele corrige sua prova. ☺

Prefira consistência diferida, consistência postergada, ou até mesmo o termo em inglês "*eventual consistency*"

- Modelo do sistema
- Leituras monotônicas
- Escritas monotônicas
- *Read-your-writes* (leia-suas-escritas)
- *Write-follows-reads* (escrita-segue-leituras)

Objetivo

Mostrar que talvez manter a consistência em todo o sistema seja desnecessário se nos concentramos no que os **clientes** precisam, ao invés daquilo que deve ser mantido pelos servidores.

Exemplo

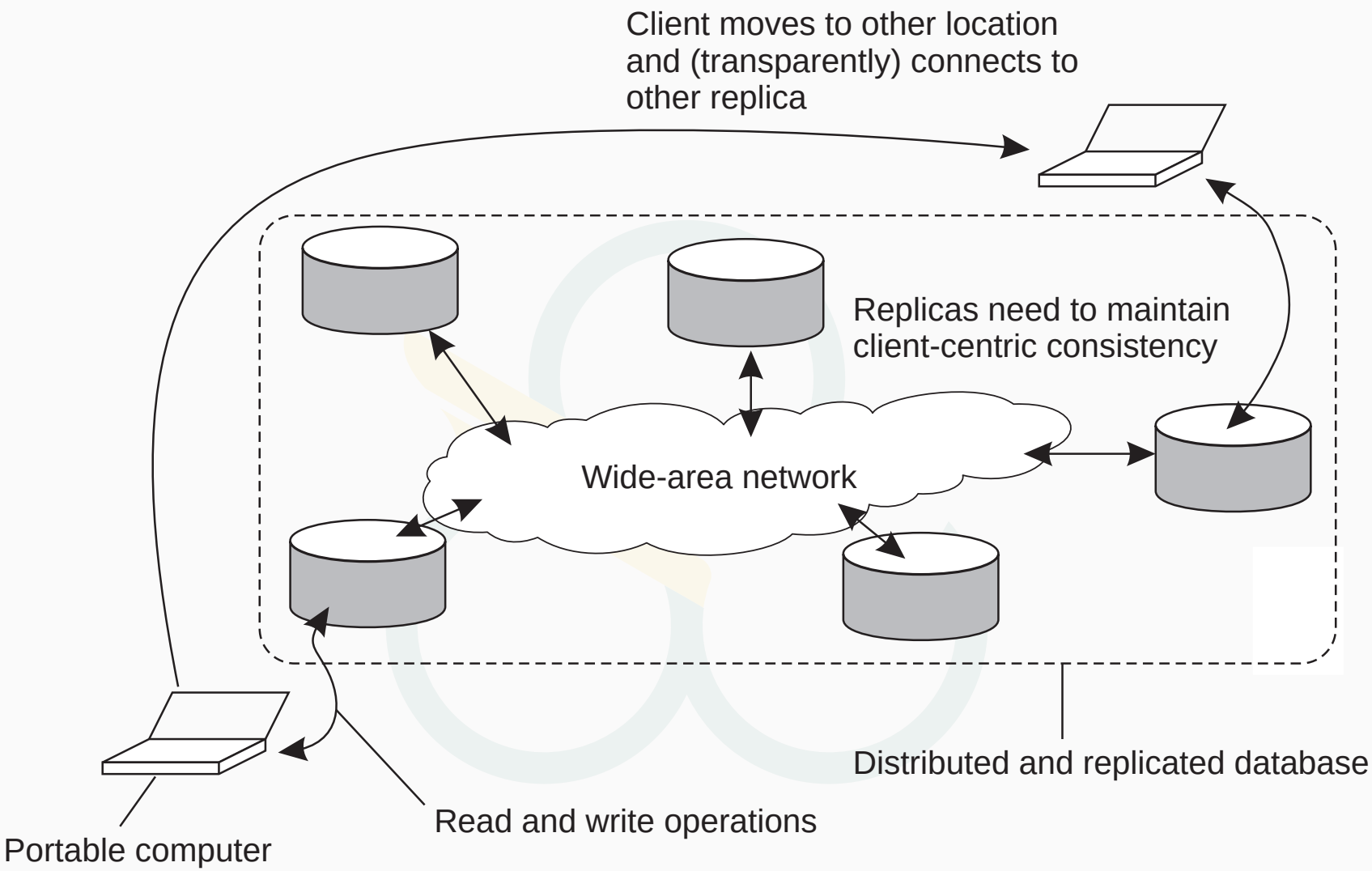
Considere um sistema de banco de dados distribuídos no qual você tem acesso pelo seu notebook. Assuma que seu notebook seja o *front end* do seu banco de dados.

- no local *A* você acessa o banco de dados e realiza leituras e atualizações
- no local *B* você continua seu trabalho, mas, a não ser que você continue acessando o mesmo servidor de antes, você poderá detectar algumas inconsistências:
  - suas atualizações em *A* podem ainda não terem sido propagadas para *B*
  - você pode estar lendo entradas mais novas do que aquelas disponíveis em *A*
  - suas atualizações em *B* podem eventualmente conflitar com aquelas em *A*

Observação

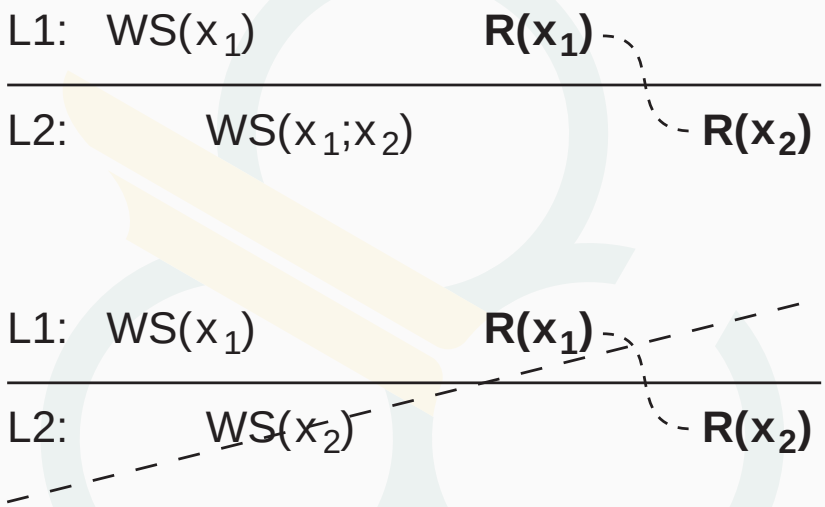
A única coisa que você realmente precisa é que as entradas que você atualizou e/ou leu em *A* estejam em *B* do modo que você as deixou em *A*. Nesse caso, o banco de dados parecerá consistente **para você**

ARQUITETURA BÁSICA



Definição

Se um processo ler o valor de um item *x*, quaisquer leituras sucessivas de *x* feitas por esse processo sempre devolverão o mesmo valor ou um valor mais recente.



Leituras realizadas por um processo *P* em duas cópias locais diferentes do mesmo *data store*. (a) Uma leitura monotônica consistente; (b) um *data store* que não provê leituras monotônicas

Notação

- $W_1(x_2)$ é a operação de escrita feita pelo processo P_1 que leva à versão x_2 de x
- $W_1(x_i; x_j)$ indica que P_1 produziu a versão x_j baseado na versão anterior x_i
- $W_1(x_i|x_j)$ indica que P_1 produziu a versão x_j concorrentemente a versão x_i

Exemplo

Leituras automáticas das atualizações em seu calendário pessoal vindas de diferentes servidores. Leituras monotônicas garantem que o usuário veja todas as atualizações, independentemente do servidor que originou a leitura

Exemplo

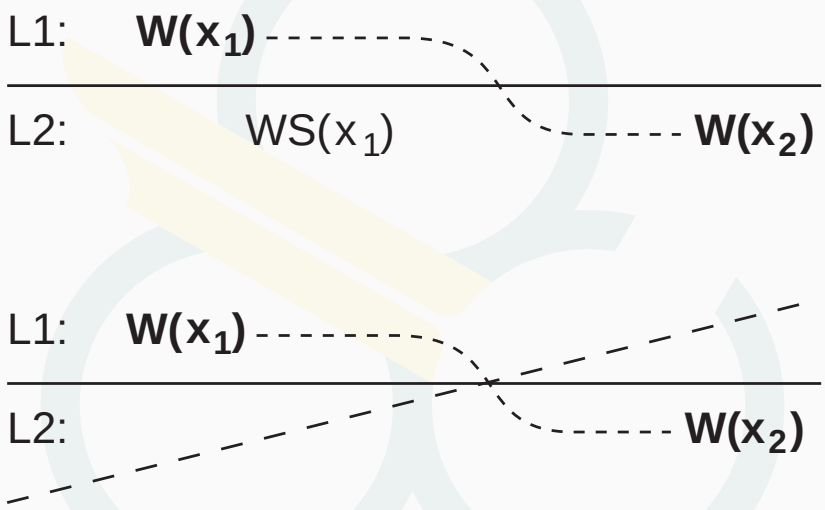
Leituras automáticas das atualizações em seu calendário pessoal vindas de diferentes servidores. Leituras monotônicas garantem que o usuário veja todas as atualizações, independentemente do servidor que originou a leitura

Exemplo

Ler (sem modificar) as mensagens enquanto você estiver em movimento. Toda vez que você se conectar a um servidor de e-mails diferente, o servidor irá descarregar (pelo menos) todas as atualizações do servidor que você visitou antes

Definição

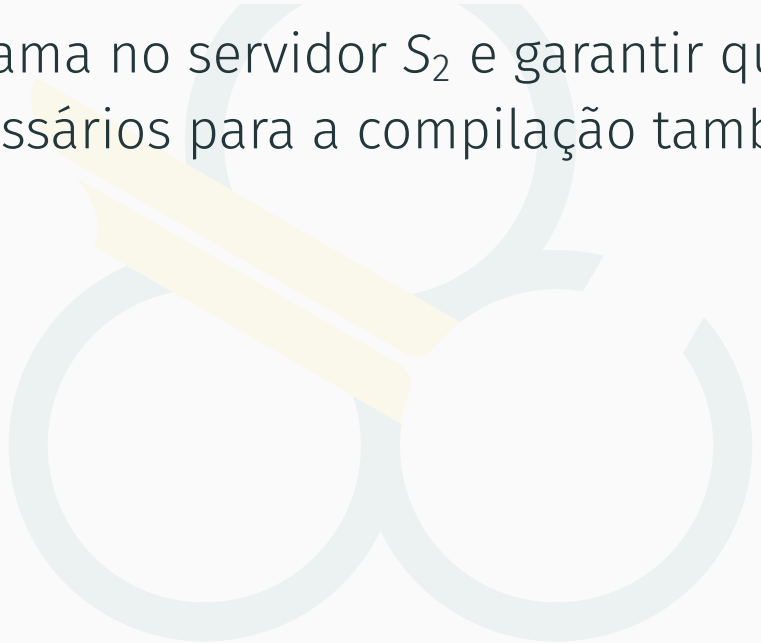
Uma escrita monotônica feita por um processo em um dado x é terminada antes de quaisquer operações de escrita sucessivas em x por esse mesmo processo.



Ou seja, se tivermos duas escritas sucessivas $W_k(x_i)$ e $W_k(x_j)$, então não importa onde $W_k(x_j)$ acontece, sempre teremos $W_k(x_i; x_j)$.

Exemplo

Atualizar um programa no servidor S_2 e garantir que todos os componentes necessários para a compilação também estejam em S_2



Exemplo

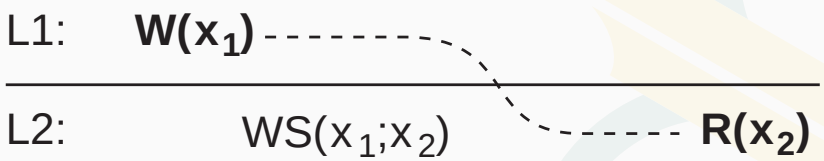
Atualizar um programa no servidor S_2 e garantir que todos os componentes necessários para a compilação também estejam em S_2

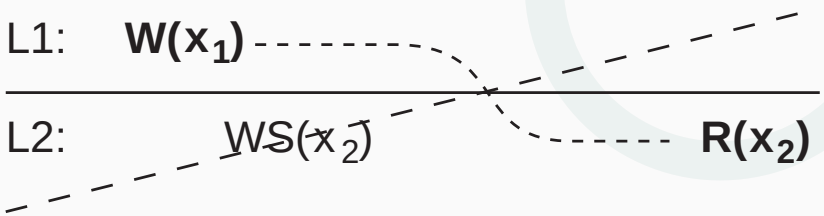
Exemplo

Manter versões de arquivos replicados na ordem correta em todos os lugares (propagando as versões antigas para o servidor onde a versão mais nova está instalada)

Definição

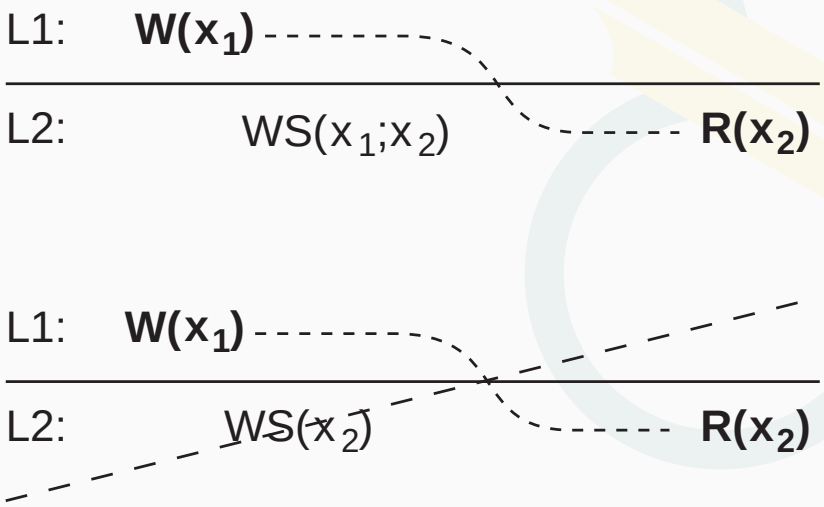
O efeito de uma operação de escrita realizada por um processo no item *x* sempre será visto por operações de leituras de *x* pelo mesmo processo.





Definição

O efeito de uma operação de escrita realizada por um processo no item *x* sempre será visto por operações de leituras de *x* pelo mesmo processo.

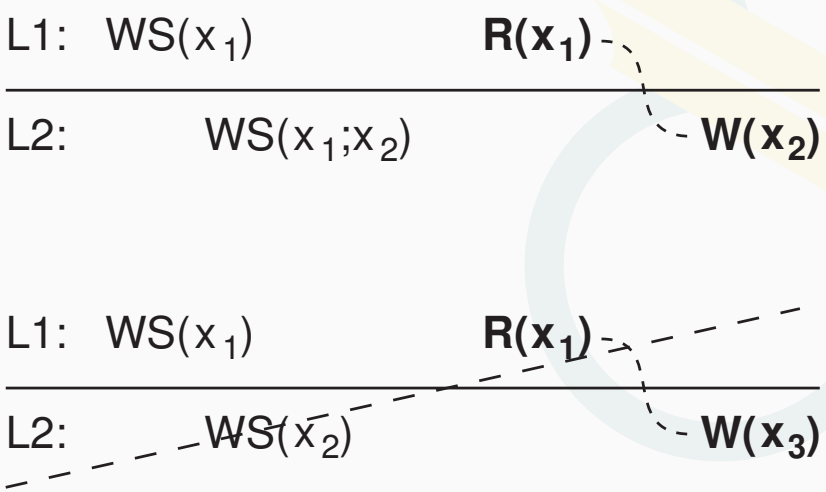


Exemplo

Atualizar sua página web e garantir que o navegador web mostre a versão mais nova ao invés de mostrar a versão em cache

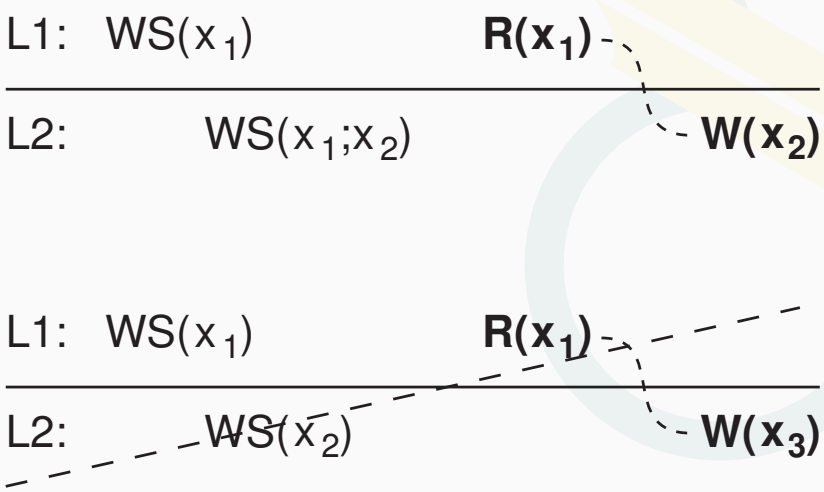
Definição

Uma operação de escrita feita por um processo no item *x* após uma operação de leitura de *x* no mesmo processo é garantidamente realizada no mesmo valor de *x* que foi lido (ou num valor mais novo)



Definição

Uma operação de escrita feita por um processo no item *x* após uma operação de leitura de *x* no mesmo processo é garantidamente realizada no mesmo valor de *x* que foi lido (ou num valor mais novo)



Exemplo

Ver os comentários a um artigo publicado apenas se você tiver o artigo original (uma leitura “puxa” as operações de escrita correspondentes)

GERENCIAMENTO DE RÉPLICAS

- posicionamento de servidores de réplicas
- replicação de conteúdo e posicionamento
- distribuição de conteúdo

Ideia

Encontrar as K melhores posições de uma lista de N possibilidades

- iterativamente selecionar as melhores posições de $N - K$ para as quais a distância média até os clientes é mínima e então escolher o próximo melhor servidor (a primeira posição escolhida é a que minimiza a distância média até todos os clientes). **Computacionalmente caro**

Ideia

Encontrar as K melhores posições de uma lista de N possibilidades

- iterativamente selecionar as melhores posições de $N - K$ para as quais a distância média até os clientes é mínima e então escolher o próximo melhor servidor (a primeira posição escolhida é a que minimiza a distância média até todos os clientes). **Computacionalmente caro**
- selecionar o K-ésimo maior **sistema autônomo** e colocar um servidor no host "melhor conectado". **Computacionalmente caro**

Ideia

Encontrar as K melhores posições de uma lista de N possibilidades

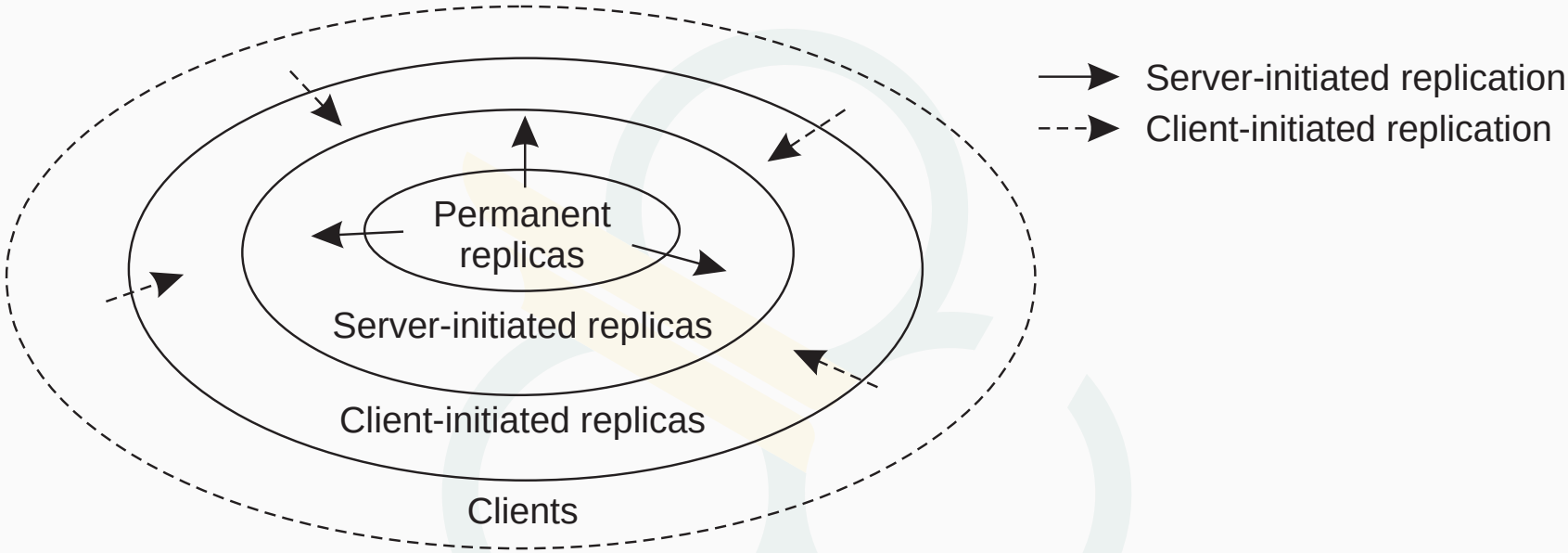
- iterativamente selecionar as melhores posições de $N - K$ para as quais a distância média até os clientes é mínima e então escolher o próximo melhor servidor (a primeira posição escolhida é a que minimiza a distância média até todos os clientes). **Computacionalmente caro**
- selecionar o K-ésimo maior **sistema autônomo** e colocar um servidor no host "melhor conectado". **Computacionalmente caro**
- posicionar os nós em um espaço geométrico d-dimensional, onde a distância reflete a latência. Identificar as K regiões mais densas e colocar um servidor em cada uma delas. **Computacionalmente barato**

Distingue diferentes processos

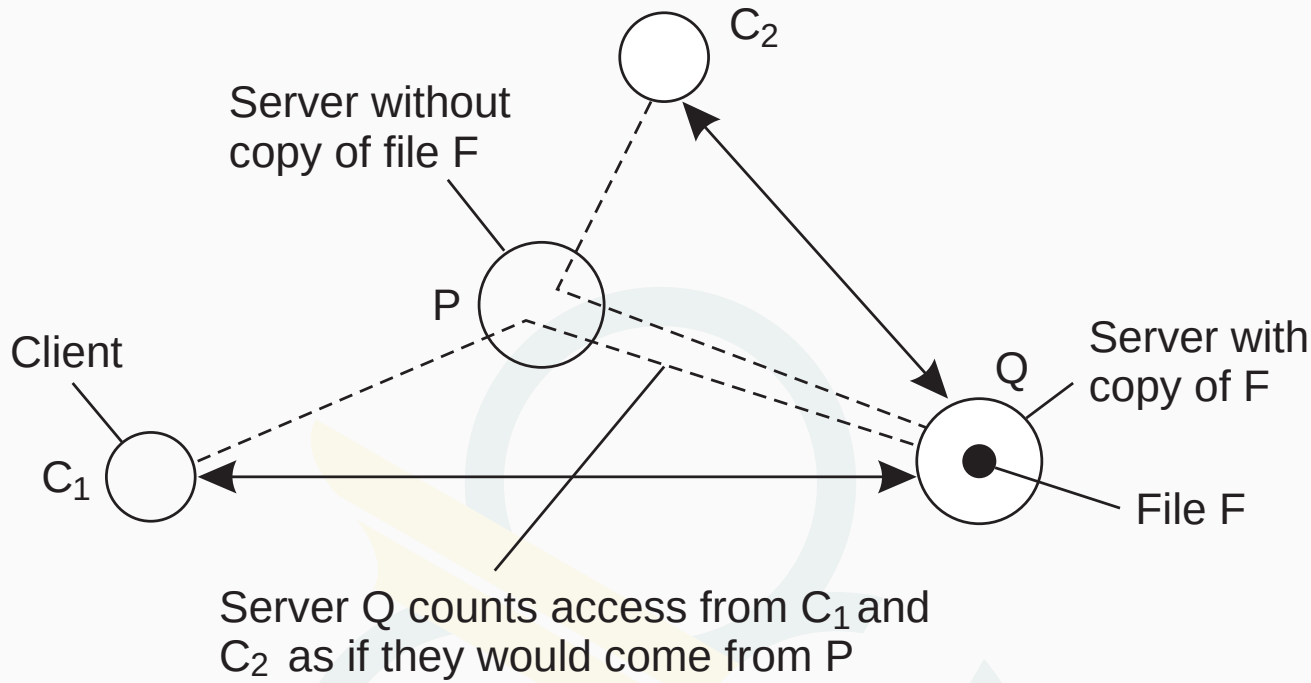
Um processo é capaz de hospedar uma réplica de um objeto ou dado:

- **réplicas permanentes:** processo/máquina sempre tem uma réplica
- **réplica iniciada pelo servidor:** processos que podem hospedar uma réplica dinamicamente, sob demanda de um outro servidor ou *data store*
- **réplica iniciada pelo cliente:** processos que podem hospedar uma réplica dinamicamente, sob demanda de um cliente (cache do cliente)

REPLICAÇÃO DE CONTEÚDO



RÉPLICAS INICIADAS PELO SERVIDOR



- mantenha o número de acessos aos arquivos, agregando-os pelo servidor mais próximo aos clientes que requisitarem o arquivo
- número de acessos cai abaixo de um threshold $D \Rightarrow$ descartar arquivo
- número de acessos acima de um threshold $R \Rightarrow$ replicar arquivo
- número de acessos entre D e $R \Rightarrow$ migrar arquivo

Modelo

Considere apenas uma combinação cliente–servidor:

- propaga apenas a **notificação/invalidação** de uma atualização (normalmente usada por caches)
- transfere dados de uma cópia para outra (bancos de dados distribuídos): replicação passiva
- propaga **operações** de atualização para outras cópias: replicação ativa

Nota

Nenhuma abordagem é melhor que outra, seu uso depende da largura de banda disponível e a razão leituras/escritas nas réplicas

pushing *iniciada pelo servidor*; uma atualização é propagada mesmo que o alvo não tenha pedido por ela

pulling *iniciada pelo cliente*; uma atualização solicitada pelo cliente

Observação

Podemos trocar dinamicamente entre os métodos *pulling* e *pushing* com o uso de **leases**: um contrato no qual o servidor promete enviar (*push*) atualizações para o cliente até que o *lease* expire.

Problema

Fazer com que a data de expiração do *lease* dependa do comportamento do sistema (*leases* adaptativos):

Problema

Fazer com que a data de expiração do *lease* dependa do comportamento do sistema (*leases* adaptativos):

- **leases com idade**: um objeto que não for modificado nos últimos tempos não será modificado em um futuro próximo, então conceda um *lease* que dure bastante

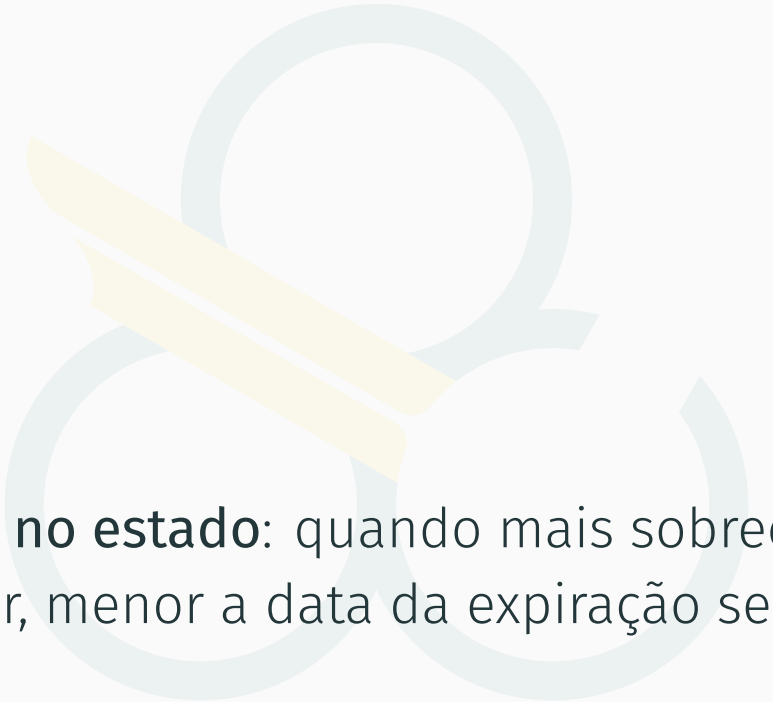
Problema

Fazer com que a data de expiração do *lease* dependa do comportamento do sistema (*leases* adaptativos):

- **lease baseado na frequência de renovação**: quanto maior a frequência com que o cliente requisitar o objeto, maior a data de expiração para aquele cliente (para aquele objeto)

Problema

Fazer com que a data de expiração do *lease* dependa do comportamento do sistema (*leases* adaptativos):



- **lease baseado no estado**: quando mais sobrecarregado o servidor estiver, menor a data da expiração se torna

Problema

Fazer com que a data de expiração do *lease* dependa do comportamento do sistema (*leases* adaptativos):

- **leases com idade**: um objeto que não for modificado nos últimos tempos não será modificado em um futuro próximo, então conceda um *lease* que dure bastante
- **lease baseado na frequência de renovação**: quanto maior a frequência com que o cliente requisitar o objeto, maior a data de expiração para aquele cliente (para aquele objeto)
- **lease baseado no estado**: quando mais sobrecarregado o servidor estiver, menor a data da expiração se torna

Por que fazer tudo isso? Para tentar reduzir ao máximo o estado do servidor, mas ainda assim prover consistência forte.

PROTOCOLOS DE CONSISTÊNCIA

Descrevem a implementação de um modelo de consistência específico.

- consistência contínua
- protocolos *primary-based*
- protocolos de replicação de escrita

Modo de Operação

- por simplicidade considere apenas um dado, representado por x
- $W(x)$ representa uma operação de escrita em x
- $val(W(x))$ ou simplesmente $val(W)$, representa a valor (numérico) pelo qual x foi atualizado após a operação de escrita W. Por simplicidade assuma que $\forall\ W : val(W) > 0$
- cada servidor S_i tem um log de todas as operações de escrita denotado por L_i
- W é inicialmente enviado para uma das N réplicas. Essa réplica é denotada por $origin(W)$.
- $TW[i,j]$ é o resultado (o valor de x) das *escritas executadas pelo servidor S_i em x que originaram de S_j*:

$$TW[i,j] = \sum\{val(W)|origin(W) = S_j \wedge W \in L_i\}$$

- Temos que $TW[i, i]$ é o valor de x resultado da agregação de todas as mudanças enviadas à S_i
- Nosso objetivo é, a qualquer momento t, manter o valor de x em S_i denotado por v_i com um desvio limitado em comparação com o seu valor real denotado por v

Assim:

Seja v_0 o valor inicial de x

$$v = v_0 + \sum_{k=1}^{N} TW[k, k]$$

Valor v_i de x na réplica i:

$$v_i = v_0 + \sum_{k=1}^{N} TW[i, k]$$

Como $\forall\ W : val(W) > 0$ temos $v_i \leq v$

Problema

Precisamos garantir que $v - v_i < \delta_i$ para todo servidor S_i

Problema

Precisamos garantir que $v - v_i < \delta_i$ para todo servidor S_i

Abordagem

Faça cada servidor S_k manter uma **visão** $TW_k[i, j]$ do que ele acredita ser o valor de $TW[i, j]$. Essa informação pode ser enviada por **gossip** quando uma atualização for propagada.

- Em outras palavras, S_i propaga a atualização recebida de S_j para S_k (na verdade para as réplicas de x que ele conhece) que *armazena* numa tabela o último estado que ele ouviu falar de $TW[i, j]$ a partir do próprio S_i.

Problema

Precisamos garantir que $v - v_i < \delta_i$ para todo servidor S_i

Abordagem

Faça cada servidor S_k manter uma **visão** $TW_k[i, j]$ do que ele acredita ser o valor de $TW[i, j]$. Essa informação pode ser enviada por **gossip** quando uma atualização for propagada.

- Em outras palavras, S_i propaga a atualização recebida de S_j para S_k (na verdade para as réplicas de x que ele conhece) que *armazena* numa tabela o último estado que ele ouviu falar de $TW[i, j]$ a partir do próprio S_i.

Nota:

$$0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j]$$

Solução

S_k envia as operações de seu log para S_i *quando perceber que* $TW_k[i,k]$ *está ficando muito longe de* $TW[k,k]$; em particular quando

$$TW[k,k] - TW_k[i,k] > \delta_i/(N-1)$$

Solução

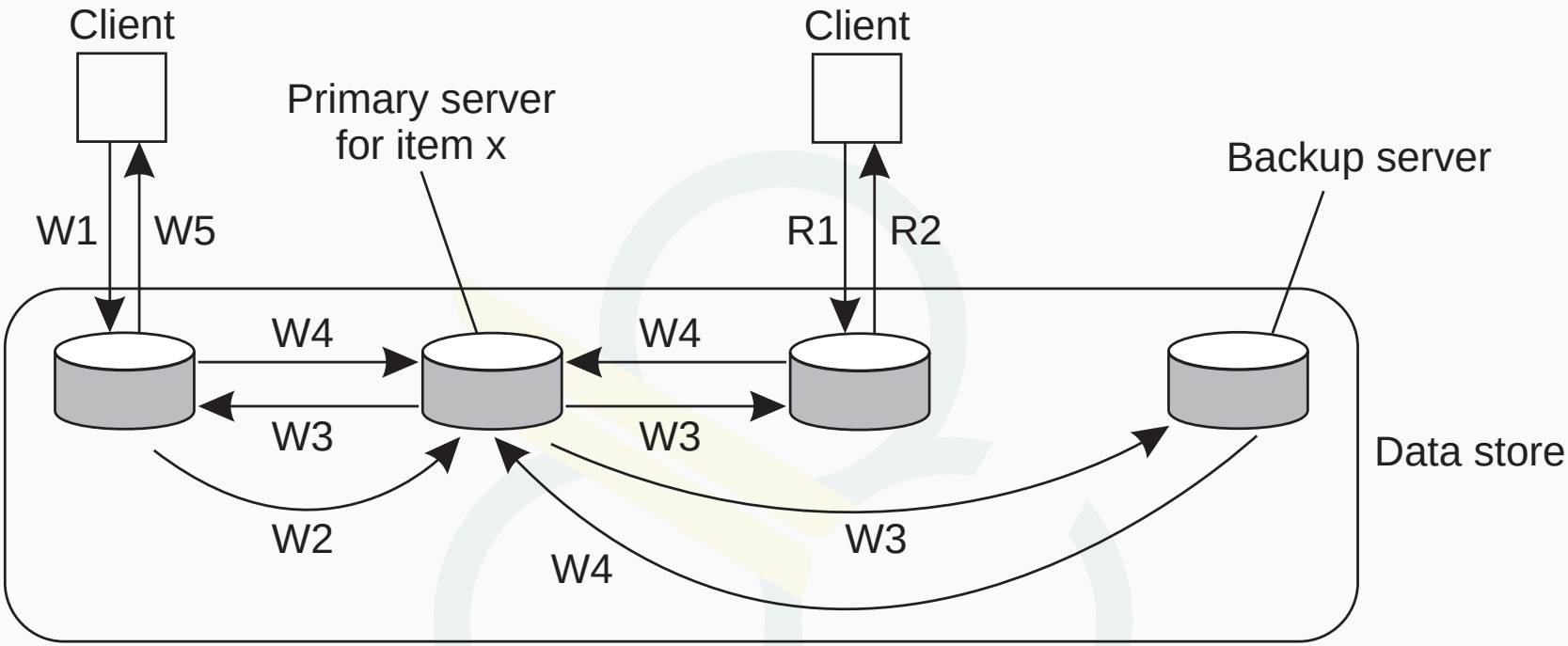
S_k envia as operações de seu log para S_i *quando perceber que* $TW_k[i,k]$ *está ficando muito longe de* $TW[k,k]$; em particular quando

$$TW[k,k] - TW_k[i,k] > \delta_i/(N-1)$$

Nota

A defasagem (*staleness*) pode ser lidada de forma análoga, mantendo no log o que foi visto por último de S_i.

Remote-write backup



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

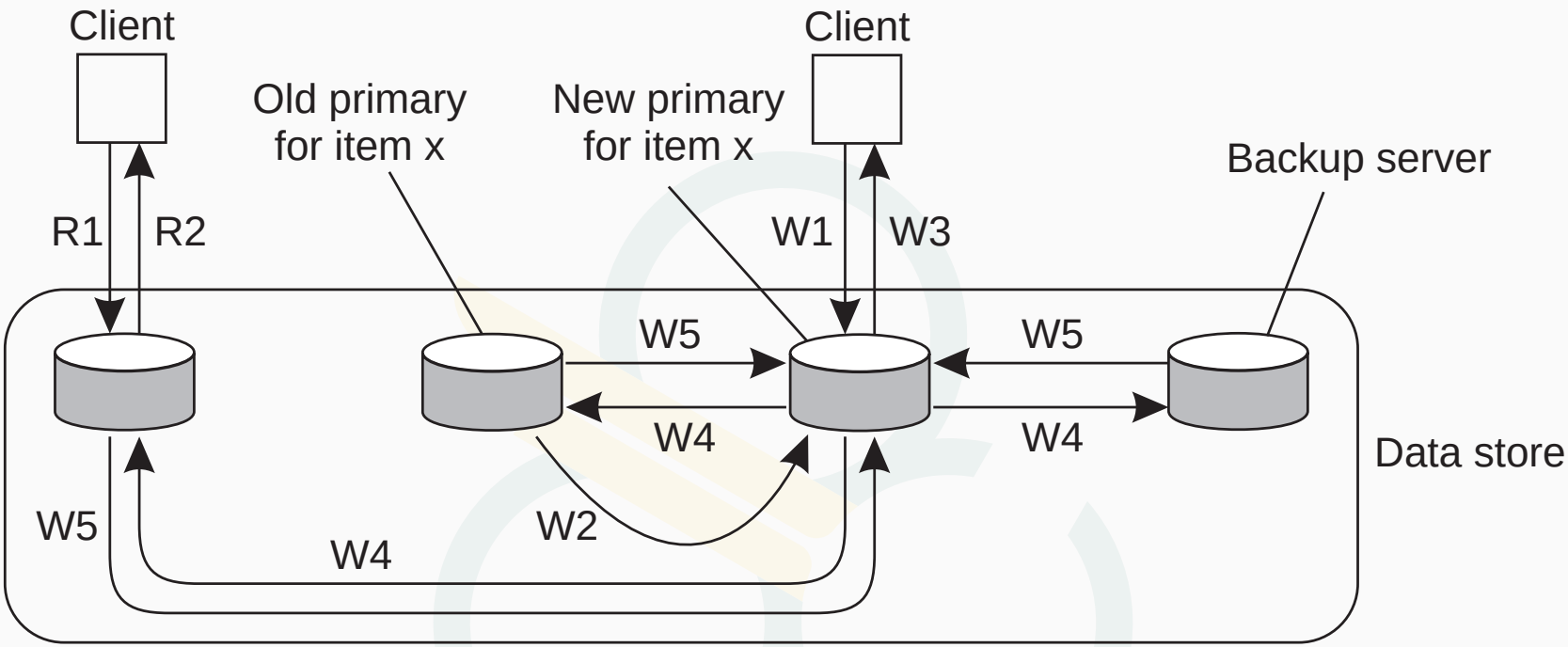
R1. Read request
R2. Response to read

Exemplo de backup com um primary protocol

É tradicionalmente aplicado em bancos de dados distribuídos e sistemas de arquivos que requerem um alto grau de tolerância a falhas. As réplicas são colocadas, em geral, numa mesma LAN.

Garante consistência sequencial

Primary-based protocol 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

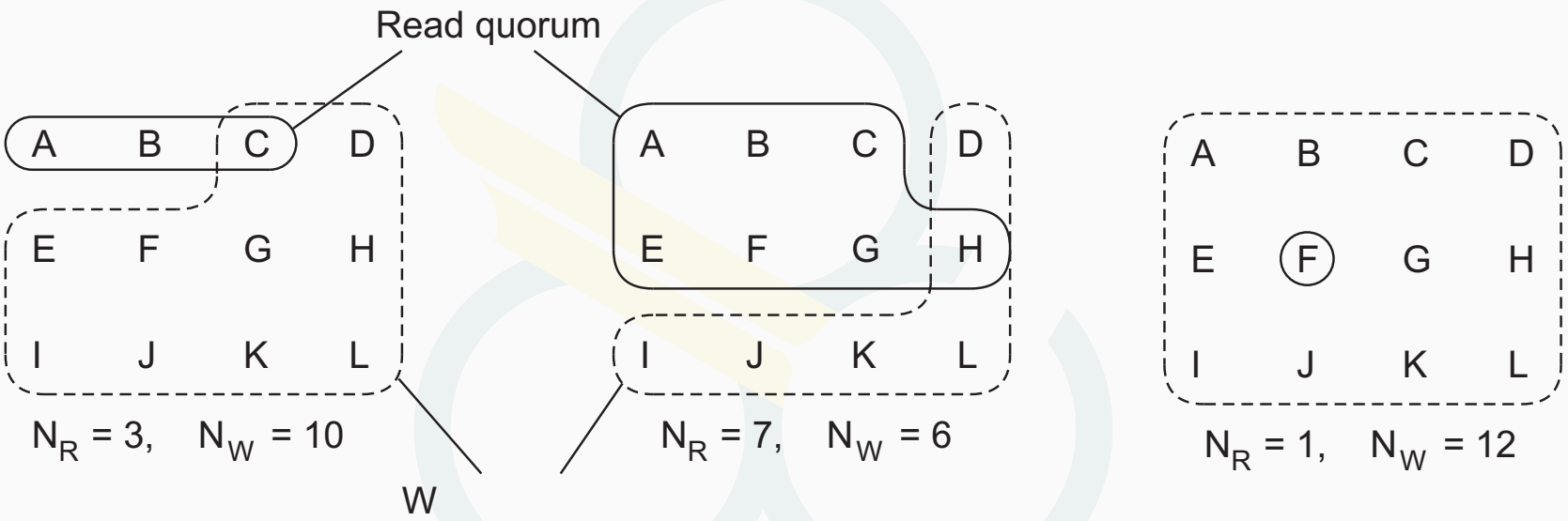
R1. Read request
R2. Response to read

Exemplo de um primary protocol para backup com escritas locais

Computação móvel em modo desconectado (envia todos os arquivos relevantes para o usuário antes do usuário se desconectar e atualiza mais tarde).

Quorum-based protocols

Garante que toda operação é realizada quando existir uma maioria de votos: distingue o **quorum de leitura** do **quorum de escrita**:



necessários: $N_R + N_W > N$ e $N_W > N/2$