ACH 2147 — DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO DISTRIBUÍDOS

ALGORITMOS DE ELEIÇÃO

Daniel Cordeiro

20 e 22 de junho de 2018

Escola de Artes, Ciências e Humanidades | EACH | USP

Princípio

Um algoritmo precisa que algum dos processos assuma o papel de coordenador. A pergunta é: como selecionar esse processo especial dinamicamente?

Nota

Em muitos sistemas o coordenador é escolhido manualmente (ex: servidores de arquivos). Isso leva a soluções centralizadas com um ponto único de falha.

Perguntas

1. Se um coordenador é escolhido dinamicamente, até que ponto podemos dizer que o sistema será centralizado e não distribuído?
2. Um sistema inteiramente distribuído (ou seja, um sem um coordenador) é sempre mais robusto que uma solução centralizada/coordenada?

HIPÓTESES BÁSICAS

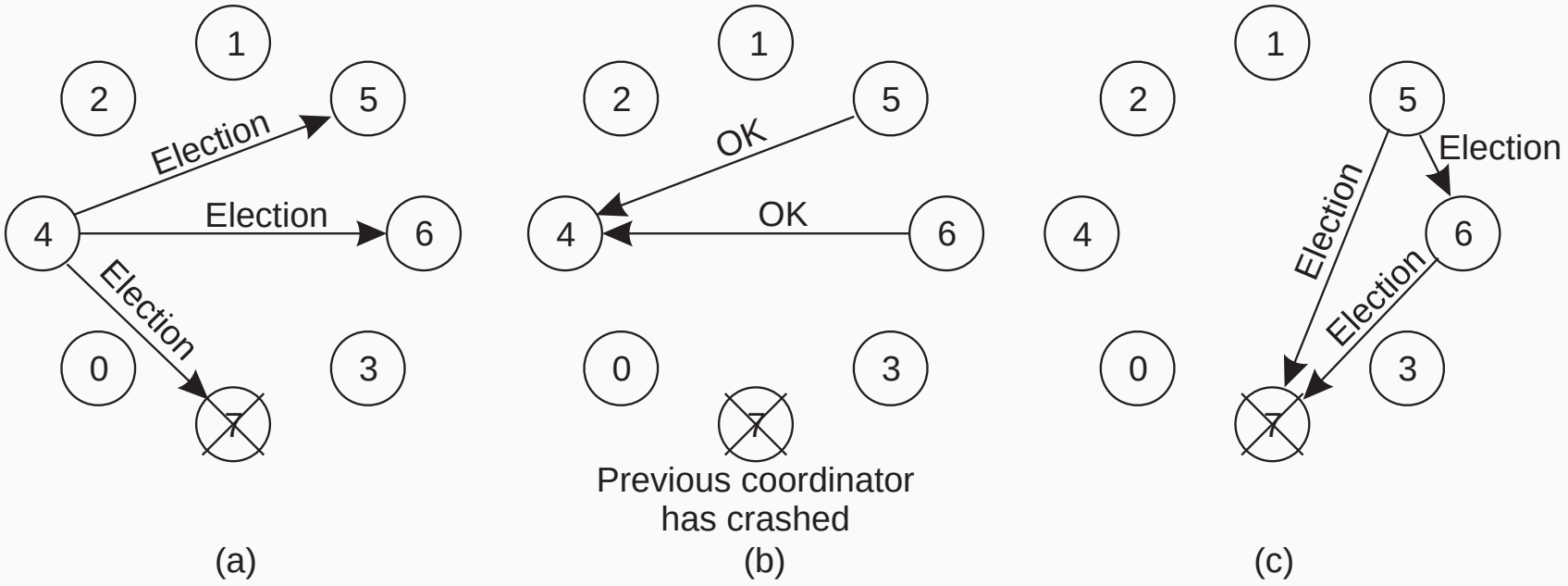
- Todos os processos possuem um `id` único
- Todos os processos conhecem os `id`s de todos os outros processos no sistema (mas eles não tem como saber se os nós estão funcionando ou não)
- A eleição significa identificar o processo de maior `id` que está funcionando em um dado momento

Princípio
Considere N processos $\{P_0, \ldots, P_{N-1}\}$ e seja $id(P_k) = k$. Quando um processo P_k perceber que o coordenador não está mais respondendo às requisições, ele começa uma nova eleição:

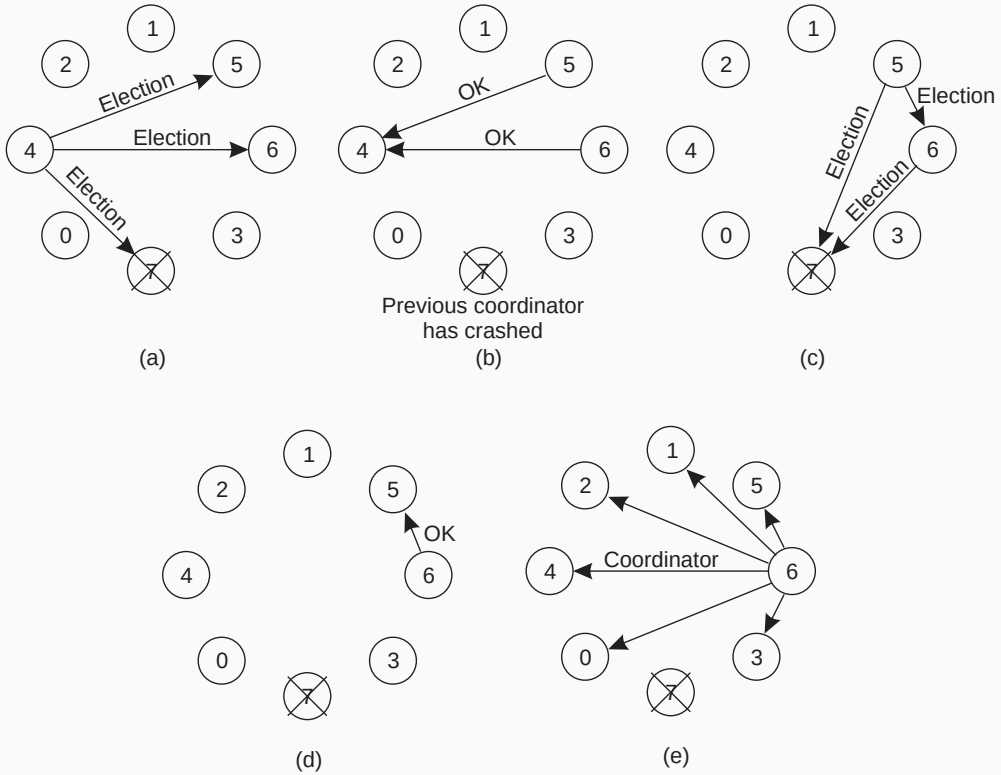
1. P_k envia uma mensagem `ELECTION` para todos os processos com identificadores maiores que o seu: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
2. Se ninguém responder, P_k ganha a eleição e se torna o coordenador
3. Se um dos nós com maior id responder, esse assume[1] a eleição e o trabalho de P_k termina.

[1] O maior sempre ganha, por isso o nome de “algoritmo do valentão”. 😜

ALGORITMO DE ELEIÇÃO — "BULLY"



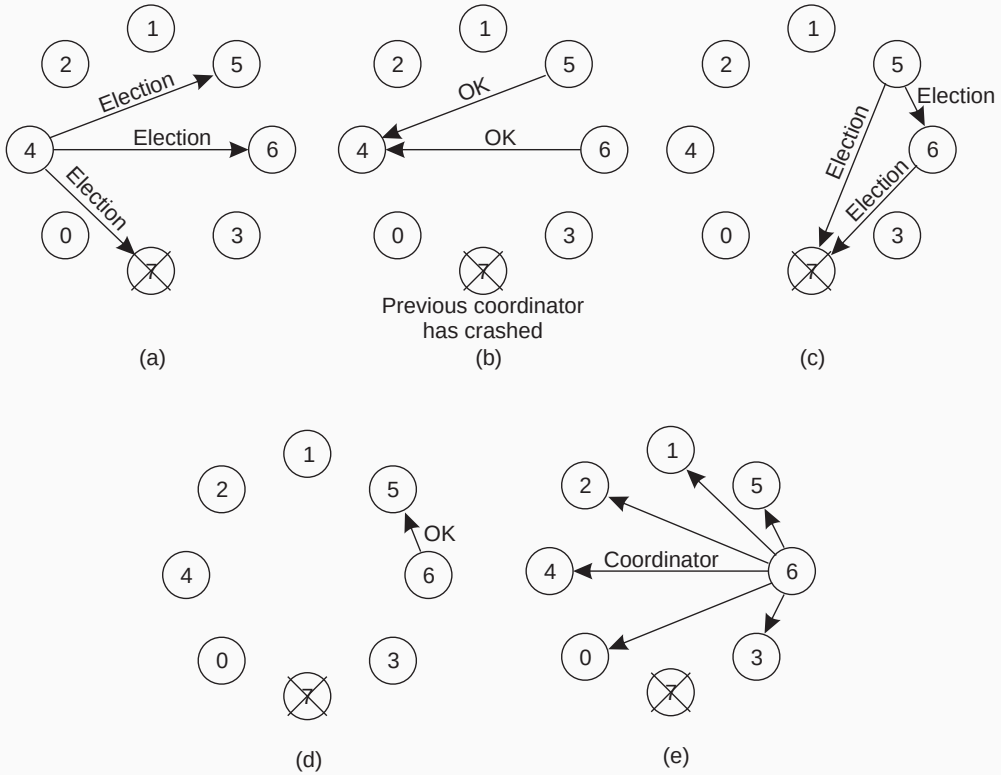
ALGORITMO DE ELEIÇÃO — “BULLY”



Cuidado

Estamos assumido algo importante aqui. O quê?

ALGORITMO DE ELEIÇÃO — “BULLY”



Cuidado

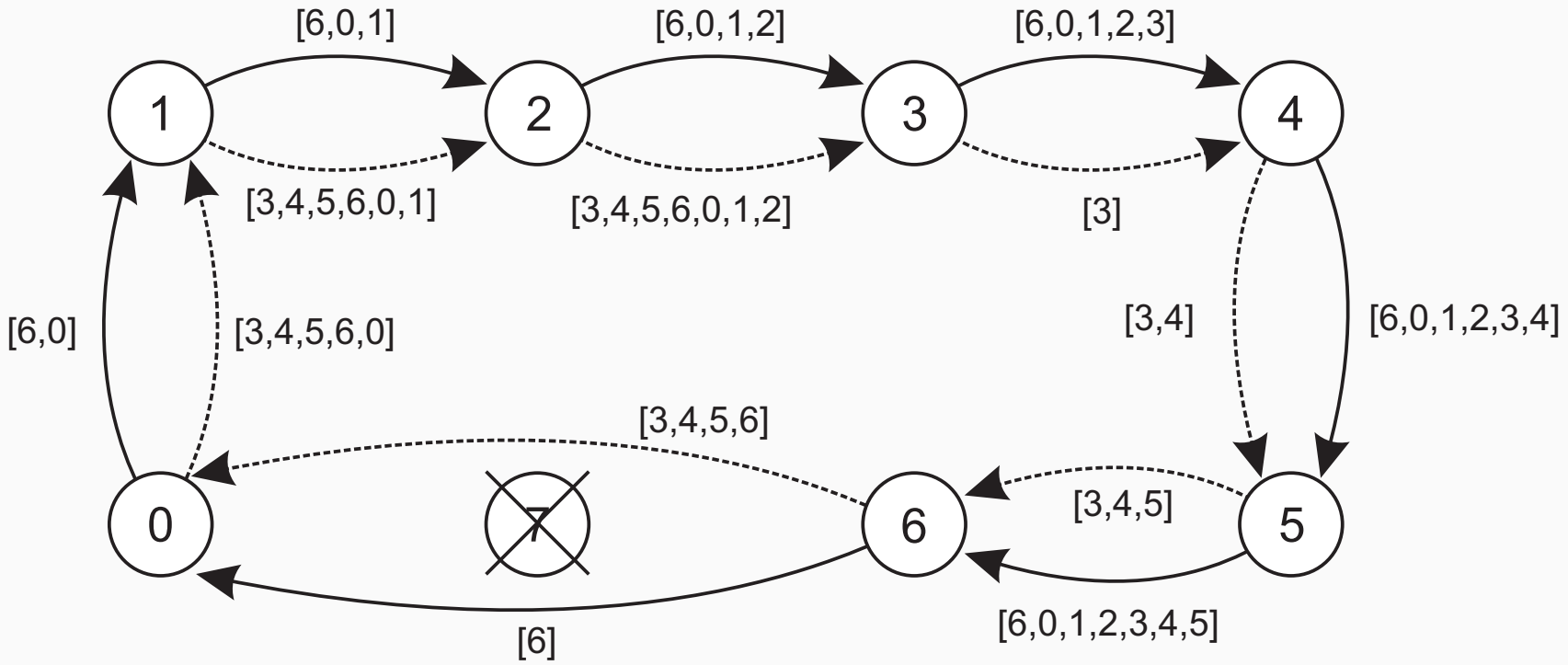
Estamos assumido algo importante aqui. O quê?

Assumimos que a comunicação é confiável

Princípio
A prioridade dos processos são obtidas organizando-os em um anel (lógico). Processos com prioridade mais alta devem ser eleitos como coordenador.

- qualquer processo pode iniciar a eleição ao enviar uma mensagem de eleição ao seu sucessor. Se um sucessor estiver indisponível, a mensagem é enviada ao próximo sucessor
- se uma mensagem for repassada, o remetente se adiciona na lista. Quando a mensagem voltar ao nó que iniciou, todos tiveram a chance de anunciar a sua presença
- o nó que iniciou circula uma mensagem pelo anel com a lista de nós “vivos”. O processo com maior prioridade é eleito coordenador

ELEIÇÃO EM UM ANEL



- As linhas contíguas mostram as mensagens da eleição iniciada por P_6
- As linhas pontilhadas se referem a eleição iniciada por P_3

ELEIÇÃO DE UM SUPERPEER

Como escolher um nó para ser um superpeer de forma que:

- nós normais acessem o superpeer com pouca latência
- superpeers sejam distribuídos homogeneamente por toda a rede de *overlay*
- seja mantida uma fração pré-definida de superpeers em relação ao número total de nós
- cada superpeer não deve ter que servir a mais de um número fixo de nós normais

DHTs

Reserve uma parte do espaço de IDs para os superpeers. **Exemplo:** se S superpeers são necessários em um sistema que usa identificadores de m-bits, reserve os $k = \lceil \log_2 S \rceil$ bits mais à esquerda para os superpeers. Em um sistema com N nós, teremos, em média, $2^{k-m}N$ superpeers.

Roteamento para superpeers

Envie uma mensagem para a chave p para o nó responsável por p AND $\underbrace{11\cdots11}_{k}\underbrace{00\cdots00}_{m-k}$.

Modelos de consistência

Consistência & Replicação

- 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

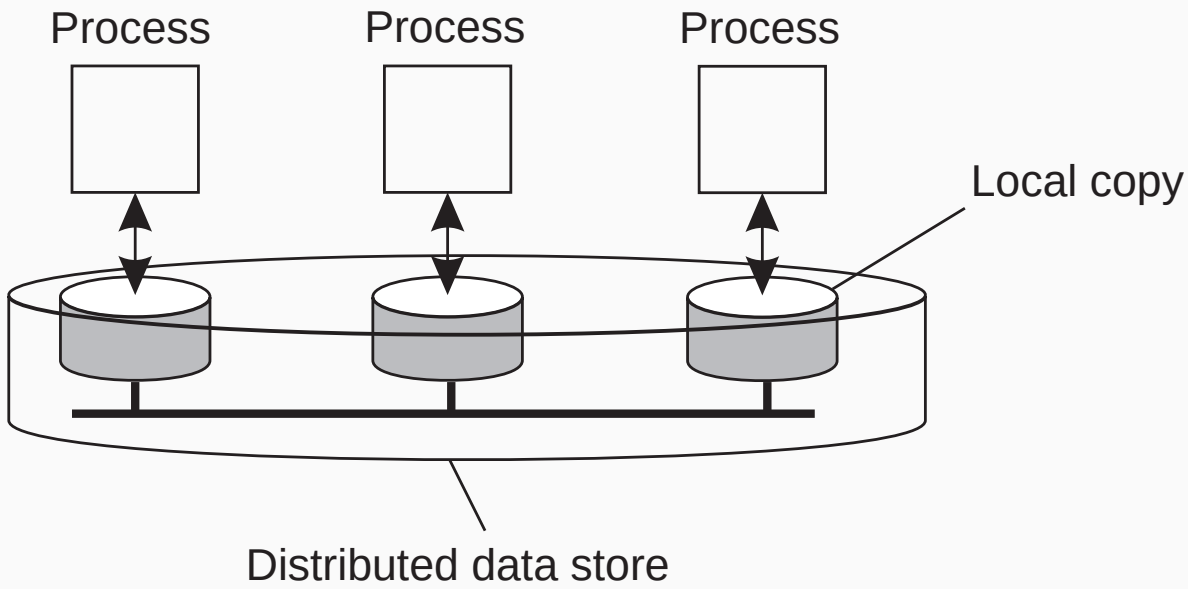
Modelos de consistência centrados em dados

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

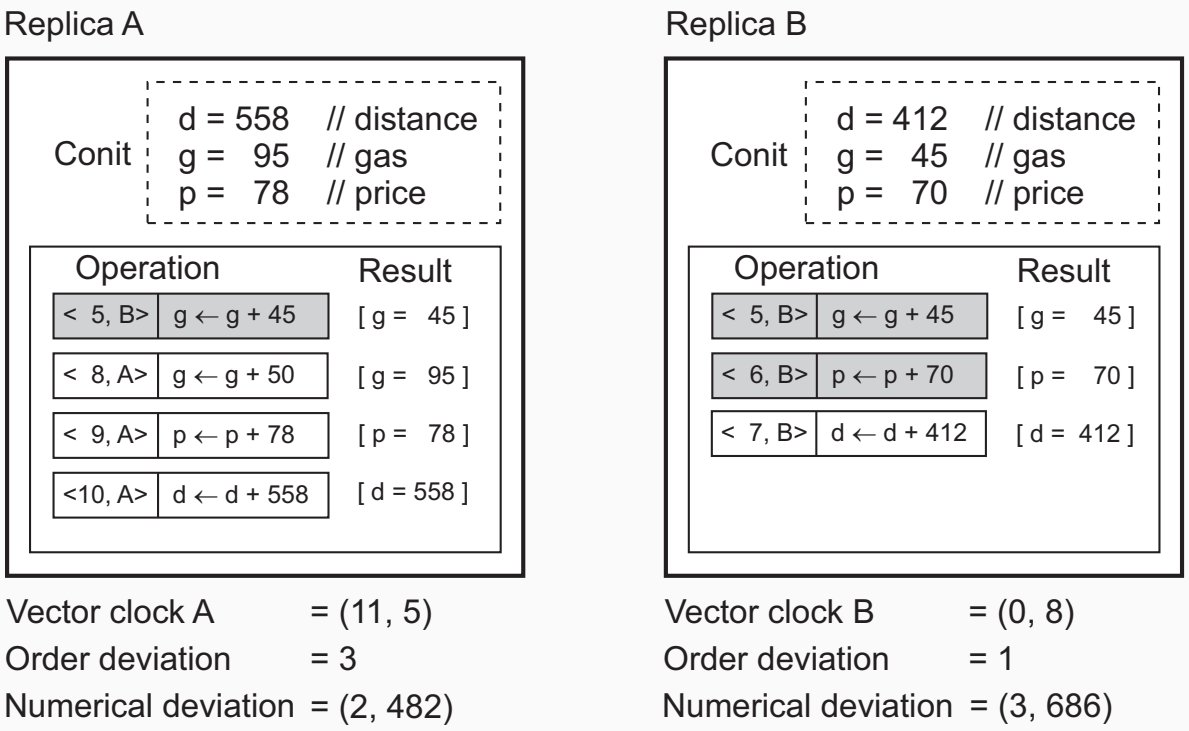


Observação
Podemos considerar diferentes **graus de consistência**:

- 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

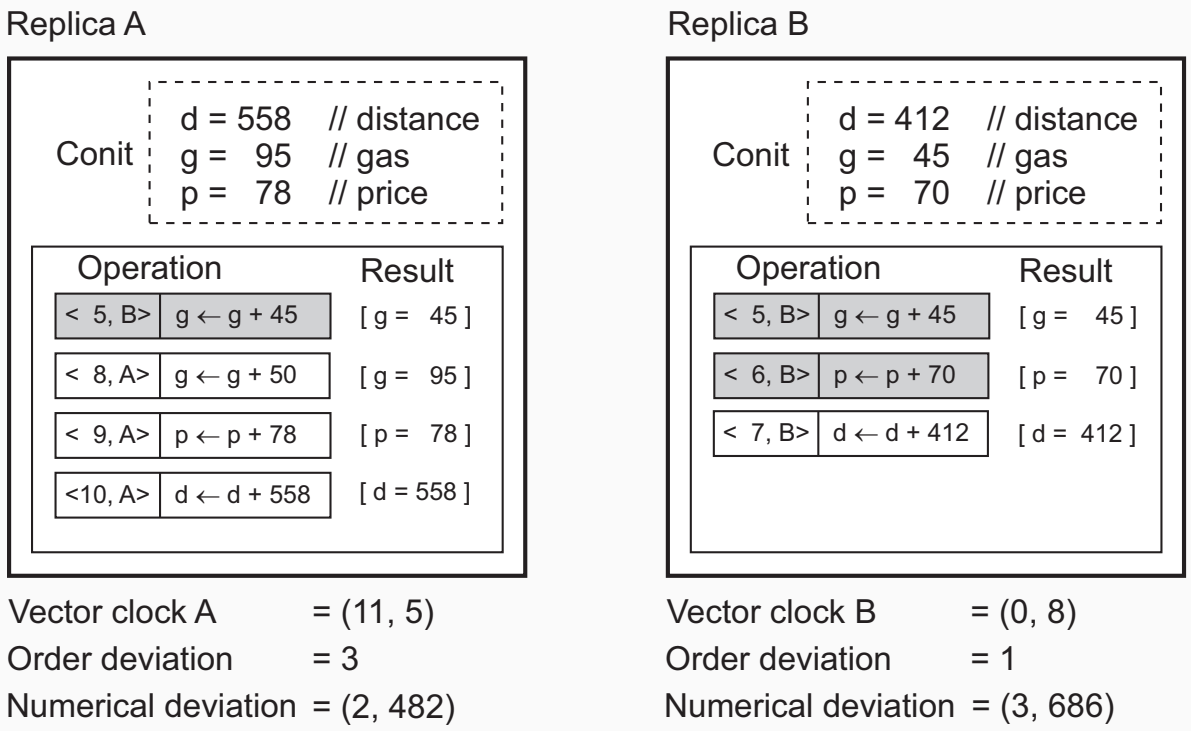
EXEMPLO: CONIT



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*)

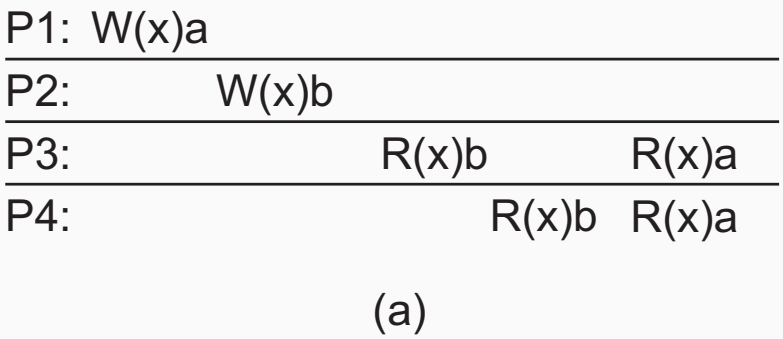
EXEMPLO: CONIT



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)

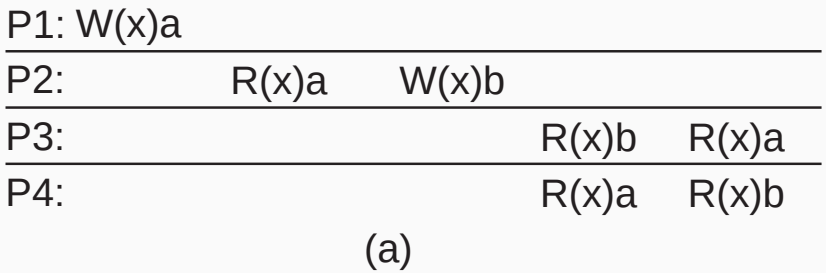
Definição
O resultado de qualquer execução é o mesmo, como se as operações de todos os processos fossem executadas na mesma ordem sequencial e as operações de cada processo aparecer nessa sequência na ordem especificada pelo seu programa.



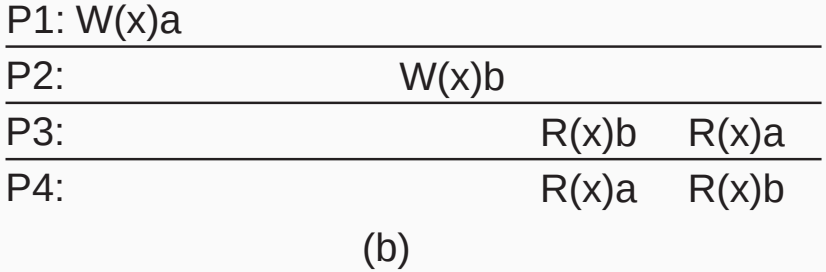
(a) é um *data store* com consistência sequencial; (b) não apresenta consistência sequencial

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)



(b)

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

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.

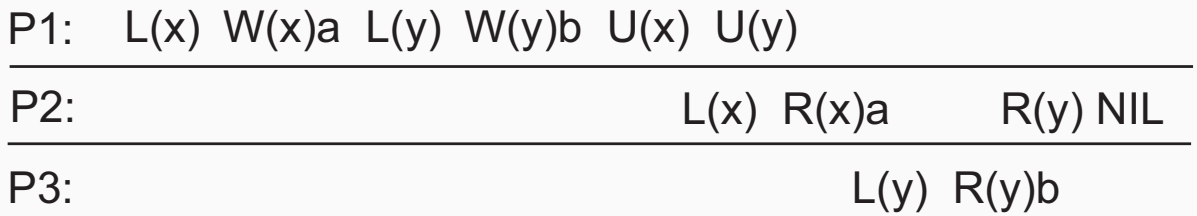


Figura: Um sequência de eventos que respeita a consistência de entrada.

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

- 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

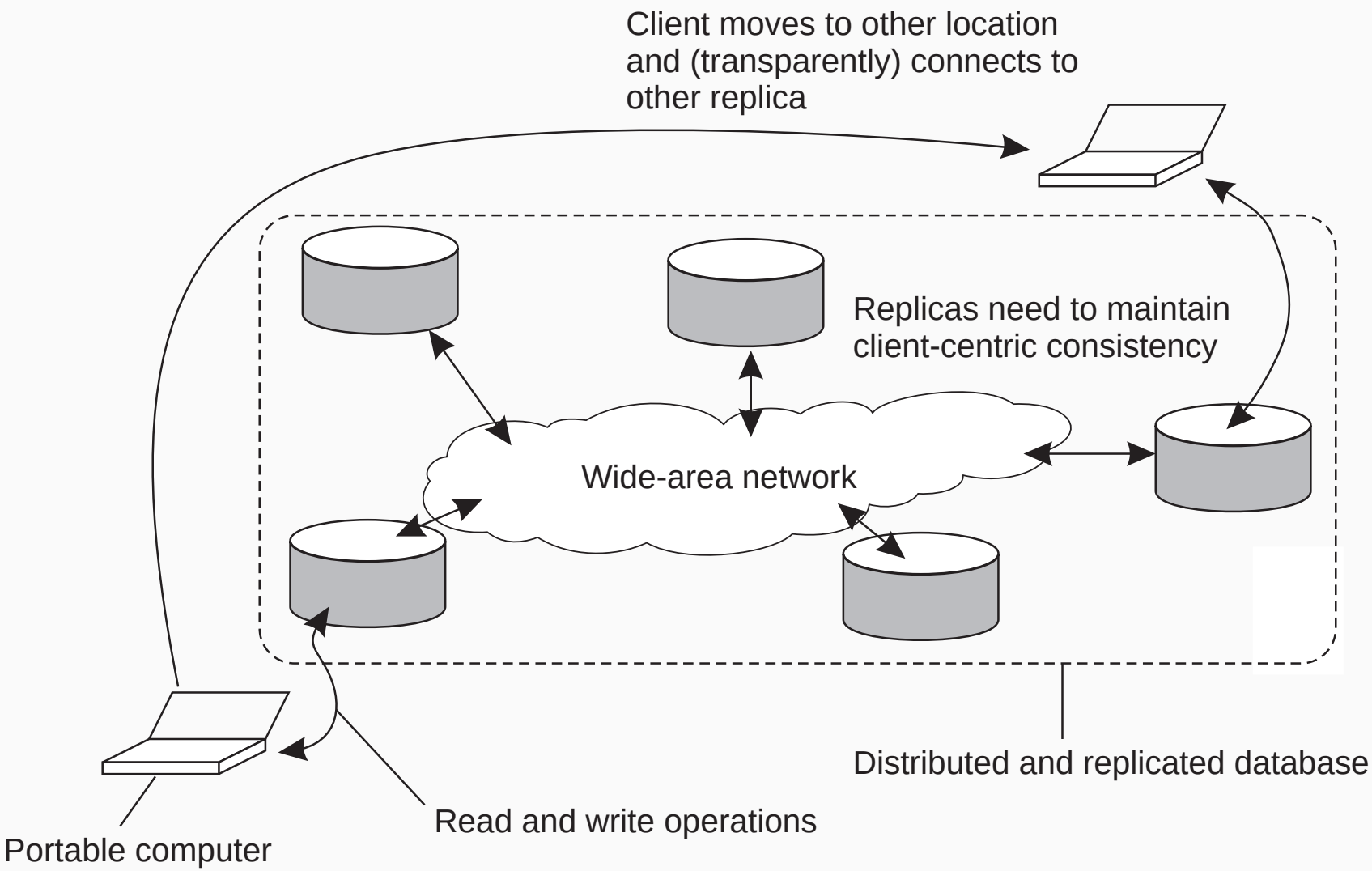
Considere um sistema de banco de dados distribuídos no qual você tem acesso pelo seu notebook. Assuma que seu notebook seja o *front end* do seu banco de dados.

- 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 àquelas em *A*

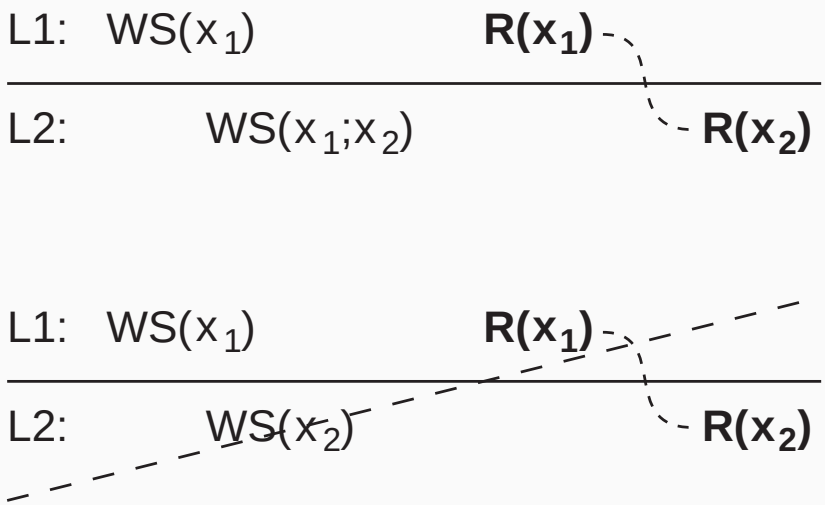
Observação

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

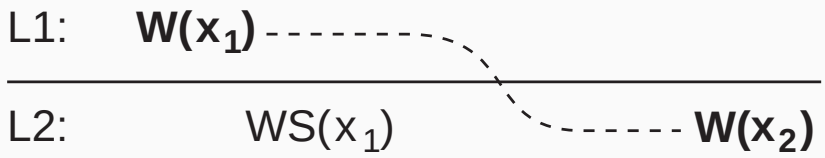
Exemplo

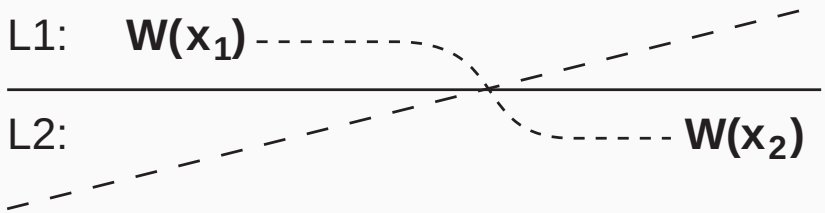
Leituras automáticas das atualizações em seu calendário pessoal vindas de diferentes servidores. Leituras monotônicas garantem que o usuário veja todas as atualizações, independentemente do servidor que originou a leitura

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 $W_k(x_i; x_j)$.

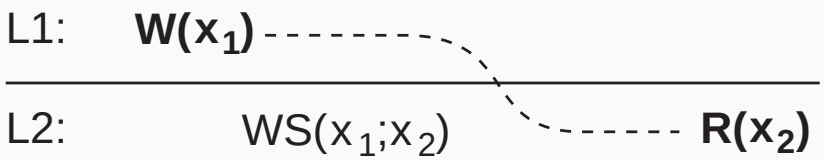
Exemplo

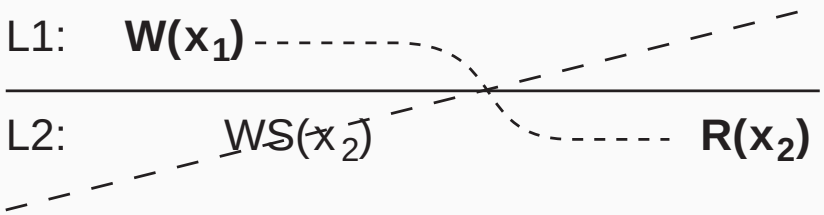
Atualizar um programa no servidor S_2 e garantir que todos os componentes necessários para a compilação também estejam em S_2

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

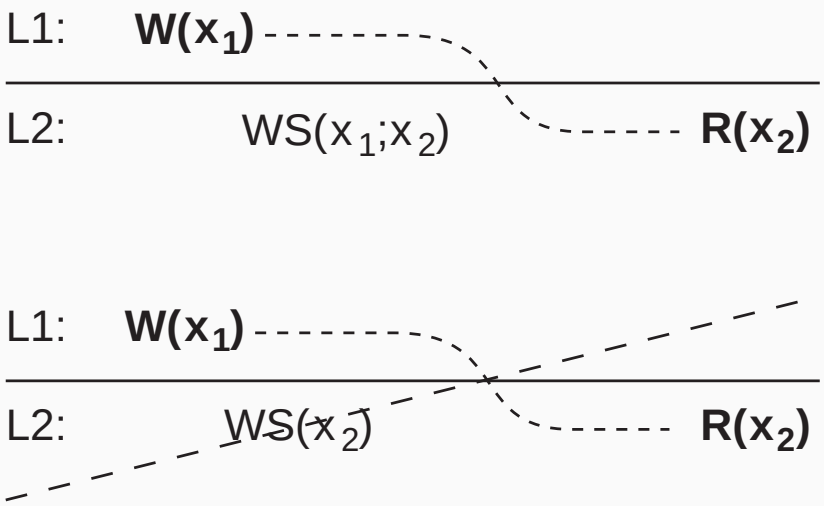
Exemplo
Manter versões de arquivos replicados na ordem correta em todos os lugares (propagando as versões antigas para o servidor onde a versão mais nova está instalada)

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.



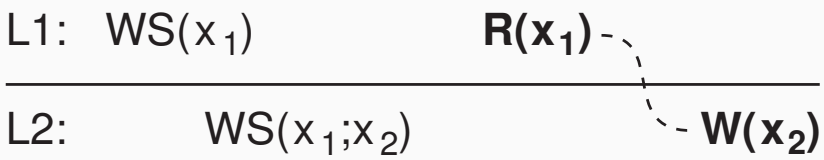


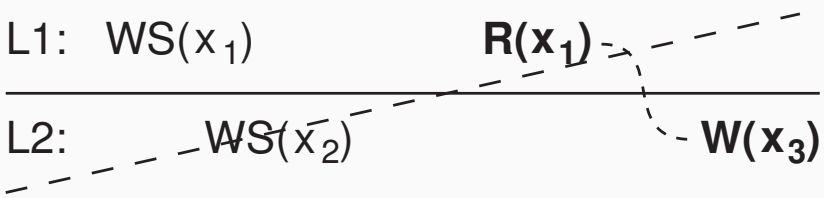
Definição
O efeito de uma operação de escrita realizada por um processo no item *x* sempre será visto por operações de leituras de *x* pelo mesmo processo.



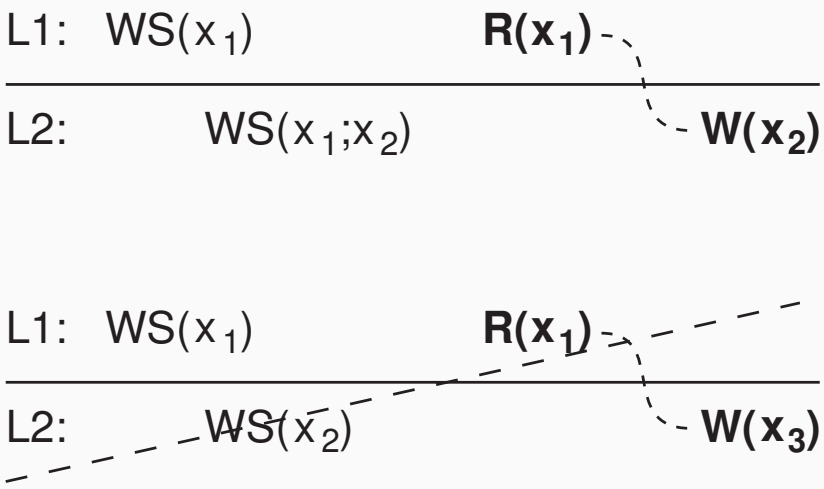
Exemplo
Atualizar sua página web e garantir que o navegador web mostre a versão mais nova ao invés de mostrar a versão em cache

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

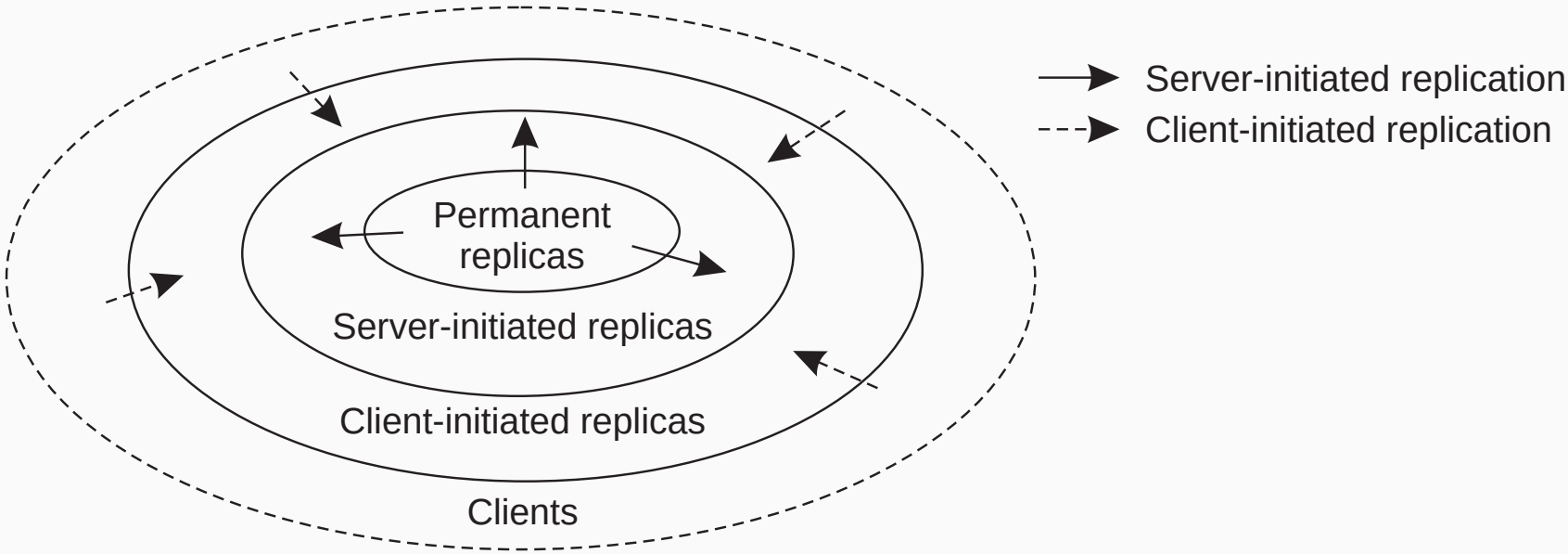
- iterativamente selecionar as melhores posições de $N - K$ para as quais a distância média até os clientes é mínima e então escolher o próximo melhor servidor (a primeira posição escolhida é a que minimiza a distância média até todos os clientes). **Computacionalmente caro**
- selecionar o K-ésimo maior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

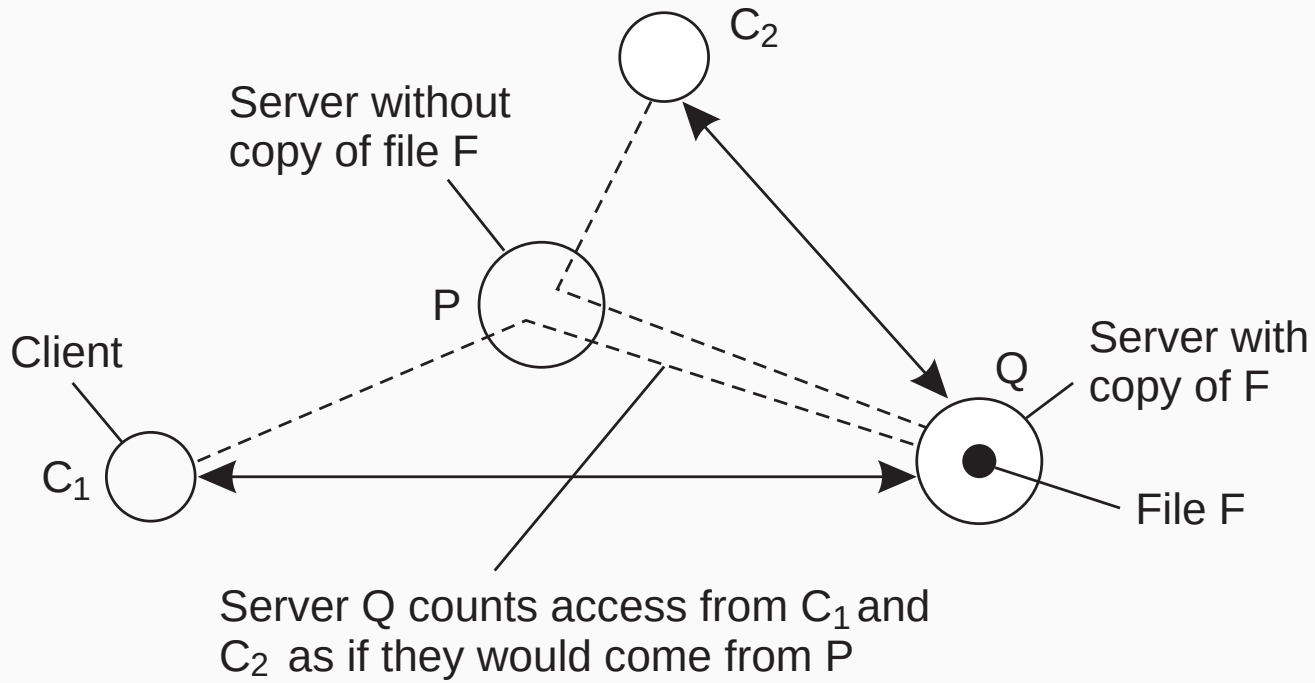
Um processo é capaz de hospedar uma réplica de um objeto ou dado:

- **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.