Sistemas Distribuídos
Aula 18

Roteiro

- Replicação
- Conflitos
- Modelos de consistência
- Modelos de consistência no cliente

Replicação de Dados

- Por que replicar dados em um sistema?

- **Desempenho**: permite reduzir tempo de resposta
  - menos carga, acesso local
- **Confiabilidade**: permite recuperar de falhas
  - robustez do sistema

Replicação é fundamental!

- Utilizado amplamente em sistemas reais
- Ex. CDNs, DNS (*root servers*), NTP, BitTorrent, etc

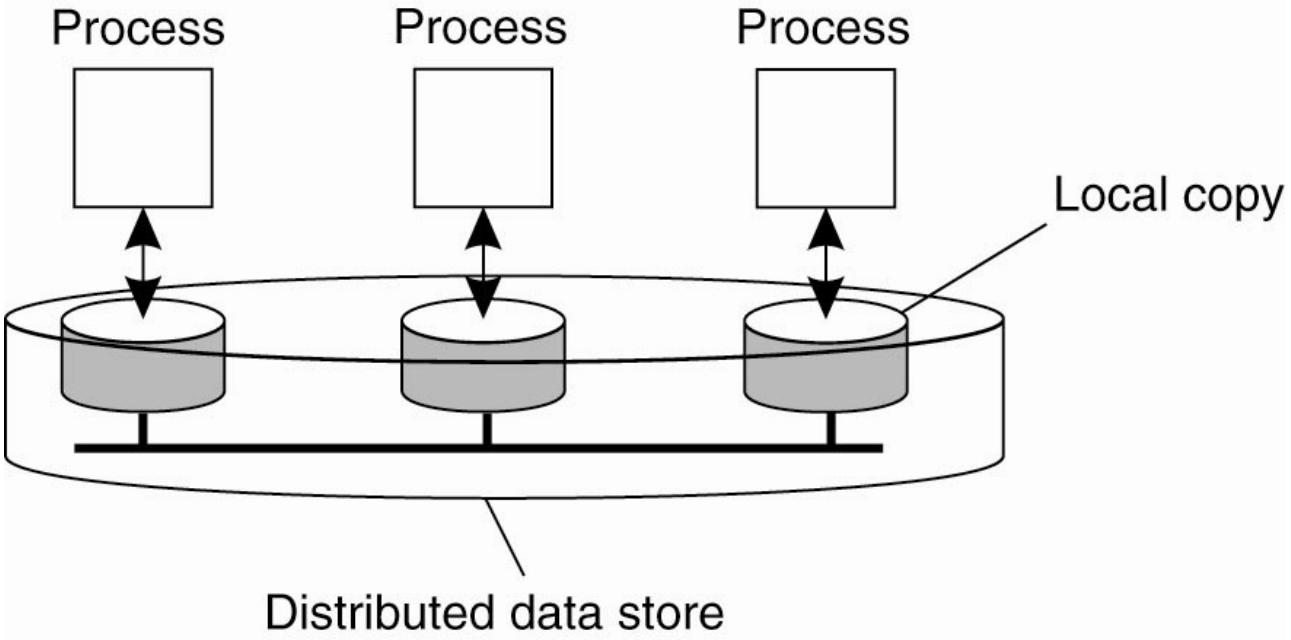
Problemas ao Replicar

- Replicação tem custos
  - monetário: custo do hardware adicional
  - computacional: aumento da complexidade
- Problema fundamental que surge ao replicar dados

Consistência!

- Diferentes réplicas precisam ser atualizadas
- Conflito entre operações de leitura e escrita
- Sistema precisa oferecer algum tipo de consitência dos dados

Modelo Clássico de Replicação



- Diferentes processos acessam réplica local
- Leitura/escrita feita nas réplica locais
- Sistema de armazenamento implementa um *modelo de consistência*
  - oferece algum tipo de consistência entre as réplicas locais para os processos

Conflitos ao Replicar

- Operações de leitura e escrita do mesmo dado podem ocorrer em diferentes processos
- conflito *read-write*
  - um ou mais processo le e outro processo escreve no mesmo dado
- conflito *write-write*
  - dois ou mais processos escrevem no mesmo dado
- Condição de corrida entre as operações nas diferentes réplicas
  - réplicas podem ter valores diferentes para o mesmo dado

Resolvendo Conflitos

- **Ideia 0**: garantir ordenação total das operações de leitura e escrita nas diferentes réplicas
  - todas as réplicas executam as operações na mesma ordem
  - *totally ordered multicast* (já vimos)
- Em geral, não é adequado
  - alto custo (mensagens) compromete escalabilidade
  - alto retardo inviabiliza o uso mais geral
- Sistema pode não precisar de tanta ordem para funcionar adequadamente

Modelo de Consistência

- **Ideia**: sistema implementa um *modelo de consistência* de dados
- Modelo de consistência é um contrato entre aplicação e o armazenamento dos dados
  - define alguma ordem para os conflitos
  - menos estrito que ordenação total das operações
- Em geral, consistência mais estrita → mais ordem (previsibilidade) → mais complexo
- Aplicação tem que ter conhecimento do modelo

Consistência do Sistema

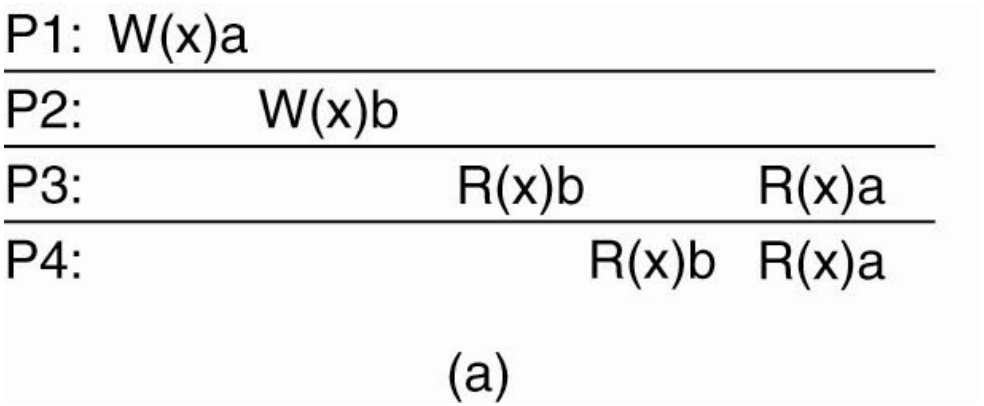
- **Ideia 1**: não oferecer nenhum tipo de consistência
  - ordem das operações é definida pela ordem de chegada nas réplicas
- Baixa complexidade: apenas enviar operações para réplicas
- Dificulta para a aplicação
  - mesmo dado pode ter valores distintos em réplicas diferentes
  - aplicação tem que estar ciente disso!
- Extremo oposto da ordenação total das operações

Consistência Sequencial

- **Ideia**: colocar em sequência as operações de leitura/escrita em cada processo
- Execução válida de uma dada sequência de operações de leitura
  - todos processos observam a mesma sequência ao lerem o mesmo objeto
  - modelo garante isto em todos os processos
- Aplicação tem que estar pronta para as várias possíveis sequências
  - modelo não especifica qual sequência será observada pelos processos

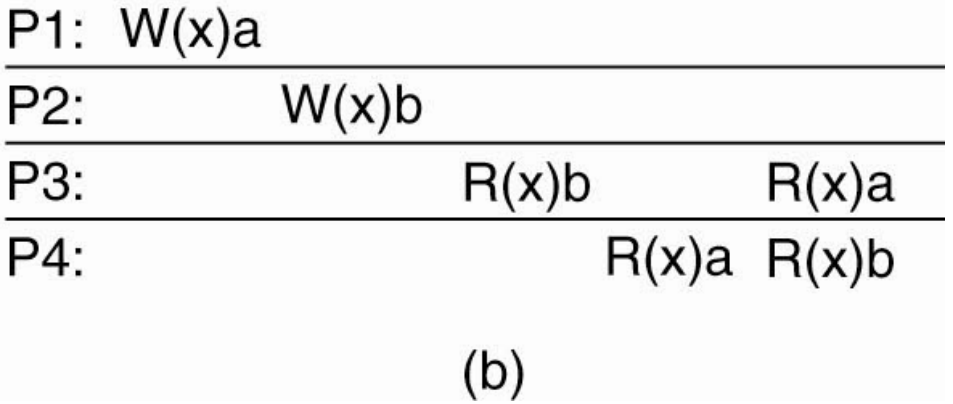
Exemplo

- W(x)a : escreve no endereço x o valor a
- R(y)b : leitura do endereço y retornou b
- 4 processos (cada um com sua réplica local)



(a)

Execução respeita consistência sequencial



(b)

Execução não respeita consistência sequencial (sequência de leitura em P3 é diferente da sequência de leitura em P4)

Exemplo 2

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

- Três processos, três variáveis (inicialmente 0)
- O que pode e não pode ser impresso?
  - sem modelo de consitência, qualquer sequência
- Com consistência sequencial?
- Ex: 11 11 11 ? S
  10 10 11 ? S
  00 00 11 ? N
  11 10 11 ? N

Consistência Causal

- **Ideia**: colocar em sequência as operações que possam possuir causalidade
  - dentro do mesmo processo e entre processos (através de write e read no mesmo objeto)
  - write concorrentes ocorrem em qualquer ordem
- Todos processos veem operações causais na mesma ordem
- Processos diferentes podem ver writes concorrentes em ordem diferente
- Parecido com ordenação parcial induzida por relógios lógicos

Exemplo

- W(x)a : escreve no endereço x o valor a
- R(y)b : leitura do endereço y retornou b
- 4 processos (cada um com sua réplica local)

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

Respeita consistência causal

- P1 escreve em x valor “a”,
- P2, P3, P4 leem x com valor “a”
  - evento escrita ocorre antes
- P1 e P2 escrevem em x valor “c” e “b”
  - writes concorrentes podem ser vistos em qq ordem

Exemplo 2

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

Não respeita consistência causal

P3 não pode ler o valor “b” e depois ler “a”, pois P2 escreve “b” em x depois do valor ter lido “a”

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

Respeita consistência causal

writes concorrentes (por P1 e P2) podem ser lidos em qq ordem pelos outros processos

Focando no Cliente

- Muitas aplicações possuem modelo de acesso aos dados restrito
  - muitas operações de leitura
  - poucas operações de escrita distribuídas (maioria ocorre em um processo)
  - cliente define escopo dos dados
- Exemplos
  - webmail, whatsapp, Dropbox
- Consequência: redução (eliminação) dos conflitos read-write e write-write

Exemplo com Dropbox

- Usuário Dropbox usando dois computadores
  - cada computador possui cópia local (réplica)
- Trabalha em casa (C1) nos arquivos
- Vai para a UFRJ e trabalha (C2) nos arquivos

- **O que é de fato importante?**
- Arquivo F modificado em C1 esteja atualizado em C2 quando F for acessado em C2
  - se F não for acessado em C2 usuário não sabe da “inconsistência”
  - não precisa atualizar C2 instantaneamente

Exemplo com GMail

- Usuário GMail acessa sua conta via aplicativo em smartphone
- Lê emails no Rio, pega um voo para Paris, acessa emails em Paris
  - Gmail tem servidores replicados
- **O que é de fato importante?**
- Caixa de entrada em Paris é ao menos tão atual quanto caixa de entrada no Rio
  - usuário não sabe que emails que ainda não estavam no Rio também não estão em Paris!

Consistência eventual

Consistência Centrada no Cliente

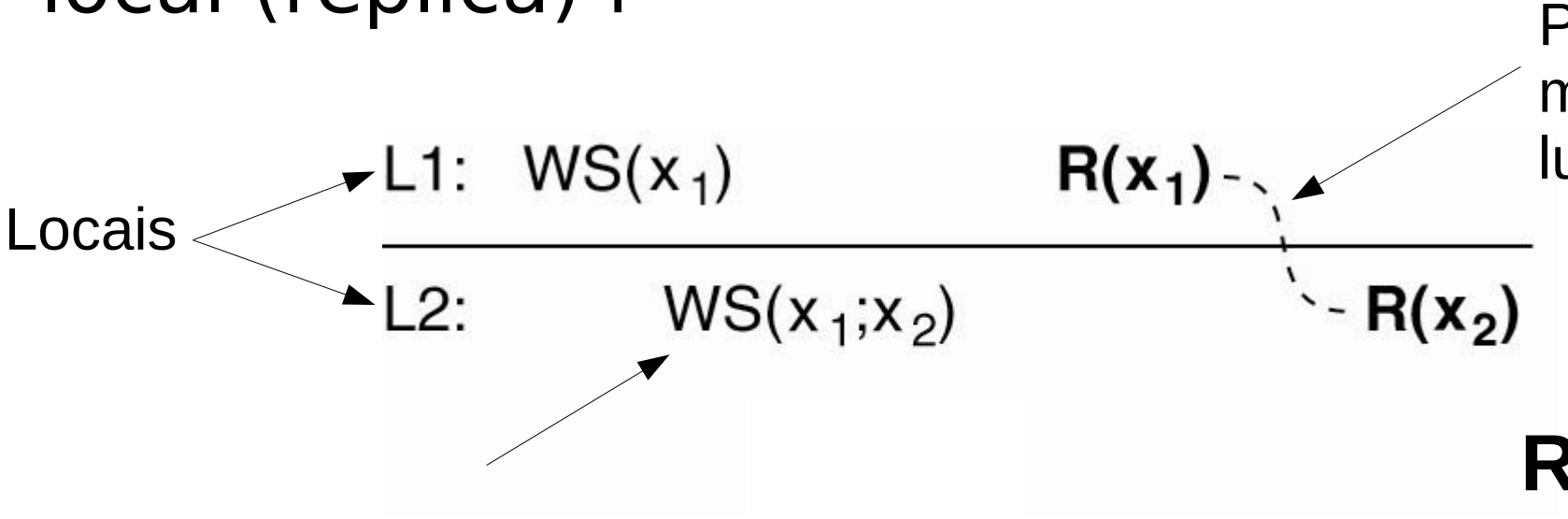
- Modelos anteriores são contrato da aplicação com o sistema (*system-wide*)
  - para todos os processos
- Mais difíceis de garantir, maior complexidade de implementação
- Modelo de consistência centrado no cliente
  - contrato de consistência com o cliente
  - mais fácil de ser implementado
- Cliente não é tão rígido
  - ex. tolera atualização eventual dos dados

Reads Monotônico

- Cliente (processo) faz operação de leitura/escrita em diferentes locais (réplica)
  - ex. app do seu smartphone acessando o Gmail enquanto você viaja pela mundo
- Modelo de consitência “read monotônico”
  - se processo P ler valor em x, próximas leituras por P de x devem ter valores iguais ou mais recentes
- “mais recente” definido por causalidade ou com relógio real
- Garante que sequência de leituras do cliente não voltam atrás

Exemplo de Reads Monotônico

- Único processo P acessa dados mudando de local (cada local possui sua réplica)
- $WS(x_i)$: write (pelo sistema) no endereço x no local (réplica) i



Respeita!

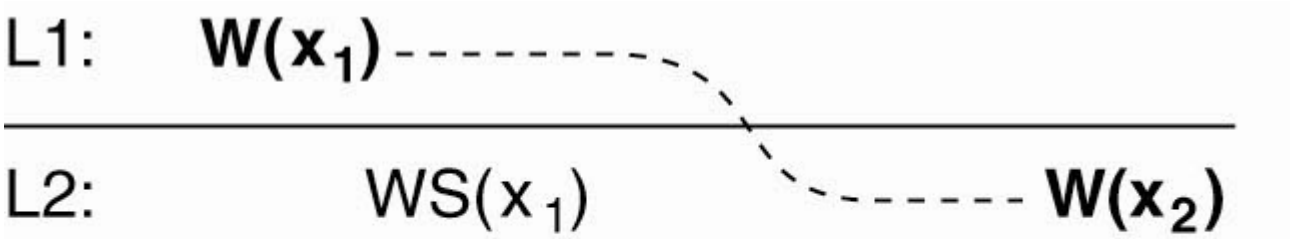
Write em x_2 que ocorre depois de write em x_1

Writes Monotônico

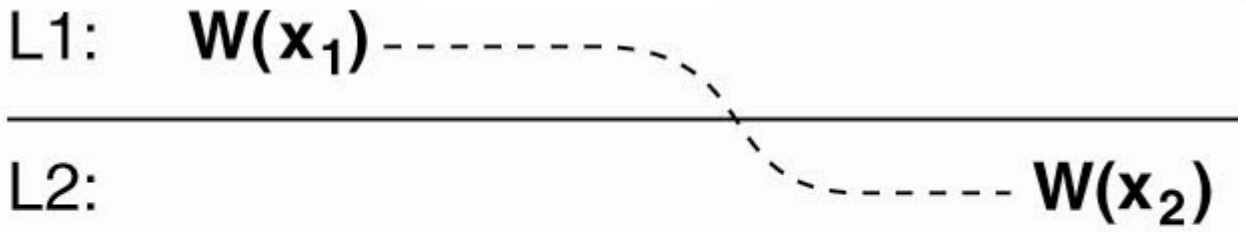
- Modelo de consitência “writes monotônico”
  - operações de escrita ocorrem em toda as réplicas na ordem em que são executadas por P
- Assume que não há writes concorrentes
  - apenas um processo P ativo por cliente
  - local do processo pode variar
- Sistema executa operações de escrita nas réplicas

Exemplo de Write Monotonico

- Único processo P acessa dados mudando de local (cada local possui sua réplica)
- $W(x_i)$: write (pelo usuário) no endereço x no local (réplica) i



Respeita



Não respeita

Sistema não escreveu x_1 em L2 antes do cliente escrever x_2