Transações Distribuídas

Sistemas Distribuídos e Programação Concorrente

Prof. MSc. Rodrigo D. Malara

Conteúdo

- Introdução a Transações
  - Tipos, Propriedades, Demarcação
- Implementação de Transações
  - Shadow Versions, Writeahead Log
- Transações Distribuídas
  - Commit de uma fase e de duas fases
- Controle de Concorrência
  - Timestamps e Controle de Concorrência Otimista

Introdução à Transações

- Objetivo
  - Garantir que todos os objetos gerenciados por um servidor permaneçam consistentes mesmo com uma queda do servidor
- Características das Transações
  - São iniciadas pelo cliente
  - Conjunto de operações agrupadas atomicamente
  - Livres da interferência causada por outros clientes
  - Semântica “Tudo ou nada”
  - Duráveis → armazenamento permanente
- Originárias dos SGBDs
  - Objetos são registros (ou tuplas) de uma tabela

Tipos de Transações

- Envolvendo 1 objeto apenas
  - Sincronização simples é suficiente
  - Ex: `synchronized` em Java ou mutex em C
- Envolvendo vários objetos em 1 servidor
  - Transações
- Envolvendo vários objetos em vários servidores
  - Transações distribuídas
    - Ex: JTA – Java Transactions API

Obs: 1 objeto pode estar em apenas um servidor embora seja possível replicá-lo

Propriedades ACID
Harder e Reuter (1983)

- Atomicidade
  - Ou tudo ou nada
- Consistência
  - Mantém o sistema em um estado confiável
- Isolação
  - Efeitos intermediários de uma transação não podem ser percebidos por outras
- Durabilidade
  - Estado final é persistido em memória não-volátil

Demarcação de Transações

- Início
  - Deve ser solicitado pelo cliente: open ou begin
- Término
  - Falha ocorreu: abort ou rollback
    - Caso alguma mudança tenha ocorrido ela é desfeita
  - Sucesso: close ou commit
    - Novo estado é persistido

Demarcação de Transações (2)

Pode ser realizada pelo

- Cliente
  - Início, fechamento e/ou cancelamento presentes no código-fonte do cliente
- Middleware (ex: ORB Corba ou container EJB)
  - Início e término das TXNs são declarados em arquivos de instalação ou anotações no código
  - Permitem mudanças sem manutenção no código
  - Performance é um pouco pior
  - Facilita suporte à transações distribuídas

Implementação de transações

Dois métodos são apresentados:

- ***Shadow Versions***
  - Cria-se cópia dos dados a serem alterados.
  - Alterações são efetuadas nas cópias
  - Ao fim da transação cópia substitui dados antigos.
- ***Writeahead Log*** *ou lista de intenções ou journal*
  - *A*ntes do dado ser modificado grava-se informações em um *Log*
    - Qual a transação está efetuando alterações
    - Os dados que estão sendo alterados
    - Os valores antigos e novos.
  - Após o *Log* ter sido gravado as alterações são feitas.
  - Em caso de falha, é possível fazer o rollback usando *Log*

Transações Distribuídas

- A ação de *commit* deve ser “instantânea” e indivisível.
- Cooperação de muitos processos
  - Máquinas distintas
  - Cada qual com um conjunto de objetos envolvidos na transação
- Um processo é designado como o **coordenador** (normalmente o próprio cliente que inicia a transação)
- Os demais processos são designados como **subordinados**
- Uma transação deve ser abortada caso um dos servidores apresente qualquer tipo de falha
  - Envolve abortar a transação nos servidores que não apresentaram falha
- Deve haver um consenso antes de commitar

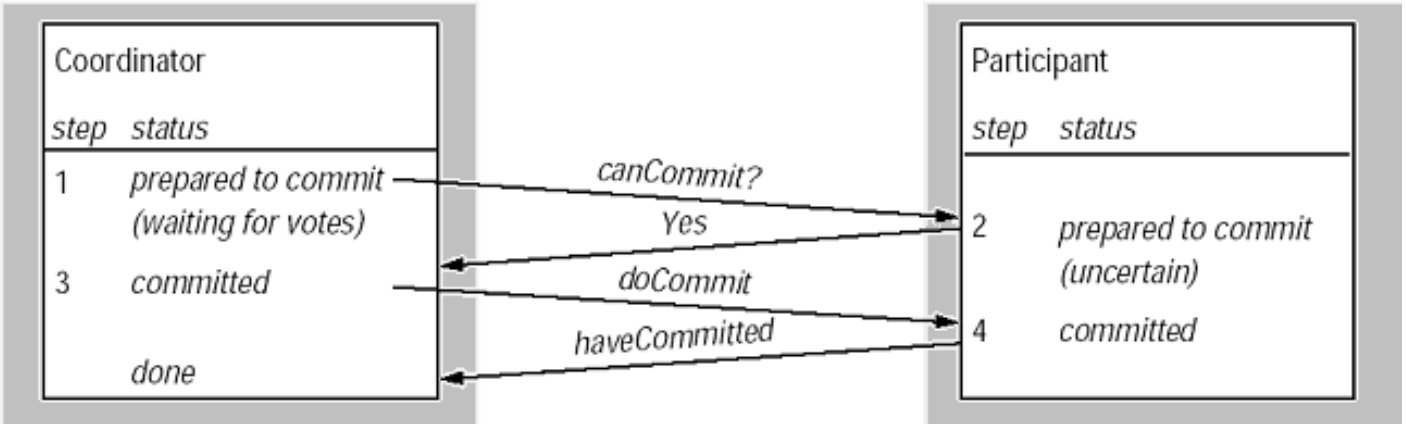
Protocolo de *commit* em uma fase

- Coordenador:
  - Geralmente é o processo cliente
  - Solicita que os participantes iniciem uma TXN
  - Coordena a realização da seqüencia de operações
  - Solicita commit ou abort
- Desvantagem:
  - Não prevê o caso de um dos participantes falhar
    - *Ex: 3 servidores commitam mas 1 falha*
    - *Fere Consistência do ACID*

Commit em 2 fases

- Fase 1 – VOTAÇÃO
  - Coordenador registra “prepare” em *log*
    - *E*nvia a mensagem “prepare” para os subordinados
  - Um subordinado registra “ready” / “abort” em *log*
    - *E*nvia a mensagem “ready” / “abort” para o coordenador
  - O coordenador coleta todos as mensagens “ready”
- Fase 2 - DECISÃO
  - O coordenador registra a decisão em *log*
    - Envia mensagem “commit” / “abort” para os subordinados
  - Um subordinado registra “commit” / “abort” em *log*
    - Toma a ação correspondente
    - Envia mensagem “finished” ao coordenador

Commit em 2 Fases



- *canCommit?(trans)-> Yes / No*
- *doCommit(trans)*
- *doAbort(trans)*
- *haveCommitted(trans, participant)*
- *getDecision(trans) -> Yes / No*

Fonte: G. Coulouris et al., *Distributed Systems: Concepts and Design, Third Edition.*

Commit em 2 Fases: Problemas

- Se o coordenador cair:
  - Dados ficarão bloqueados até que o coordenador esteja novamente disponível
  - *"presumed abort"*: se o coordenador cair antes de um participante estar preparado, o participante pode abortar sem esperar pela sua volta
- Esperas Longas – o que fazer ?
  - Coordenador a espera dos votos dos participantes
  - Participante aguardando o canCommit?(T) do coordenador ou
  - Participante aguardando o doCommit(T) or abortCommit(T) do coordenador
  - Participante envia um getDecision(T) para o coordenado
- Complexidade do Algoritmo: exige no mínimo 3(N-1) mensagens para completar a transação

Controle de Concorrência

- Objetivos:
  - Garantir a propriedade de isolamento
  - Garantir que os dados continuem consistentes após acessados por transações concorrentes
    - Ex: Duas transações acessando a mesma conta bancária
  - Situação Ideal: Acesso serializado aos recursos compartilhados
- Normalmente, implementado como um algoritmo de duas fases:
  - Bloqueio de Duas Fases (*two-phase locking*)
  - Variação do Algoritmo do Anel (Exclusão Mútua – Cap. 12)

Bloqueio em Duas Fases

- O algoritmo two-phase locking possui as seguintes fases:
  - 1ª fase: obtenção de *locks*
  - 2ª fase: liberação de *locks <ACERTAR>*
- *Strict two-phase locking*
  - *Bloqueio em duas fases estrito*
  - *Locks (travas) adquiridos são mantidos até que a transação seja cancelada ou confirmada*
  - Vantagem: impede os "*cascading aborts*"
    - Rollback de diversas transações devido às mesmas terem compartilhado de um objeto cujas modificações foram canceladas
  - Desvantagem: *deadlocks*

Controle de Concorrência (2)

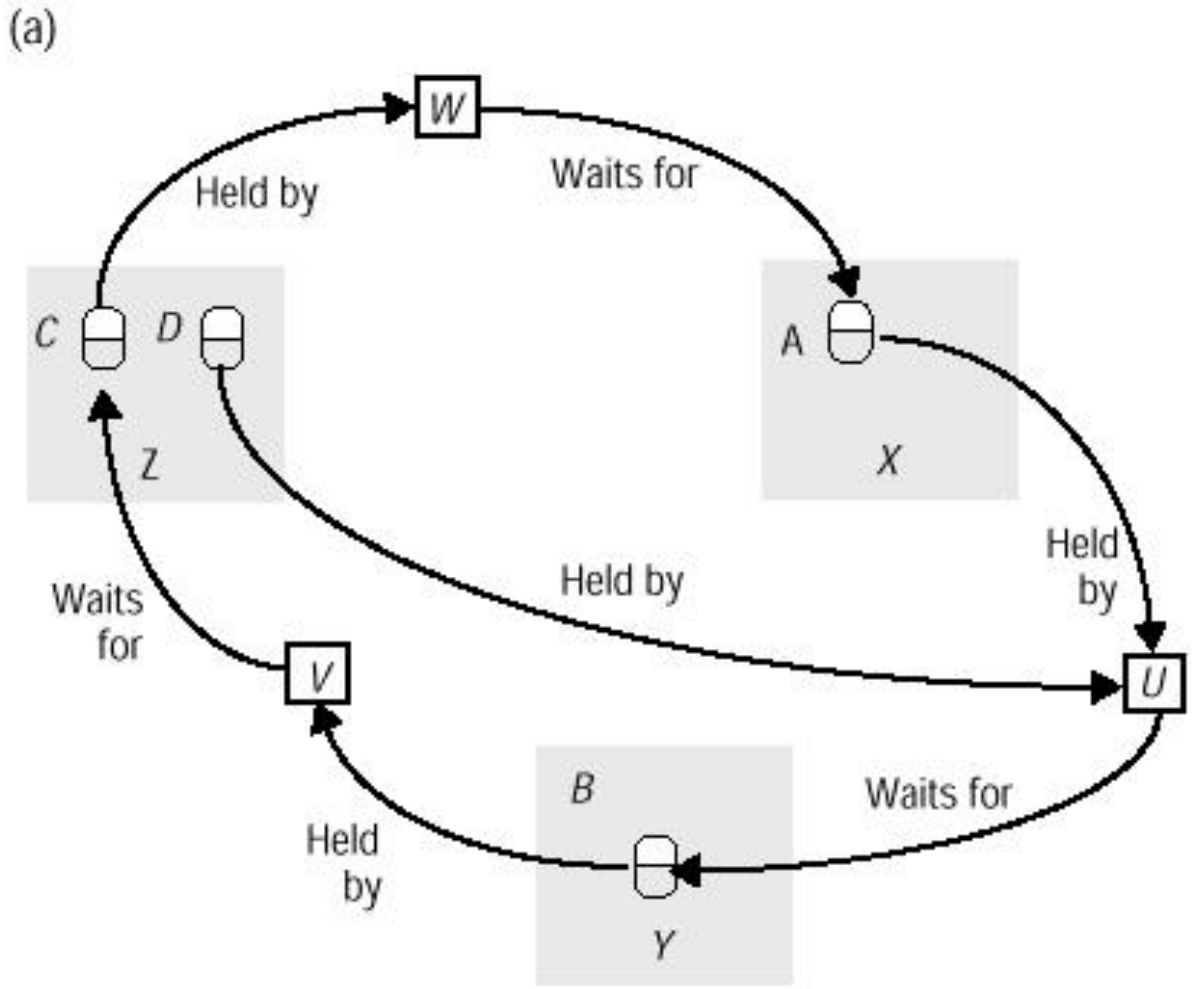
- Outras opções usadas por SGBDs:
  - *Timestamps*
    - Usar Algoritmo de Lamport (Cap. 12) para manter consistência do tempo global
  - Controle de Concorrência Otimista
    - Usa grafos de Holt para detectar deadlocks
      - Cap. 3 – Sistemas Operacionais Tanenbaum

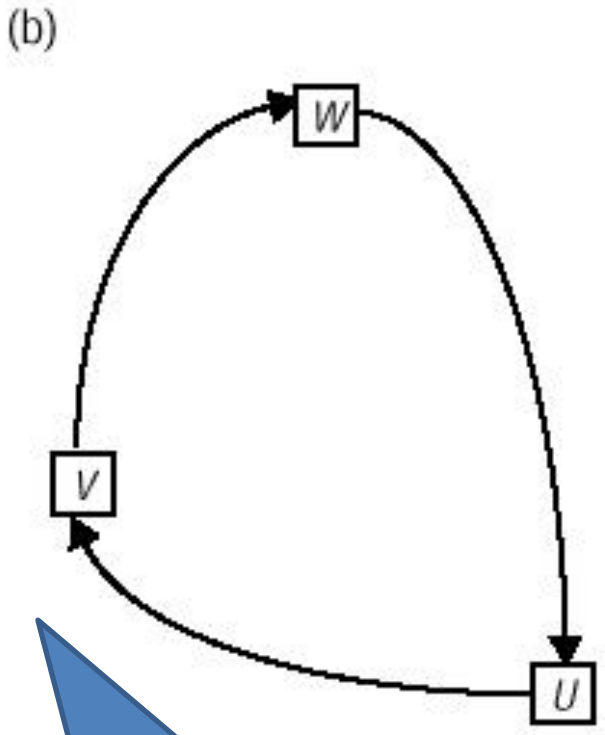
Controle de Concorrência Otimista (2)

U		*V*		*W*	
d.deposit(10)	lock *D*				
		b.deposit(10)	lock *B*		
a.deposit(20)	lock *A*		at *Y*		
	at *X*				
				c.deposit(30)	lock *C*
b.withdraw(30)	wait at *Y*				at *Z*
		c.withdraw(20)	wait at *Z*		
				a.withdraw(20)	wait at *X*

Fonte: G. Coulouris et al., *Distributed Systems: Concepts and Design, Third Edition.*

Controle de Concorrência Otimista (2)





Grafo cíclico indica deadlock

Fonte: G. Coulouris et al., *Distributed Systems: Concepts and Design, Third Edition.*

Exercícios de Fixação

1. Cite as características de uma transação
2. Como funciona o commit de uma fase e qual o seu maior problema em um SD?
3. Como funciona o commit de duas fases ?
4. Qual recurso o controle de concorrência otimista usa para detector deadlocks ?

Exercícios de Fixação (2)

Verifique se no escalonamento abaixo ocorrem deadlocks.

Transação U processo x	**Transação V** processo y	**Transação W** processo z
beginTXN Select * from Func where id = 1 Update Func set salario = salario + 200 where id = 1 Update Func set salario = 0 where id =2 commitTXN	beginTXN Select * from Func where id = 2 Update Func set salario = salario + 110 where id = 2 Update Func set salario = salario * 1.1 where id = 1 commitTXN	begin TXN Delete from Func where id = 1 Update Func set salario = salario * 1.2 from Func where id = 3 commitTXN