Transações Distribuídas

FELIPE CUNHA

Conceito de Transação

Transações podem ser vistas como um grupo de operações combinadas em uma unidade lógica de trabalho.

São usadas para controlar e manter a **consistência** e a **integridade** de cada ação em uma transação, a despeito dos erros que poderão ocorrer no sistema.

Conceito de Transação

Uma **transação** define uma **sequência de operações que é garantida por um servidor**, para ser **atômica** na presença de **múltiplos clientes** e na classe de **falhas por *crash*** de **processos** em servidores.

Transação

Do ponto de vista do cliente, **uma transação é uma sequência de operações que formam um única etapa,** transformando os dados de um servidor de **um estado consistente** para **um outro estado consistente.**

O cliente é provido com operações para marcar o **início e o fim de uma transação.**

Transações

Transação: execução de um conjunto de operações satisfazendo-se as seguintes propriedades (ACID):

- Atomicidade: ou todas as operações de uma transação têm seus efeitos registrados no sistema ou então nenhuma delas é registrada (tudo-ou-nada).
- Consistência: uma transação deve provocar uma transição do sistema de um estado consistente para outro estado consistente.
- Isolamento: a execução de uma transação não deve interferir na execução de nenhuma outra transação
- Durabilidade: uma vez completada, a transação é registrada em algum meio de armazenamento permanente

Operações em uma conta bancária usados numa transação

deposit(amount)
- deposita amount na conta

withdraw(amount)
- retira amount da conta

getBalance() -> amount
- retorna o balance da conta

setBalance(amount)
- Altera balanço da conta

Transação T:
- a. withdraw(100);
- b. deposit(100);
- c. withdraw(200);
- b. deposit(200);

Operações no coordenador da transação

openTransaction() -> trans;

- Inicia nova transação e divulga um TID (idendificador de transação) único.

closeTransaction(trans) -> (commit, abort);

- Finaliza uma transação: um *commit* significa que a transação foi efetuada com sucesso; um *abort* indica que ela não foi completada.

abortTransaction(trans);

- aborta a transação.

Ciclos de vida de uma transação

Bem sucedida	*Abortada pelo Cliente*	*Abortada pelo Servidor*
openTransaction	*openTransaction*	*openTransaction*
operação	*operação*	*operação*
operação	*operação*	*peração*
•	•	Servidor aborta transação → •
•	•	•
operação	*operação*	*ERRO em operação*
		Reportado ao cliente
closeTransaction	*abortTransaction*	

Problema da perda de atualização (lost update problem)

a = $100 b = $200 c = $300

Linha do tempo

Transaction *T* :		**Transaction *U*:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

Resultado Obtido

Os efeitos de permitir as transações T e U executarem concorrentemente como na figura "lost update", ambas as transações obtém o *balance* de B como $200 e então *deposit* $20.

O resultado é incorreto, aumentando o *balance* de B em $20 ao invés de $42.

Por que ?? Erro !!!

O “update” de U é perdido porque T sobrescreve *balance* de B sem ver o “update” de U.

Ambas as transações tem de ler o valor inicial de *balance* de B, antes de qualquer delas escrever o novo valor de *balance* de B.

The “lost update” problem

O problema de “lost update” ocorre quando duas transações T e U lêem o valor “velho” de uma variável (balance) e então usa ele para calcular o novo valor dessa variável (balance).

Erro de leitura (dirty read) quando a transação T aborta

Transaction *T*:		**Transaction *U*:**	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

Resolvendo “lost update”

Pode-se resolver o problema “lost update” por meio de uma **equivalência serial de intercalações de transações T e U**.

Intercalação serialmente equivalente de *T* e *U*

a = $100 b = $200 c = $300

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

Uma intercalação serialmente equivalente de T e U

T Transaction	U Transaction
balance = b.getBalance()	*balance = b.getBalance()*
*b.setBalance(balance*1.1)*	*b.setBalance(balance*1.1)*
a.withdraw(balance/10)	*c.withdraw(balance/10)*

balance = b.getBalance()	**$200**		
*b.setBalance(balance*1.1)*	**$220**		
		balance = b.getBalance()	**$220**
		*b.setBalance(balance*1.1)*	**$242**
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

Regras para conflito das operações read e write

Operações das várias transações		*Conflito*	*Razão*
read	*read*	Não	Porque um par de operações de leitura não dependem da ordem em que foram executadas
read	*write*	Sim	A ordem altera o resultado final
write	*write*	Sim	A ordem altera o resultado final

Equivalência serial

Todos os acessos a um dado devem ser serializados em relação às transações.

Protocolos de controle de concorrência devem serializar as transações.

Soluções para concorrência de transações:

- Travamento com controle distribuído de deadlocks: dado escrito por uma transação é travado até o final da transação.
- Controle otimista de concorrência: supõe-se que não ocorrerá conflito. Na hora do commit, verifica-se problemas de concorrência. Uma das transações envolvidas no problema deve dar um abort.
- Rótulos de tempo: cada transação tem um tempo associado e os dados tem o tempo do último acesso armazenado. Transações abortam quando descobrirem que executaram uma operação tarde demais (por exemplo, perderam uma atualização).

Equivalência Serial

Como implementar no computador ???

Usa-se, para controle de concorrência, o mecanismo de Locks.

Transações *T* and *U* com Locks

Transaction : *T*		Transaction : *U*	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s unlock on *B*
closeTransaction	unlock *A*, *B*	•••	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B* *C*

Locks (Travas)

Um exemplo simples de mecanismo para a disposição das transações em série, é o uso de locks (travas) exclusivos

Nesse esquema, um Lock tenta impedir o acesso (travar) a qualquer dado que esteja para ser usado por qualquer operação da transação de um cliente

Locks

Se um cliente solicitar o acesso a um dado que já está travado devido a transação de outro cliente, o pedido será suspenso e o cliente querendo acessar, deverá esperar até que o objeto seja destravado

Transações Distribuídas

Transação Distribuída: transação cujas operações devem ser executadas em várias das estações de uma rede.

- Envolve atualização de dados em múltiplos BD

Exemplo: transferência bancária de uma conta localizada em uma agência A para uma agência B

- UPDATE A.conta SET saldo= saldo - 100 WHERE num = 49950;
- UPDATE B.conta SET saldo= saldo + 100 WHERE num = 80410;
- COMMIT;

Dificuldade: garantia global de atomicidade

- Ou todos servidores efetivam a transação (commit)
- Ou transação é abortada em todos os servidores

Solução: Protocolo para Garantia de Atomicidade (Atomic Commit Protocol)

Two Phase Commit Protocol (2PC)

Protocolo para garantia de atomicidade de transações distribuídas

Executado em todas as estações que tomam parte de uma transação distribuída

Supõe a existência de um nodo *coordenador*, o qual é responsável pelo monitoramento da transação distribuída

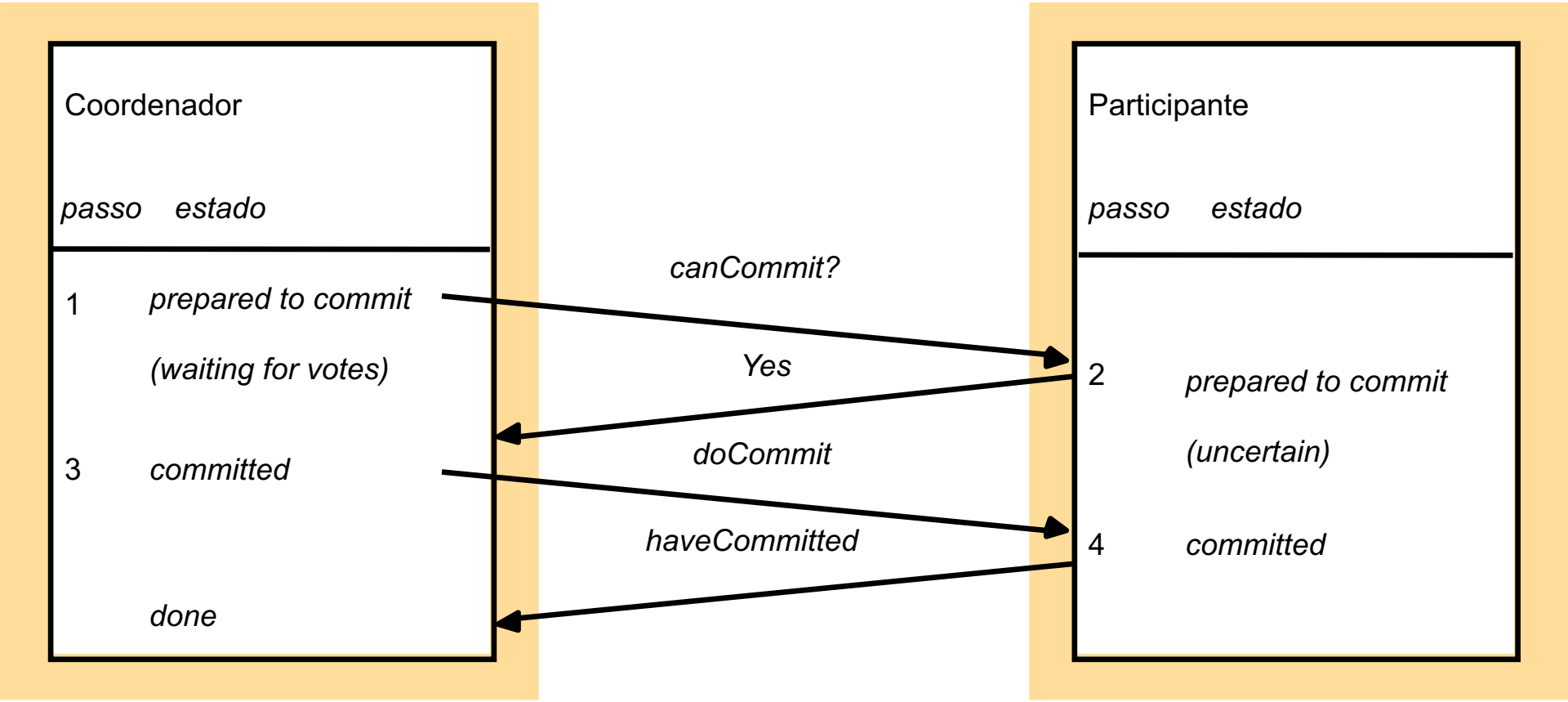
- Normalmente, é o próprio nodo que disparou a transação

Duas fases:

- Fase 1: Votação
- Fase 2: Apuração

Suposição: transações já foram disparadas para os *n* nodos participantes

Comunicação no 2PC



2PC: Fase 1 (Votação)

Coordenador envia mensagem Prepare para todos os participantes

Participante responde com um Sim ou Não. No caso de resposta Não, participante descarta transação.

2PC: Fase 2 (Apuração)

Têm início após o recebimento de n votos

Caso todos os votos sejam Sim

- Coordenador envia mensagem Global_Commit para todos os participantes, que então realizam commit e respondem com um ACK
- Após receber n ACKS, coordenador sinaliza sucesso da transação

Caso haja um único voto Não

- Coordenador envia mensagem Global_Abort para todos os participantes, que então abortam a transação
- Após receber n ACKS, coordenador finaliza protocolo

2PC: Número de Mensagens Trocadas

Observações:

- Nodos não podem alterar seu voto
- Um único *Não* tem poder de veto

Número de mensagens trocadas:

- n mensagens do tipo *Prepare*
- n mensagens contendo os votos dos participantes
- n mensagens *Global_Abort* ou *Global_Commit*
- n *ACKs*

Total de mensagens: 4n

2PC: Tratamento de Falhas

Coordenadores e participantes entram em alguns estados em que ficam esperando mensagens uns dos outros

E se estas mensagens não chegarem ?

- Devido, por exemplo, a problemas na rede

Soluções: *timeout*, para abandonar estado de espera

Após o *timeout*:

- Abortar a transação
- Reenviar uma mensagem

2PC: Situações Possíveis de Falhas

Situação 1: Coordenador esperando voto de um ou mais participantes

- Se voto não chegar dentro de um certo intervalo de tempo (*timeout*), coordenador aborta a transação
- Deve enviar antes mensagem *Global_Abort* para todos os participantes

Situação 2: Coordenador esperando ACK de participante

- Se ACK não chegar dentro de um certo intervalo de tempo (*timeout*), deve reenviar mensagem *Global_Commit* ou *Global_Abort* e então voltar a esperar um ACK.
- Veja que, neste caso, coordenador permanece bloqueado, esperando ACK.

2PC: Situações Possíveis de Falhas

Situação 3: Participante que votou *Sim* e que encontra-se esperando *Global_Commit* ou *Global_Abort*

- Não pode tomar uma decisão unilateral (exemplo: reverter seu voto para *Sim*)
- Deve permanecer bloqueado, aguardando resultado da transação

Logo, falhas têm os seguintes efeitos sobre o Protocolo 2PC

- Podem requerer retransmissão de mensagens
- Podem dar origem a bloqueios

2PC: Implementações

SGBD com suporte a distribuição de dados:

- Exemplos: Oracle, Microsoft SQL Server, IBM DB2 etc

Monitores de Transação:

- Componente essencial em um sistema/cliente servidor de 3 camadas que necessita acessar diversos BD
- Principais tarefas:
  - Garantir atomicidade de transações distribuídas (geralmente via 2PC)
  - Multiplexar carga de acesso a um SGBD
- Exemplos: Microsoft Transaction Server (MTS), BEA Tuxedo, IBM Encina, Java Transaction Server etc