UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC

CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT

GABRIEL DUESSMANN

VICTOR EDUARDO REQUIA

TRANSAÇÃO E CONTROLE DE CONCORRÊNCIA

JOINVILLE

2023

GABRIEL DUESSMANN

VICTOR EDUARDO REQUIA

TRANSAÇÃO E CONTROLE DE CONCORRÊNCIA

Trabalho apresentado ao Curso de Tecnologia em Análise e Desenvolvimento de Sistemas (TADS), Centro de Ciências Tecnológicas, Universidade do Estado de Santa Catarina, como trabalho na disciplina de Sistemas Distribuídos

Orientador: Prof. Dr. Adriano Fiorense.

JOINVILLE

2023

LISTA DE FIGURAS

Figura 1: Interface Account. 7

Figura 2: Exemplo de sincronização na linguagem Java. 8

Figura 3: Exemplo de implementação da interface Coordinator. 12

Figura 4: Alternativas para realização de transações. 12

Figura 5 - Transações aninhadas. 13

Figura 6 - Compatibilidade de travas. 15

Figura 7 – Compatibilidade entre travas de leitura, escrita e confirmação. 18

Figura 8 – Regras da fase de validação. 21

SUMÁRIO

1 **INTRODUÇÃO**……6

2 **TRANSAÇÕES**……6

2.1 SINCRONIZAÇÃO SIMPLES (SEM TRANSAÇÕES)

2.2 CONCEITOS INICIAIS DE TRANSAÇÕES

2.3 PROPRIEDADE DAS TRANSAÇÕES

2.4 MODELO DE FALHAS PARA TRANSAÇÕES

2.5 RECUPERAÇÃO E FALHAS NAS TRANSAÇÕES

2.6 IMPLEMENTAÇÃO DE TRANSAÇÕES EM OBJETOS RECUPERÁVEIS

3 **TRANSAÇÕES ANINHADAS**……12

4 **TRAVAS**……14

4.1 IMPLEMENTAÇÃO DAS TRAVAS

4.2 IMPASSES

4.3 AUMENTO DE CONCORRÊNCIA EM ESQUEMAS DE TRAVAMENTO

4.3.1 TRAVAMENTO DE DUAS VERSÕES

4.3.2 TRAVAS HIERÁRQUICAS

5 CONTROLE DE CONCORRÊNCIA OTIMISTA..……19

5.1 FASE DE TRABALHO

5.2 FASE DE VALIDAÇÃO

5.2.1 VALIDAÇÃO PARA TRÁS

5.2.2 VALIDAÇÃO PARA FRENTE

5.2.3 INANIÇÃO

5.3 FASE DE ATUALIZAÇÃO

6 ORDENAÇÃO POR CARIMBO DE TEMPO..……..……..……..……..……..……...22

6.1 REGRA DE ESCRITA

6.2 REGRA DE LEITURA

6.3 ORDENAÇÃO POR CARIMBO DE TEMPO DE VERSÃO MÚLTIPLA

6.4 REGRA DE ESCRITA NA ORDENAÇÃO POR CARIMBO DE TEMPO DE VERSÃO MÚLTIPLA

7 COMPARAÇÃO DOS MÉTODOS DE CONTROLE DE CONCORRÊNCIA..……25

8 CONSIDERAÇÕES FINAIS..……..……..……..……..……..……..……..……26

1 INTRODUÇÃO

Esse trabalho se refere ao resumo do Capítulo 16 do livro Sistemas Distribuídos**:** Conceitos e Projetos, versão 5. Neste Capítulo do livro, os autores discutem a aplicação de transações e controle de concorrência em objetos compartilhados simultaneamente por múltiplos servidores.

Uma transação define uma sequência de operações no servidor de forma que sejam sempre atômicas, na qual deve poder ser revertida em caso de falhas. As transações aninhadas são estruturas a partir do conjunto de outras transações, de maneira a permitir uma maior concorrência.

O controle de concorrência utiliza de métodos e algoritmos para definir qual transação pode acessar o objeto desejado em determinado tempo a fim de evitar quaisquer problemas pelo seu uso compartilhado entre diversas transações concorrentes.

Neste trabalho, os primeiros dois Capítulos abordam o assunto referente a transações. Nos três capítulos a seguir, é abordado as três estratégias de controle de concorrência. No Capítulo 7 é realizada a comparação entre as estratégias e exemplificado alguns cenários onde o uso de uma abordagem é mais adequado. E por fim, o último Capítulo é destinado às considerações finais deste trabalho e resumo.

2 TRANSAÇÕES

Pela definição de Coulouris, uma transação define uma sequência de operações no servidor que garante que elas sejam atômicas mesmo com a presença de falhas do cliente e até mesmo do servidor. Ou seja, o objetivo de uma transação é garantir que todos os objetos gerenciados por um servidor permaneçam em um estado consistente ao serem acessados por várias transações e na presença de falhas do servidor. É importante destacar que as transações de um cliente também são consideradas indivisíveis em relação às transações de outros clientes, no sentido do qual as operações não podem observar efeitos parciais das operações de outra.

2.1 Sincronização simples (sem transações)

Caso o servidor não seja bem projetado, as operações executadas em clientes diferentes, podem interferir umas nas outras, com isso, podemos causar interferência nos resultados e valores dos objetos. Antes de usar transações, podemos utilizar estratégias de sincronização simples para resolução deste tipo de problema.

Operação atômica no servidor: Por mais que as threads são vantajosas pois aumentam consideravelmente o desempenho em muitos servidores, por efetuar operações de forma concorrente. Devemos tomar cuidados para projetar os sistemas para contexto multithreaded. Caso contrário, podemos ter diversos efeitos colaterais, principalmente quando lidamos com objetos e funções que acessam a mesma região de memória. Com isso, podemos ter inconsistência e erros nas operações. Na figura 1, podemos ver um exemplo de problema com sincronização

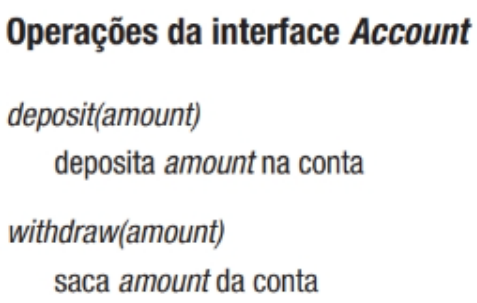

Figura 1: Interface Account

Neste caso, os métodos deposit e withdraw (depósito e saque) é possível que duas ou mais ações que são executadas concorrentemente, sejam interpostas e tenham efeitos estranhos nas variáveis da interface Account. Para resolver este tipo de problema, utilizamos métodos de sincronização para garantir que, nos trechos que podem gerar inconsistência multithread, apenas uma thread seja executada por vez, tendo um funcionamento análogo a um semáforo de trânsito.

Para exemplificar, na linguagem Java, utilizamos o token synchronized para garantir que apenas uma thread por sua vez acesse o objeto ou função específica. Por exemplo, vamos transformar o método deposit em um método síncrono:

```
public synchronized void deposit(int amount) throws RemoteException{
    // adiciona amount ao saldo da conta
}
```

Figura 2: Exemplo de sincronização na linguagem Java

Neste caso, caso duas threads disputem pelo método, o primeiro irá travar o acesso de outro método (locket) assim, se outra thread invocar um dos métodos sincronizados, será bloqueado até que a trava (lock) seja liberada. Essa forma de sincronização, obriga que as threads sejam executadas em tempos diferentes nos trechos de código que podem conter inconsistência com concorrência, garantindo também que as variáveis de instância de um único objeto sejam acessadas de maneira consistente.

Para exemplificar, sem a sincronização, duas invocações distintas do método deposit poderiam ler o saldo antes de uma delas o ter incrementado, resultando em um valor incorreto. Por isso é importante que, todo método que acesse uma variável de instância que possa variar, deve ser sincronizado.

*Melhorando a cooperação dos clientes por meio da sincronização das operações do servidor***:** Neste cenário, lidamos com um clássico problema da computação. O problema do fornecedor e consumidor. Nesta situação, a operação solicitada por um cliente pode não ser concluída até que uma outra operação solicitada por outro cliente tenha sido efetuada. Isso acontece quando alguns clientes são produtores e outros são consumidores. É possível que os consumidores tenham que esperar até que o ou os produtores tenham fornecido mais recursos. Isso pode ocorrer também quando os clientes estão compartilhando recursos. Os clientes que precisam do recurso, talvez tenham que esperar que outros clientes o liberem.

Na linguagem Java, podemos contornar o problema citado anteriormente, com os métodos wait e *notify*. Uma thread chama *wait* em um objeto para ficar suspensa e permitir que outra thread execute um método desse objeto. Uma thread chama notify para informar qualquer outra thread que esteja esperando nesse objeto que ela alterou alguns de seus dados. O acesso ainda é atômico quando as threads esperam umas pela outra pois, uma thread que chama wait, libera sua trava (lock) e fica suspensa como uma única ação atômica. Quando for reiniciada, após ser

notificada, ela deve adquirir uma nova trava sobre o objeto e retomar a execução após a espera, uma thread que chama notify (dentro de um método síncrono) conclui a execução desse método antes de liberar a trava sobre o objeto.

Quando os threads sincronizam suas ações em um objeto com wait e notify, o servidor prende as requisições que não podem ser atendidas imediatamente e o cliente espera por uma resposta até que outro cliente tenha produzido o que ele precisa. Sem a capacidade de sincronizar as threads dessa maneira, um cliente que não pode ser atendido, precisa tentar novamente em um momento posterior. Isto é frustrante tanto para o cliente como para o servidor, pois envolverá o cliente na consulta sequencial do servidor e o servidor na execução de requisições extras. Também existe o problema na qual, outros clientes possam fazer suas requisições antes de esperar que o cliente tente novamente.

2.2 Conceitos iniciais de transações

Historicamente, as transações têm origem nos SGBS (Sistemas de Gerenciamento de Banco de Dados), que tratava uma transação como a execução de um programa que acessa um banco de dados. Nos sistemas distribuídos, as transações foram introduzidas nos servidores de arquivos transacionais como XDFS [Mitchell e Dion 1982]. Neste contexto, uma transação é a execução de uma sequência de requisições de cliente para operações de arquivo.

Apesar das soluções vistas anteriormente, para evitar o uso de transações, existem situações na qual, os usuários exigem que uma sequência de requisições separadas para um servidor seja atômica com as seguintes circunstâncias.

- Sendo livres de interferências de operações sendo efetuadas em nome de outros clientes concorrentes
- Todas as operações sejam concluídas com êxito ou não tenham nenhum efeito na presença de falhas do servidor

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

2.3 Propriedade das Transações

Na maioria dos casos, as transações se aplicam aos objetos recuperáveis e tendem a ser atômicas. Nessas transações, existem alguns aspectos para torná-las transações atômicas

Atomicidade: os efeitos são atômicos mesmo quando o servidor falha. Ou seja, caso exista uma falha, a recuperação deve ser com o todo da transação e não somente na parte em que falhou. Quando existe o êxito, todos os efeitos da transação são gravados nos objetos. Essa propriedade também é conhecida por *tudo ou nada*.

Durabilidade: após uma transação ter sido concluída com êxito,todos os seus efeitos serão salvos em armazenamento permanente.

Isolamento: Esta propriedade garante que cada transação deve ser realizada sem interferência de outras transações. Também deve garantir que as transações de um cliente sejam indivisíveis em relação a outros clientes. Ou seja, as operações não podem observar efeitos parciais das operações de outra.

Consistência: Esta propriedade garante que os resultados produzidos ao final de uma transação serão consistentes mesmo com concorrência entre as transações. Nesse contexto, as transações mesmo que se executadas em processamento concorrente, elas terão o mesmo resultado caso tivessem sido executadas com processamento sequencial

2.4 Modelo de falhas para transações

Para falhas de discos, servidores e comunicação em transações distribuídas, em 1981, Lamport propôs um modelo para contornar ou diminuir esses problemas. Nesse modelo, a alegação é de que, os algoritmos de correção de falhas, funcionam na presença de falhas previsíveis, mas nenhuma alegação é feita a respeito do comportamento quando ocorre um desastre. Embora possam ocorrer erros, eles podem ser detectados e tratados, antes que qualquer comportamento incorreto ocorra. O modelo tem os seguintes princípios:

As escritas em *armazenamento permanentemente podem falhar*, ou não gravando nada ou gravando um valor errado. O armazenamento de arquivos pode também deteriorar, mas, as leituras feitas no armazenamento permanente podem detectar (pela soma de verificação) quando um bloco de dados está danificado.

Os servidores podem falhar ocasionalmente. Quando um servidor danificado é substituído por um novo, primeiro sua memória volátil é colocada em um estado no qual não conhece nenhum dos valores anteriores à falha. Depois disso, ele executa procedimentos de recuperação, usando informações do meio de armazenamento permanente e aquelas obtidas a partir de outros processos, para configurar os

valores dos objetos, incluindo aqueles relacionados ao protocolo de confirmação. Quando um processo falha, ele é obrigado a parar para que seja impedido de enviar mensagens erradas e de escrever valores errados no meio de armazenamento permanente. Isto é, ele não pode produzir falhas arbitrárias. Falhas podem ocorrer a qualquer momento, inclusive durante a recuperação dela mesma.

Pode haver um atraso arbitrário antes da chegada de uma mensagem. Podem ocorrer casos, no qual as mensagens são perdidas, duplicadas ou corrompidas. Os problemas das mensagens corrompidas, podem ser detectados por meio de uma soma de verificação. Em casos em que as mensagens corrompidas não são detectadas ou as mensagens são falsas, são consideradas desastres.

Estes princípios do modelo são usados para projetar um sistema estável e confiável para resistir a qualquer falha de uma forma simples.

2.5 Recuperação e falhas nas transações

Em geral, objetos gerenciados por servidores são armazenados em memória volátil como a RAM (Random Access Memory) ou em memória persistente (Disco rígido, SSD, Fita magnética). Como o armazenamento em memória persistente é mais seguro em caso de falhas, umas das técnicas de recuperação para objetos e dados armazenados em memória volátil é armazenar informações suficientes da memória volátil para a persistente para que, em casos de falhas do processo do servidor, o estado dos objetos sejam recuperados. Com isso, tornamos possível uma maior abrangência de objetos que podem ser recuperados.

Usando a definição de transação, com elas, podemos tratar as falhas por colapso de processos e falhas por omissão na comunicação (não é para todo comportamento arbitrário ou bizantino). Os objetos que podem ser recuperados depois da falha do servidor, são chamados de objetos recuperáveis.

Em linhas gerais, os servidores devem garantir que a transação inteira seja executada e que os resultados em caso de êxito, sejam armazenados em memória persistente. Nos casos em que os objetos são armazenados em memória volátil, os dados mais importantes devem ser guardados no armazenamento persistente para, em caso de uma transação falhar, os dados serem recuperados e os efeitos serem completamente desconsiderados. Essa propriedade será de extrema importância para manter os conceitos ACID (Atomicidade, Consistência, Isolamento e Durabilidade) em uma transação, como veremos posteriormente.

2.6 Implementação de transações em objetos recuperáveis

Para implementar transações em objetos recuperáveis, primeiro temos que ter a garantia de que os objetos são recuperáveis. Então, cada transação é criada e gerenciada por um coordenador, que implementa a interface Coordinator, mostrada na Figura 3

openTransaction( ) → trans;

inicia uma nova transação e gera um TID *trans* exclusivo. Esse identificador será usado nas outras operações da transação.

closeTransaction(trans) → (commit, abort);

termina uma transação: um valor de retorno *commit* indica que a transação foi confirmada; um valor de retorno *abort* indica que ela foi cancelada.

abortTransaction(trans);

cancela a transação.

Figura 3: Exemplo de implementação da interface Coordinator

Para cada transação, o coordenador irá atribuir um identificador TID. O cliente invoca o método openTransaction do coordenador para abrir uma nova transação. O retorno do método invocado pelo cliente é o identificador TID. Quando a transação é finalizada, o cliente invoca o método closeTransaction. Após o método closeTransaction, todos os objetos recuperáveis devem ser salvos. Caso o cliente queira abortar uma transação durante a execução, poderá ser invocado o método abortTransaction que deverá reverter toda a execução da transação para o estado estável. A Figura 4, mostra as três possíveis alternativas das transações.

Bem-sucedida	*Cancelada pelo cliente*	*Cancelada pelo servidor*	
openTransaction	*openTransaction*		*openTransaction*
operação	*operação*		*operação*
operação	*operação*		*operação*
•	•	o servidor cancela	•
•	•	*transação* →	•
operação	*operação*		*ERRO de operação relatado ao cliente*
closeTransaction	*abortTransaction*		

Figura 4: Alternativas para realização de transações

3 TRANSAÇÕES ANINHADAS

As transações aninhadas permitem que as transações sejam compostas de outras transações, assim, várias transações podem ser iniciadas dentro de uma transação. A transação mais externa em um conjunto de transações é chamada de transação de nível superior, e as demais de subtransações. A Figura 1 apresenta

um diagrama de transações aninhadas, sendo *T* a transação de nível superior, e as demais subtransações dessa transação *T.*

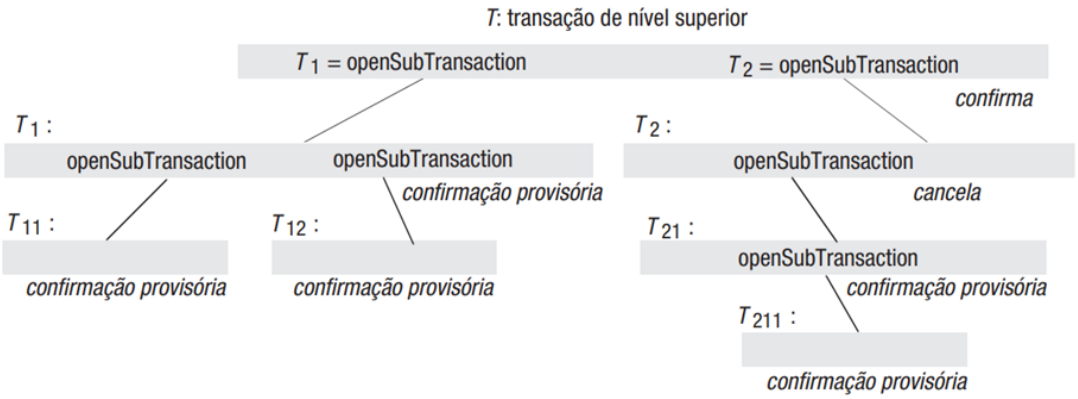

Figura 5 - Transações aninhadas

Uma subtransação é atômica para sua ascendente com relação às falhas de transação e ao acesso concorrente. As substransações que estão no mesmo nível podem ser executadas concorrentemente, com acesso aos objetos organizados em série.

Cada subtransação pode falhar independente de sua ascendente e de outras subtransações. Quando uma substransação é cancelada, a transação ascendente pode escolher outra substransação alternativa para completar a tarefa.

Para evitar problemas entre as transações pais e filhas, as seguintes regras devem ser seguidas:

- Uma transação só pode ser confirmada ou cancelada após suas transações descendentes tiverem sido concluídas.
- Quando uma subtransação é concluída, ela toma uma decisão independente de ser confirmada provisoriamente ou ser cancelada.
- Quando uma transação ascendente é cancelada, todas as suas subtransações também são canceladas.
- Quando uma subtransação é cancelada, a transação ascendente pode decidir se vai cancelar ou não.
- Se a transação de nível superior é confirmada, todas as subtransações que foram confirmadas provisoriamente também podem ser

confirmadas, desde que nenhuma de suas ascendentes tenha sido cancelada.

4 TRAVAS

Nesse esquema, o servidor tenta impedir o acesso a qualquer objeto que é usado por qualquer outra operação de transação de um cliente. Se um cliente solicitar o acesso a um objeto que já está travado devido à transação de outro cliente, a requisição será suspensa e o cliente deverá esperar até que o objeto seja destravado.

A equivalência serial exige que todos os acessos de uma transação a um objeto sejam executados em série com relação aos acessos feitos por outras transações. Todos os pares de operações conflitantes de duas transações devem ser executadas na mesma ordem. Para garantir isso, uma transação não pode solicitar novas travas após ter liberado uma. Na primeira fase de cada transação, novas travas são adquiridas, enquanto que na segunda fase, essas são liberadas. Esse processo é chamado de travamento de duas fases.

Como as transações podem ser canceladas, são necessárias execuções restritas para evitar leituras sujas e escritas prematuras, podendo ser retardada até que outras transações que escreveram no mesmo objeto tenham sido confirmadas ou canceladas. Esse processo é chamado de travamento de duas fases restrito, na qual a presença de uma trava impede que outras transações leiam ou escrevam os objetos. Mesmo quando uma transação é confirmada, o travamento de um objeto deve ser mantido até que as atualizações sejam armazenadas permanentemente.

Um dos protocolos de concorrência para resolver o conflito de operações de leitura e escrita, permitindo várias transações concorrentes sobre um objeto, é usar travas de leitura e travas de escrita no objeto, onde o objeto pode ter no máximo uma trava. Antes que uma operação de leitura de uma transação seja executada, deve ser alocada uma trava de leitura no objeto. Antes que a operação de escrita seja executada, deve ser alocada uma trava de escrita no objeto. Quando for impossível alocar uma trava imediatamente, a transação deverá esperar até que seja possível. Duas operações de leitura de diferentes transações não entram em

conflito, pois as transações podem compartilhar a mesma trava de leitura, por isso as travas de leituras nesse cenário podem ser chamadas de travas compartilhadas.

Com isso, temos as seguintes regras de conflito:

1. Se uma transação T já executou uma operação de leitura sobre um objeto, uma transação concorrente U não pode escrever esse objeto até que T seja confirmada ou cancelada.

2. Se uma transação T já executou uma operação de escrita sobre um objeto, uma transação concorrente U não pode ler nem escrever esse objeto até que T seja confirmada ou cancelada.

Figura 6 - Compatibilidade de travas

Para um objeto		*Trava solicitada*	
		leitura	*escrita*
Trava já alocada	*nenhum*	OK	OK
	leitura	OK	espera
	escrita	espera	espera

Fonte: Coulouris, Dollimore, Kindberg, et al (2013).

A Figura 2 resume as regras de conflito apresentadas. No lado esquerdo, mostra as travas já alocadas para o objeto, se existirem, e nas colunas a direita, apresenta-se o resultado ao solicitar um novo tipo de trava. Nos casos onde o resultado é de espera, deve ser aguardado até que a transação que possui a trava seja confirmada ou cancelada, pois então as travas sobre o objeto são removidas e novas travas requisitadas por uma nova transação podem ser adicionadas.

As travas também podem ser promovidas, na qual ocorre uma conversão de uma trava em uma outra mais forte e menos exclusiva. A trava de leitura permite outras travas de leitura, enquanto a trava de escrita, não. Portanto, uma trava de escrita é mais exclusiva do que a trava de leitura. Entretanto, não é seguro rebaixar uma trava mantida por uma transação antes que ela seja confirmada, pois pode

possibilitar execuções de outras transações que sejam inconsistentes com a equivalência serial.

4.1 IMPLEMENTAÇÃO DAS TRAVAS

As travas são implementadas por um servidor, chamado de gerenciador de travas. Este servidor armazena um conjunto de travas e cada trava seria uma instância de uma classe *Lock*. Cada instância dessa classe possui os seguintes atributos:

- O identificador do objeto travado;
- Os identificadores das transações que concorrentemente mantêm as travas;
- Um tipo de trava.

Essa classe conta com dois métodos importantes, o método *acquire* para adicionar uma nova trava e *release* para remover a trava de uma transação sobre um objeto. O método *acquire* recebe como parâmetros um identificador de transação e o tipo de trava requisitado. Ele verifica se a requisição pode ser atendida. Caso outra transação possua uma trava em modo conflitante, ele ativa *wait*, o que faz a transação esperar até escutar um *notify* emitido pela liberação da trava. Quando finalmente a condição é satisfeita, o algoritmo executa os seguintes passos:

- Se nenhuma outra transação possui trava, adiciona a transação no grupo de proprietários e configura o tipo da trava;
- Se outra transação possui trava que permite compartilhamento, apenas adiciona a dada transação no grupo de proprietários;
- Se essa transação é proprietária, mas está solicitando uma trava mais forte, promove a trava.

O método *release* recebe como argumento o identificador da transação, e remove esse identificador do grupo de proprietários, configura o tipo da trava como nenhum e chama o método "notifyAll", no qual notifica todas as *threads* que estão aguardando.

4.2 IMPASSES

O uso de travas pode levar a impasses (*deadlocks).* O impasse é um estado no qual cada membro de um grupo de transações está esperando algum outro membro liberar uma trava. Essa transação pode durar um longo período de tempo, resultando em muitos objetos sendo travados e permanecendo assim, impedindo que outros clientes utilizem os respectivos objetos.

Uma solução simples para evitar os impasses é adquirir as travas de todos os objetos usados por uma transação quando ela inicia. Tal transação não pode entrar em um impasse com outras transações, mas ela restringe desnecessariamente o acesso a recursos compartilhados. Além disso, às vezes é impossível prever, no início de uma transação, quais objetos serão usados.

Uma estratégia comumente utilizada para solução de impasses é a limitação do tempo de bloqueio. Nessa estratégia, cada trava recebe um período de tempo limitado, durante o qual ela é invulnerável. Após esse tempo, a trava se torna vulnerável, e a partir desse momento, qualquer outra transação que estiver esperando para acessar o objeto protegido irá conseguir acessá-lo, pois a trava pertencente à transação anterior é cancelada e sua trava sobre o objeto é retirada.

4.3 AUMENTO DE CONCORRÊNCIA EM ESQUEMAS DE TRAVAMENTO

Com o tempo, foram desenvolvidas novas implementações baseadas no esquema de travamento que visa aumentar a concorrência do algoritmo.

4.3.1 Travamento de Duas Versões

O travamento de duas versões trata-se de um esquema otimista que permite a transação gravar versões de tentativa dos objetos, enquanto outras leem a versão confirmada dos mesmos objetos. As operações de leitura só ficam em espera se outra transação estiver correntemente confirmando o mesmo objeto.

As transações que pedem para serem confirmadas precisam esperar até que outras transações de leitura tenham terminado sua operação. Portanto, esse esquema permite mais concorrência, entretanto, as transações de escrita correm o risco de espera ou anulação, quando tentarem ser confirmadas. A Figura 3 apresenta a compatibilidade dessa estratégia.

Figura 7 – Compatibilidade entre travas de leitura, escrita e confirmação

Para um objeto		*Trava a ser configurada*		
		leitura	*escrita*	*confirmação*
Trava já configurada	*nenhum*	OK	OK	OK
	leitura	OK	OK	espera
	escrita	OK	espera	–
	confirmação	espera	espera	–

Fonte: Coulouris, Dollimore, Kindberg, et al (2013).

Essa estratégia utiliza três tipos de trava: trava de leitura, de escrita e de confirmação. Antes que a operação de leitura de uma transação seja efetuada, é adicionada uma trava de leitura sobre o objeto, sendo bem-sucedido a não ser que o objeto já tenha uma trava de confirmação, e terá que esperar. Na operação de escrita, o objeto requisitado não pode ter nem trava de escrita e nem de confirmação. Quando uma transação pede para ser confirmada, o coordenador de transação tenta converter todas as travas de escrita dessa transação em travas de confirmação.

4.3.2 Travas Hierárquicas

Algumas aplicações exigem que a granularidade seja maior, onde uma transação deseja executar operações em diversos objetos ao mesmo tempo, de forma que todos esses devem receber travas. Para reduzir a sobrecarga de tipo de travamento, foi proposto uma hierarquia de travas com diferentes granularidades.

Em cada nível, a configuração de uma trava ascendente tem o mesmo efeito que configurar todas as travas descendentes equivalentes. Antes que um nó descendente receba uma trava de leitura ou escrita, a intenção de usar essa trava é configurada no nó ascendente e em seus ancestrais. Neste modelo, é usado um terceiro tipo de trava, a de intenção, que combina propriedades de uma trava de

leitura com uma intenção de escrita. A trava de intenção é compatível com outras travas de intenção, mas entra em conflito com as travas de leitura e escrita.

5 CONTROLE DE CONCORRÊNCIA OTIMISTA

A partir da estratégia de travamento sobre objetos, várias desvantagens foram identificadas, sendo algumas delas:

- A manutenção da trava representa uma sobrecarga que não está presente em sistemas que não suportam acesso concorrente a dados compartilhados. Mesmo as transações somente de leitura, que possivelmente não afetam a integridade dos dados, devem usar travas para garantir que os dados não sejam modificados simultaneamente por outras transações.
- Pode resultar em impasses. A prevenção de impasses reduz significativamente a concorrência e, portanto, as situações de impasse devem ser resolvidas com o uso de tempos limites ou com detecção de impasses. Nenhuma dessas soluções é totalmente satisfatória para o uso em programas interativos.
- Para evitar os cancelamentos em cascata, as travas não podem ser liberadas até o final da transação, o que pode reduzir o potencial de concorrência.

A estratégia apresentada neste Capítulo é chamada de otimista, baseado na observação de que, na maioria das aplicações, a probabilidade das transações de dois clientes acessarem o mesmo objeto é baixa. As transações podem prosseguir como se não houvesse nenhuma possibilidade de conflito, até que o cliente conclua sua operação e feche a transação. Quando surge um conflito, alguma transação geralmente é cancelada e precisará ser reiniciada pelo cliente. Cada transação passa por três fases.

5.1 FASE DE TRABALHO

Durante a fase de trabalho, cada transação tem uma versão tentativa de cada um dos objetos que atualiza. Trata-se de uma cópia da versão do objeto confirmada mais recente. O uso de versões de tentativa permite que a transação seja cancelada sem nenhum efeito sobre os objetos.

As operações de leitura são executadas imediatamente, acessando uma versão de tentativa para essa transação, se já existir; caso contrário, acessa o valor do objeto confirmado mais recente. As operações de escrita registram os novos valores dos objetos como valores de tentativa, no qual são invisíveis para outras transações neste momento.

5.2 FASE DE VALIDAÇÃO

Quando a requisição *closeTransaction* para fechar a transação é invocada, a transação é validada para estabelecer se suas operações sobre os objetos entram em conflito ou não com as operações de outras transações sobre os mesmos objetos. Se a validação for bem-sucedida, a transação poderá ser confirmada. Se falhar, uma forma de solução de conflito deve ser usada, e a transação corrente, ou em alguns casos, aquelas conflitantes, precisarão ser canceladas.

A validação usa as regras de conflito de leitura e escrita para garantir que a transação seja seriamente equivalente com relação a todas as outras transações sobrepostas, isto é, todas as transações que ainda não tinham sido confirmadas no momento que essa transação começou.

Para ajudar na realização da validação, cada transação recebe um número ao entrar na fase de validação. Se a transação for validada e terminar com sucesso, ela manterá esse número; se falhar nas validações e for cancelada, ou se a transação for somente de leitura, o número será liberado para uma nova atribuição posterior. Esses números são inteiros e atribuídos em ordem crescente, de forma que a sequência das transações define sua posição no tempo.

O teste de validação é baseado no conflito entre operações em pares de transação *Ti* e *Tv*. Para que uma transação *Tv esteja* de acordo com a validação em relação a uma transação sobreposta *Ti*, suas operações devem obedecer às regras descritas na Figura 4.

Figura 8 – Regras da fase de validação

T_V	T_i	Regra	
escrita	*leitura*	1.	T_i não deve ler objetos escritos por T_V.
leitura	*escrita*	2.	T_V não deve ler objetos escritos por T_i.
escrita	*escrita*	3.	T_i não deve escrever em objetos modificados por T_V e T_V não deve escrever em objetos modificados por T_i.

Fonte: Coulouris, Dollimore, Kindberg, et al (2013).

Uma simplificação pode ser obtida criando-se a regra de que apenas uma transação pode estar na fase de validação e atualização em dado momento. Quando duas transações não puderem se sobrepor na fase de atualização, a regra 3 é satisfeita. Nota que essa restrição sobre as operações de escrita produz execuções restritas, que podem ser implementadas como uma seção crítica para que apenas um cliente possa executar por vez.

A validação das regras 1 e 2 devem ser testadas sobre as sobreposições entre os objetos de pares de transação *Tv* e *Ti*. Existem duas formas de validação, a validação para trás e a para frente. A validação para trás verifica a transação sendo submetida a validação com outras transações sobrepostas precedentes, isto é, aquelas que entram na fase de validação antes dela. A validação para frente verifica a transação sendo submetida à validação com as transações posteriores, que ainda estão ativas.

5.2.1 Validação Para Trás

Na validação para trás, como todas as operações de leitura das transações sobrepostas anteriores foram executadas antes que a validação de *Tv* começasse, essas não serão afetadas pelas escritas da transação corrente, o que satisfaz a regra 1.

A validação da transação *Tv* verifica se seu conjunto de leitura se sobrepõe a qualquer um dos conjuntos de escrita das transações sobrepostas anteriores, para atender a regra 2. Se houver qualquer sobreposição, a validação falhará e a transação será cancelada.

5.2.2 Validação Para Frente

Na validação para frente de uma transação, a regra 2 é satisfeita automaticamente, pois as transações ativas só escrevem depois que a transação corrente tiver sido finalizada.

Para atender a regra 1, é preciso comparar se o conjunto de escritas da transação *Tv* não possui nenhuma interseção com o conjunto de leitura de outras transações sobre o mesmo objeto. Caso a validação falhar, tem-se algumas escolhas que podem ser tomadas, sendo essas:

- Cancelar a transação que está sendo validada.
- Adiar a validação até um momento posterior, quando as transações conflitantes tiverem terminado.
- Cancelar todas as transações ativas conflitantes e confirmar a transação que está sendo validada.

5.2.3 Inanição

Em sistemas que contém reinício automático de transações canceladas, não há nenhuma garantia que essa transação irá passar nas verificações, pois pode continuar entrando em conflito com outras transações ativas. Para evitar esses casos raros de inanição, o servidor que utiliza o controle de concorrência otimista deve garantir que um cliente não tenha sua transação cancelada repetidamente. Quando detectados esses casos particulares, tal transação deve receber acesso exclusivo para uso de uma seção crítica protegida por um semáforo.

5.3 FASE DE ATUALIZAÇÃO

Se uma transação é validada, nesta última fase, todas as alterações em versões de tentativa são registradas de forma permanente no objeto.

6 ORDENAÇÃO POR CARIMBO DE TEMPO

Na estratégia baseada na ordenação por carimbo de tempo, cada operação em uma transação é validada ao ser executada. Se a operação não puder ser validada, a transação é cancelada imediatamente. Cada transação recebe um valor de carimbo de tempo ao iniciar, e esse carimbo define sua posição na sequência de tempo das transações. Dessa forma, as requisições de transações podem ser totalmente ordenadas, de acordo com seus carimbos de tempo.

A requisição de uma transação para escrever em um objeto é válida se esse objeto foi lido e escrito pela última vez por transações anteriores. A requisição para ler um objeto é válida somente se esse objeto foi escrito pela última vez por uma transação anterior. Esse algoritmo presume que exista apenas uma versão de cada objeto e restringe o acesso a uma transação por vez. Entretanto, a regra de ordenação por carimbo de tempo é refinada para garantir que cada transação acesse um conjunto consistente de versões do objeto. Também deve garantir que as versões de tentativa do objeto sejam efetuadas na ordem determinada pelos carimbos de tempo das transações que as executaram.

As operações de escrita são registradas em versões de tentativa do objeto e são invisíveis para outras transações até que uma requisição *closeTransaction* seja emitida e a transação seja confirmada. Todo objeto mantém o carimbo de tempo de escrita máximo e um conjunto de versões de tentativa, cada uma das quais também tem um carimbo de tempo para a escrita e para a leitura. O carimbo de tempo de escrita do objeto confirmado é anterior ao de qualquer um de suas versões de tentativa, e o conjunto de tempo de leitura pode ser representado por seu membro máximo. Quando a operação de escrita de uma transação sobre um objeto é aceita, o servidor cria uma nova versão de tentativa do objeto, com carimbo de tempo de escrita igual ao carimbo de tempo da transação. A operação de leitura de uma transação é direcionada para a versão de tentativa com um carimbo de tempo de escrita menor do que o carimbo de tempo da transação. Quando a operação de leitura de uma transação é aceita, o carimbo de tempo da transação é adicionado em seu conjunto de carimbo de tempo de leitura. Por fim, quando uma transação é confirmada, os valores das versões de tentativa se tornam os valores dos objetos, e os carimbos de tempo das versões de tentativa se tornam os carimbos de tempo dos objetos correspondentes.

6.1 REGRA DE ESCRITA

Na regra de escrita, temos dois casos onde são comparados o carimbo de tempo da transação corrente *Tc* requisitando uma operação de escrita e outras transações *Ti* de leitura sobre o mesmo objeto; no segundo cenário, a transação *Tc* também requisita uma operação de escrita, mas compara com transações *Ti* que executam ação de escrita. No primeiro caso, *Tc* não pode escrever um objeto que

tenha sido lido por qualquer *Ti,* onde *Ti* > *Tc;* exigindo que *Tc* seja maior ou igual o carimbo de tempo de leitura máximo do objeto. Para o segundo caso, *Tc* não deve escrever um objeto que tenha sido modificado por qualquer *Ti*, onde *Ti* > *Tc*; exigindo que *Tc* seja maior que o carimbo de tempo de escrita do objeto confirmado. Para ambos os cenários, qualquer operação de escrita que chegue tarde demais deve ser cancelada, pois outra transação com um carimbo de tempo posterior já leu ou escreveu no objeto.

6.2 REGRA DE LEITURA

Para a regra de leitura, tem-se apenas um caso, onde *Tc* requisita uma operação de leitura e outras transações *Ti* requisitam escrita sobre o mesmo objeto. Nesse cenário, *Tc* não deve ler um objeto que tenha sido modificado por qualquer *Ti,* onde *Ti* > *Tc;* exigindo que *Tc* seja maior que o carimbo de tempo de escrita do objeto confirmado. Se a operação de leitura chegar tarde demais, será cancelada, pois outra transação com um carimbo de tempo posterior já escreveu no objeto.

6.3 ORDENAÇÃO POR CARIMBO DE TEMPO DE VERSÃO MÚLTIPLA

Na ordenação por carimbo de tempo de versão múltipla, é mantido uma lista de versões confirmadas antigas, assim como das versões de tentativa, para cada objeto. Essa lista representa o histórico dos valores do objeto. A vantagem de utilizar múltiplas versões é que as operações de leitura que chegam tarde demais não precisam ser rejeitadas.

Assim como na ordenação por carimbo de tempo simples, as operações de escrita e leitura funcionam da mesma forma. Porém, quando uma leitura chegar tarde, ela poderá ter permissão para ler uma versão confirmada antiga, para que não haja necessidade de cancelar a operação.

Não há nenhum conflito entre operações de escrita de diferentes transações, pois cada transação escreve sua própria versão confirmada dos objetos que acessa.

6.4 REGRA DE ESCRITA NA ORDENAÇÃO POR CARIMBO DE TEMPO DE VERSÃO MÚLTIPLA

Quando se tem o cenário onde a transação corrente *Tc* deseja realizar uma operação de escrita em um objeto que está sendo lido por outras transações *Ti*, precisa achar primeiramente a versão do objeto *D* na qual o carimbo de tempo de escrita máximo é menor ou igual a *Tc.* A versão do objeto D deve ter carimbo de tempo de leitura menor ou igual a *Tc* para que então a operação de escrita possa ser executada em uma versão de tentativa de *D* com carimbo de tempo de escrita *Tc*; se o carimbo de tempo de leitura for igual, a transação *Tc* é cancelada.

7 COMPARAÇÃO DOS MÉTODOS DE CONTROLE DE CONCORRÊNCIA

Todos os métodos de controle de concorrência apresentados possuem sobrecargas de tempo e no espaço exigido, e todos limitam até certo ponto o potencial de operação concorrente.

Ambos os métodos de travamento de duas fases e ordenação por carimbo de tempo usam algoritmos pessimistas, nas quais os conflitos entre as transações são detectados quando o objeto é acessado. A ordenação por carimbo de tempo, e em especial, a ordenação por carimbo de tempo de versão múltipla, são melhores do que o travamento de duas fases para transações somente de leitura. O travamento de duas fases é melhor quando as operações nas transações são predominantemente escritas.

Os métodos pessimistas lidam diferentemente quando é detectado um acesso conflitante a um objeto. A ordenação de carimbo por tempo cancela a transação imediatamente, enquanto o travamento faz a transação esperar, mas pode acabar cancelando-a posteriormente para evitar impasses.

Quando é usado o controle de concorrência otimista, todas as transações podem prosseguir, mas algumas são canceladas quando tentam ser confirmadas. Isso resulta em uma operação relativamente eficiente quando existem poucos conflitos.

8 CONSIDERAÇÕES FINAIS

A possibilidade de concorrência no acesso de objetos permite que sistemas se tornem mais robustos, pois permitem que diversos clientes ou transações façam operações simultaneamente. Sem a concorrência, haveria muitos gargalos em sistemas complexos, especialmente na Web onde há um grande volume de usuários e dados sendo acessados a todo momento. Entretanto, essa concorrência entre mesmos objetos pode levar a falhas e inconsistências, daí a importância de servidores aplicarem estratégias de controle de concorrência e garantir o bom funcionamento das aplicações.

Por fim, nenhum algoritmo de controle de concorrência é perfeito, mas tentam propor uma solução satisfatória dentro de seu espaço. É importante saber das diferentes abordagens para caso queira aplicar um controle de concorrência, aplicar o que mais se encaixa dentro do cenário do servidor e sistema.

REFERÊNCIAS

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim; et al. Sistemas Distribuídos.Bookman Companhia Editora Ltda, Grupo A, 2013. E-book. ISBN 9788582600542. Disponível em: https://app.minhabiblioteca.com.br/#/books/9788582600542/. Acesso em: 02 jun. 2023.

LANG, Marcelo P.; NETO, Camillo O. P. Transação em Sistemas Distribuídos e Abertos. Disponível em: http://www.batebyte.pr.gov.br/Pagina/TRANSACOES-EM-SISTEMAS-DISTRIBUIDOS-E-ABERTOS. Acesso em: 10 jun. 2023.