PADI-DSTM

Abstract

*O **PADI-DSTM** é um sistema distribuído que permite gerir objectos que residem em memória e são partilhados por programas transaccionais que correm em máquinas diferentes.*

1. Introdução

Inicialmente iremos abordar a nossa solução e compará-la com algumas alternativas que considerámos, apresentando uma vista global da sua arquitectura. De seguida falaremos das estruturas de dados e algoritmos escolhidos. Por fim, iremos abordar o trabalho a ser realizado posteriormente - tolerância a faltas - e iremos fazer uma pequena conclusão.

2 Solução

2.1 2PC vs S2PL vs Timestamps

Inicialmente considerámos seguir uma abordagem optimista. No entanto, não conhecendo a relação existente entre o número de leituras e escritas, admitimos que existe igual número de leituras e escritas, logo o número de conflitos poderá vir a ser grande.

Constatando esse potencial bottleneck de performance no nosso sistema, optámos por abandonar as soluções que usam timestamps ou two-phase-commit (2PC). Um outro argumento contra o 2PC é a possibilidade das transacções poderem abortar (se surgir algum conflito) depois de já terem realizado algum trabalho, o que implicaria refazer-se esse trabalho.

Assim, escolhemos utilizar *Strict Two Phase Locking* ou *S2PL*

2.2 Replicação activa vs replicação passiva

Depois de escolhermos qual o protocolo a usar, deparámo-nos com a escolha entre replicação activa e passiva. Inicialmente considerámos usar replicação activa com um protocolo de *Quorum Consensus*. No entanto, por esta precisar de três servidores (em vez dos dois necessários para a replicação passiva) e tendo também em conta que teríamos que enviar e receber mais mensagens do que as necessárias ao usar replicação passiva, optámos por esta última opção.

3 Estruturas de Dados

3.1 Master

Esta classe gere o sistema de memória distribuída e é responsável por armazenar os dados globais do sistema.

É importante salientar que a lista *Servidores* tem como finalidade garantir que no caso em que o servidor primário é substituído, o endereço registado é actualizado e os clientes continuam a aceder aos *PadInt*s sem perturbações. Para além disso, os *TID* são atribuídos de forma sequencial e única. De seguida apresentam-se as estruturas internas desta classe.

Variável	Descrição
TID	Último *Transaction ID* atribuído
Servidores	Estrutura que mapeia o identificador de cada servidor primário com o seu endereço

Table 1. Atributos da classe Master

3.2 Servidor

O conjunto das instâncias da classe Servidor representa a memória distribuída onde são armazenados os *PadInt*s. De seguida apresentam-se os atributos desta classe.

Variável	Descrição
Pedidos	Lista de pedidos feitos ao servidor enquanto o servidor está no modo *Freeze*
Réplica	Endereço do outro servidor
(UID,PadInt)	Estrutura que mapeia *UID*s em *PadInt*s

Table 2. Atributos da classe Servidor

A localização de cada *PadInt* depende do número total de servidores primários, ou seja, é dada por: *UID* **mod** *Nº de Servidores.*

3.3 PadInt

Esta classe representa o objecto gerido pelo *PADI-DSTM* que guarda um inteiro, onde o *lock* é apenas o TID. Esta classe é composta por:

Variável	Descrição
UID	Identificador do PadInt que representa
Valor actual	Valor no momento actual da transacção
Valor original	Valor no início da transacção
Temporizador	Usado na detecção de deadlocks
Promoção	Referência para a próxima transação a ser promovida
Leitores	Fila de transacções com locks de leitura atribuídos
Escritor	Transacção com lock de escrita atribuído
Leitores à espera	Fila de transações com locks de leitura à espera
Escritores à espera	Fila de transações com locks de escrita à espera

Table 3. Atributos da classe PadInt

3.4 Stub do PadInt

A Biblioteca envia ao cliente stubs da classe *PadInt*. Esta classe tem a seguinte estrutura:

Variável	Descrição
UID	Identificador do PadInt que representa
Biblioteca	Referência para a Biblioteca do Cliente

Table 4. Atributos da classe Stub do PadInt

Esta classe tem os seguintes métodos:

- *int Read()*: Invoca o método *Read* da Biblioteca
- *void Write(int value)*: Invoca o método *Write* da Biblioteca

3.5 Biblioteca

Esta é a classe que representa a biblioteca usada pelos clientes para comunicar com o sistema de memória distribuída. Esta classe tem a seguinte estrutura:

Variável	Descrição
Nº de Servidores	Número total de servidores primários existentes
TID	Transacção atribuída pelo *Master*
(UID, Servidor)	Lista de associações entre *UID*s e o número do seu respectivo *Servidor*
Cache de Servidores	Estrutura que mapeia o número do *Servidor* no respectivo endereço
Temporizadores	Lista de *timers* para cada servidor

Table 5. Atributos da classe Biblioteca

De seguida apresentam-se alguns métodos da Biblioteca:

- *bool init()*: A Biblioteca pergunta ao Master qual é o número de servidores primários existentes;
- *bool TxBegin()*: A Biblioteca pede ao Master para criar um novo *TID* para a transacção e regista-o;
- *PadInt CreatePadInt(int UID)*: A Biblioteca calcula qual o servidor primário onde vai alocar o novo *PadInt*. De seguida pergunta ao Master qual é o endereço do servidor que escolheu e depois de obter o endereço, pede ao servidor para alocar o novo *PadInt*. O servidor primário cria um *PadInt* inicializado a zero, sem *locks* e pede ao secundário para fazer o mesmo, só respondendo à Biblioteca, com um *ack*, depois de ter recebido o *ack* do secundário. Por fim a Biblioteca cria o *Stub* do *PadInt* para retornar ao cliente;
- *PadInt AccessPadInt(int UID)*: A Biblioteca calcula qual o servidor primário onde está alocado o *PadInt*, pede ao Master o seu endereço e pergunta ao servidor se tem o *PadInt*. Caso a resposta seja afirmativa, a Biblioteca retorna ao cliente uma nova instância do *Stub* do *PadInt*. Caso contrário retorna *null*;
- *int Read()*: A Biblioteca envia um pedido de leitura para o servidor primário onde está alocado o *PadInt*. O servidor primário invoca o método *obterLockLeitura(TID,UID)* e se obtiver o *lock* de leitura (ou se já possuir o lock de escrita), obtém o valor actual do *PadInt*. De seguida o servidor primário envia uma mensagem ao secundário para que execute o método *obterLockLeitura(TID,UID)*, de modo a que os estados fiquem coerentes. Depois do servidor executar o método e enviar um *ack* ao primário, o valor do *PadInt* é retornado ao cliente. No caso em que o *lock* de leitura não foi obtido, o servidor primário insere a transacção nos *Leitores à espera* desse *PadInt*, sendo os passos anteriores executados mal a transacção saia dos *Leitores à espera* e entre nos *Leitores*;
- *void Write(int value)*: Este método é semelhante ao *Read()* mas, altera o valor actual do *PadInt* em vez de

o ler e retorna do servidor primário para Biblioteca um *ack* em vez de um *PadInt*.

4 Algoritmos propostos

4.1 Locking

4.1.1 obterLockLeitura(TID, UID)

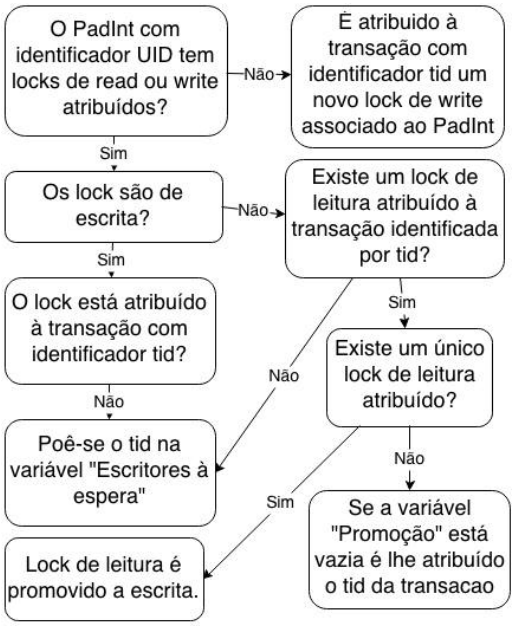

Figure 1. Obtenção de locks de escrita

4.1.2 obterLockLeitura(TID, UID)

Caso não exista no servidor um *lock* de leitura ou escrita sobre o *PadInt* identificado pelo *UID* atribuído à transacção com *TID*, verifica-se se existe um *lock* de escrita e se existir é posto na variável *Leitores à espera* o *TID*. Caso contrário é atribuído à transacção um *lock* de leitura do *PadInt*.

4.1.3 libertarLockEscrita(TID, UID)

O servidor remove o *lock* de escrita da transacção identificada por *TID*, associada ao PadInt identificado por *UID*.

É invocado o método *tiraQueueEscrita(UID)*, para que outras transações possam associar outros locks ao PadInt.

4.1.4 tiraQueueEscrita(UID)

Se no *PadInt* identificado pelo *UID*, existir na variável *Promoção* uma transacção a promover, essa transacção é da *Promoção* e é invocado o método *obterLockEscrita(TID,UID)*. Caso contrário, é verificado se existe na fila de *locks* de escrita pendentes do *UID* alguma transacção pendente:

- Se existir, o *TID* dessa transacção é removida dos *Escritores à espera* e é invocado o método *obterLockEscrita(TID,UID)*.
- Se não existir e se existe nos *Leitores à espera* algum *TID* que esteja associado a *UID*, esse *TID* é removido e é invocado o método *obterLockLeitura(TID,UID)*.

4.1.5 libertarLockLeitura(TID, UID)

O servidor remove o *lock* de leitura, da transacção identificada pelo *TID*, associada ao *PadInt* identificado por *UID*.

É invocado o método *tiraQueueLeitura(UID)*, para que outras transações possam associar outros locks ao PadInt.

4.1.6 tiraQueueEscrita(UID)

No método *tiraQueueLeitura(UID)* verifica-se se existe um único *lock* de read associado ao *PadInt* identificado por *UID* e caso exista, na variável *Promoção* algum *TID* pendente, esse *TID* é removido e é invocado o método *obterLockEscrita(TID,UID)*. No caso em que não existe nenhum *TID* na variável *Promoção*, mas existe alguma transação na variável *Escritores à espera* é removido uma transacção dessa variável e é invocado o método *obterLockEscrita(TID,UID)*.

4.1.7 escrevePadInt(TID, UID, value)

Este método chama o método *obterLockEscrita(TID, UID)* e quando o *lock* de escrita é atribuído, o valor é escrito. De seguida é retornado um *ack* à Biblioteca. Se um deadlock provocar um abort, é lançada uma excepção.

4.1.8 lePadInt(TID, UID)

Este método chama o método *obterLockLeitura(TID, UID)* e quando o *lock* de leitura é colocado na variável *Leitores*, o valor do *PadInt* é lido, sendo depois retornado à Biblioteca. No caso em que ocorre um abort devido a um deadlock é lançada uma excepção

4.2 Commit

Quando é invocado o método TxCommit, são percorridos os pares *(UID, servidor)* na estrutura descrita na Tabela 5, enviando a cada servidor primário um pedido para que faça commit de todos os *PadInts* que foram acedidos para leitura ou escrita durante o decorrer da transação atual.

Ao receber o pedido de commit, o servidor primário percorre a lista de identificadores de *PadInt*s envolvidos no commit e, para cada um deles, verifica se o *TID* recebido

como argumento pertence a alguma das variáveis da classe *PadInt* ilustradas na Tabela 3: Leitores, Escritor, Leitores à espera, Escritores à espera, Promoção.

Se *TID* que identifica a transacção estiver contido nalguma das variáveis acima descritas, este é removido dessa variável. É também criado e guardado um par *(TID,valor final da transação)*. Este par é usado se eventualmente o servidor primário deixar de responder, o secundário assumir o papel de primário e a Biblioteca lhe pedir para responder a um commit/abort de uma transacção terminada, mas à qual o primário (agora em modo *Fail*), nunca chegou a enviar uma mensagem de *ack*.

O servidor primário faz o pedido ao servidor secundário para que execute o commit, invocando o mesmo método com os mesmos argumentos. Depois de executar o pedido do primário, o secundário envia um *ack* ao primário a confirmar que executou o método.

O servidor primário recebe o *ack* do secundário e reporta à Biblioteca o sucesso do commit.

Se na etapa anterior o *TID* não pertence a nenhuma das variáveis anteriormente descritas e o *TID* da transacção a ser tratada é o mesmo que foi registado da última vez que se guardou o par descrito acima, então o valor registado no par como resultado final da transacção é devolvido como retorno à Biblioteca e o par é apagado.

É importante referir que no passo em que são removidos os *locks*, a motivação para se verificar se o *TID* se encontra na lista de variáveis acima referidas e não apenas nas variáveis *Leitores* e *Escritor*, prende-se com o facto de não existir nenhuma forma de impedir que o cliente tente obter locks e faça commit ou abort à transacção antes sequer de os ter obtido.

4.3 Abort

O método *TxAbort* é em tudo semelhante ao método *TxCommit*, a diferença é que antes de libertar cada *lock* de escrita associado a cada *UID* referenciado pela transação, o valor actual do *PadInt* é substituído pelo valor registado como sendo o valor original antes da transação o ter alterado, i.é., é reposto o valor do último commit realizado com sucesso.

4.4 Detecção de deadlocks

Para a detecção de deadlocks usamos um temporizador para cada *PadInt* existente no servidor. Este temporizador é activado quando se coloca um pedido em espera, seja para promoção de lock ou para obter lock de escrita/leitura. Quando o temporizador expira é feito abort da transação[1] (ou transações no caso de existirem locks de leitura atribuídos) que possuía o lock e escolhe-se um pedido dos que estão em espera para ser executado pela seguinte ordem: pedido de promoção, pedido de lock de escrita e finalmente pedido de lock de leitura.

[1] Admitindo que existem mais transacções à espera de serem promovidas ou para lerem/escreverem

5 Tolerância a Faltas - Fail, Freeze e Recover

Sempre que um método do servidor é invocado é verificado se este se encontra no estado freeze, fail ou normal de forma a ser realizado o comportamento adequado.

O servidor primário envia uma mensagem de *i'm alive* ao respectivo servidor secundário a cada x segundos. Caso o secundário não receba a mensagem após o tempo limite, este regista-se no Master como primário e cria uma nova instância de secundário.

Quando o antigo servidor primário, que recebeu o pedido de freeze ou fail e não enviou um *i'm alive* ao respectivo secundário, volta ao estado normal(recebeu recover) verifica se o secundário já assumiu o papel de servidor primário:

- Se não aconteceu, é enviado um *i'm alive* para o secundário. No caso de ter estado em freeze executa os pedidos que registou e continua o seu funcionamento normal, no caso de ter estado em fail continua a operar de forma normal enquanto primário.
- Se aconteceu, independentemente do estado anterior ser fail ou freeze, o servidor termina a sua execução.

Na situação inversa, ou seja, o servidor secundário não enviou a resposta a um pedido no intervalo de tempo máximo, o servidor primário cria um novo servidor secundário e quando o antigo servidor secundário receber recover, termina a sua execução independentemente do estado anterior.

No caso em que recebeu recover, vindo do estado de freeze e o servidor primário ainda está a espera da resposta, executa os pedidos pendentes, responde ao primário e continua a operar de forma normal.

O caso em que o servidor secundário retorna do estado fail, por opção nossa não acontece. A alternativa seria esperar que o servidor voltasse e fazê-lo terminar de seguida, no entanto, como à partida sabemos que se o secundário falhou, assim que o primário atender um pedido o seu estado vai ficar inconsistente, então este termina assim que receber o pedido de fail, parecendo ao servidor primário que não respondeu no intervalo de tempo máximo previsto.

6 Conclusão

A nossa solução garante as propriedades ACID para transações, consistência sequencial e tolera a falha de um servidor.