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

Este artigo elaborado no âmbito da disciplina de Plataformas para Aplicações Distribuídas na Internet, tem como objectivo fazer uma apresentação e descrição da solução escolhida para a implementação do projecto *PADI-DSTM*.

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, como não sabemos qual é a relação existente entre o número de leituras e escritas, podemos admitir que existe igual número de leituras e escritas. Neste caso o número de conflitos existentes poderá ser grande.

Considerando que isto seria um potencial bottleneck de performance no nosso sistema, optámos por abandonar as soluções que usam timestamps ou two-phase-commit (2PC). Para além disto 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 seguir 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.

2.3 Arquitectura

3 Estruturas de Dados

3.1 Master

Esta classe é o gestor do 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 for substituído, o endereço registado é actualizado e os clientes continuam a conseguir aceder aos *PadInt*s sem perturbações. Para além disto, os *TID* são atribuídos de forma sequencial e única.

De seguida apresenta-se a estrutura interna 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 quando o servidor entra no modo *Freeze*
Réplica	Endereço do outro servidor
Valor original	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. Esta classe é composta por:

Variável	Descrição
UID	Identificador do inteiro 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 Lock

Um *Lock* é apenas o *TID*.

3.5 Biblioteca

Esta é a classe que representa a biblioteca usada pelos clientes para comunicar com o sistema de memória distribuida. 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 seu respectivo *Servidor*
Cache de Servidores	Estrutura que mapeia *UID* no *Servidor* em que o *PadInt* está armazenado
Temporizadores	Lista de *timers* para cada servidor

Table 4. Atributos da classe Biblioteca

De seguida apresenta-se alguns métodos da Biblioteca[1]

[1] Os métodos de Commit e Abort serão descritos na secção de algoritmos

4 Algoritmos propostos

4.1 Locking

4.1.1 obterLockEscrita(TID, UID)

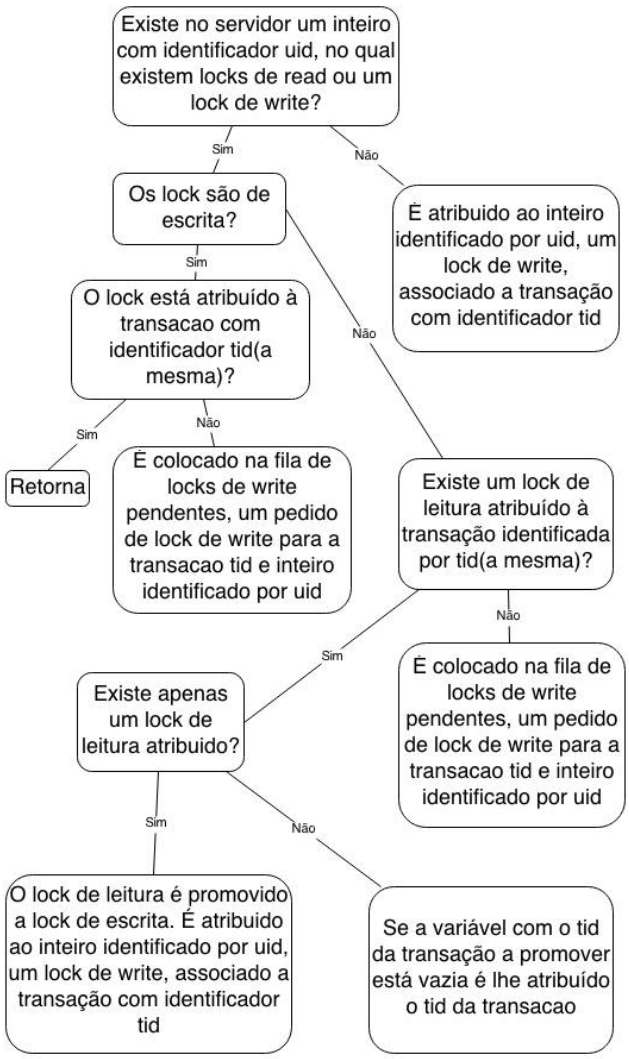

Figure 1. Algoritmo para obter locks de escrita

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*, é verificado se existe um *lock* de escrita relativo ao *PadInt* identificado por *UID*. Se existir é colocado na variável *Leitores à espera* desse PadInt o *TID* da transacção. No caso de não existir é atribuído à transacção com identificador *TID* um *lock* de leitura do *PadInt*.

4.1.3 libertarLockEscrita(TID, UID)

O servidor (seja primário ou secundário) remove o *lock* de escrita da transacção identificada por *TID*, associado ao inteiro identificado por *UID*.

É invocado o método *tiraQueueEscrita(UID)*, com o argumento *UID*, visto que se o *lock* que foi removido estava associado a esse inteiro, logicamente os pedidos a retirar da fila devem ser relativos a colocar *locks* sobre o mesmo inteiro.

4.1.4 tiraQueueEscrita(UID)

Se no *PadInt* identificado pelo *UID* existir na variável *Promoção* uma transacção a promover, essa transacção é removida da fila 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 removido da fila e é invocado o método *obterLockEscrita(TID,UID)*.
- Se não existir e se existe na fila de *locks* de read pendentes, algum *TID* que esteja associado a *UID* esse pedido é removido da fila e é invocado o método *obterLockLeitura(TID,UID)*.

4.1.5 libertarLockLeitura(TID, UID)

O servidor (seja primário ou secundário) remove o *lock* de leitura, da transacção identificada pelo *TID*, associado ao *PadInt* identificado por *UID*.

É invocado o método *tiraQueueLeitura(UID)*, com o argumento *UID*, visto que se o *lock* que foi removido estava associado a esse inteiro, logicamente os pedidos a retirar da fila devem ser relativos a colocar *locks* sobre o mesmo inteiro.

4.1.6 tiraQueueEscrita(UID)

No método *tiraQueueLeitura(UID)* verifica-se se existe apenas um *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 colocado na variável *Escritor*,

o valor é escrito. No final da escrita é retornado um *ack* à Bilbioteca. No caso em que ocorre um abort devido aos deadlocks 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 aos deadlocks é lançada uma excepção

4.2 Commit

Quando é invocado o método TxCommit, são percorridos os pares *(UID, servidor)* na estrutura descrita na Tabela 4, 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 seguintes 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, Boolean*[2]*)*. Este par é usado se eventualmente o servidor primário entrar no estado de *Fail*, o secundário assumir o papel de primário e a *Lib* 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*.

Se 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 um par (tid, valor final atribuido a um PadInt) no final da ultima transacção, então o valor registado no par como resultado final da transacção é devolvido como retorno à lib e o par é apagado.

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 a mensagem de *ack* do secundário e reporta à Biblioteca o sucesso da execução do commit.

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 tente faça commit ou abort à transacção antes sequer de os ter obtido.

4.3 Detecção de deadlocks

Para a detecção de deadlocks usámos um temporizador para cada *PadInt* existente no servidor. Este temporizador é activado quando se coloca algum pedido em espera, seja para promoção de lock ou para obter lock de escrita/leitura. Quando o temporizador expira é feito abort da transação[3] (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.

4.4 Recuperação de aborts

O método *TxAbort* é em tudo semelhante ao método *TxCommit*, a única 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.

5 Tolerância a Faltas

5.1 Fail

5.2 Freeze

6 Conclusão

[2] O boolean indica se o commit teve sucesso ou não

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