

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. Concluiremos abordando a tolerância a faltas e com resultados experimentais.

2 Solução

2.1 Optimista vs Pessimista

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

Constatando esse potencial bottleneck de performance no nosso sistema e o facto das transacções poderem abortar (se surgir algum conflito) depois de já terem realizado trabalho, o que levaria a refazer esse trabalho, optámos por abandonar as soluções optimistas.

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

2.2 Replicação activa vs replicação passiva

Depois de escolhermos qual o protocolo a usar, deparamo-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 mais servidores do que os 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.

A lista *registered servers* tem como finalidade registar os servidores primários existentes, assim como, permitir identificar quais os *PadInts* atribuídos a cada servidor. Garantindo, que no caso em que o servidor primário é substituído, o endereço registado é actualizado e os clientes continuam a aceder aos *PadInts* sem perturbações. Para além disso, os identificadores das transacções (*TID*) são atribuídos de forma sequencial e única.

Quando um servidor é criado ele regista-se no *Master* de forma a que lhe seja atribuído um papel (servidor primário ou secundário). Tendo em conta que esta solução usa replicação passiva é necessário existir um número par de servidores para atribuir um papel. Desta forma quando o primeiro servidor do par se regista ele irá ficar com estado *FailedState*, pois ainda não tem um papel atribuído e não é suposto responder a nenhum pedido por parte do cliente. Quando o segundo servidor do par se regista é lhe atribuído o papel de servidor secundário (o servidor irá ter um estado *BackupServer*). O servidor secundário irá de seguida atribuir o papel de servidor primário (estado *PrimaryServer*) ao primeiro servidor do par. Esta atribuição do papel de servidor primário é feita pelo servidor secundário de forma a diminuir a carga no *Master*.

| Variável | Descrição |
|-----------------------------|--|
| Last transaction identifier | Último <i>TID</i> atribuído |
| Registered servers | Estrutura que armazena <i>Server Registry</i> referente a cada servidor primário registado no master |

Table 1. Atributos da classe Master

3.1.1 Server Registry

A classe *Server Registry* é responsável por armazenar o identificador e endereço de cada servidor primário registado no master. Para além disto, guarda também uma lista com os identificadores dos *PadInts* (UID) que foram atribuídos ao servidor.

Na nossa solução optámos por guardar apenas os servidores primários no *Master* em detrimento de guardar também os servidores secundários. Estes últimos precisam de ser conhecidos apenas pelos servidores primários, sendo que o master necessita apenas de lhes atribuir o seu papel (de servidor secundário) quando se tentam registar.

3.1.2 LoadBalancer

Com o objectivo de distribuir a carga pelos vários servidores criamos a classe *Load Balancer*. Esta classe é usada, pelo *Master*, para decidir qual o servidor primário a que deve ser atribuído um *PadInt* na sua criação. Para esta distribuição o critério escolhido foi atribuir o novo *PadInt* ao servidor primário que actualmente tem menos *PadInts* atribuídos.

Esta classe é também utilizada na redistribuição dos *PadInts* quando um novo par (primário, secundário) de servidores é criado. Nessa altura os *PadInts* são distribuídos de forma a que cada par de servidores fique com um número de *PadInts* aproximadamente igual. Este número resulta da média entre o número total de *PadInts* criados e o número de pares de servidores existentes.

3.2 Server

O conjunto das instâncias da classe *Server* representam a memória distribuída onde são armazenados os *PadInts*.

Um dos atributos desta classe é uma instância da classe *Server machine*. Esta a classe encapsula o *Server* de forma a permitir simular que este deixou de funcionar em resposta ao pedido de *Fail*. Permite ainda reiniciá-lo como se trata-se de um novo *Server* acabado de criar.

Quando o *Server* precisa de realizar uma leitura ou escrita num dado *PadInt* usa os métodos descritos nas secções 4.1.1 e 4.1.2 para obter o lock respectivo e poder depois efectuar a leitura ou escrita, respectivamente.

| Variável | Descrição |
|----------------|------------------------------|
| Identifier | Identificador do servidor |
| Address | Endereço do servidor |
| Server state | Estado do servidor |
| Server machine | Classe que contém o servidor |

Table 2. Atributos da classe Servidor

3.2.1 Server State

A classe servidor contém um dos seguintes estados: *primary*, *backup*, *failed*, *froze*. Indicando respectivamente que o servidor é o servidor primário, secundário, recebeu pedido *Fail* ou *Freeze*. Cada estado modela o comportamento do servidor, dependendo do papel que lhe foi atribuído.

| Variável | Descrição |
|-----------------------|--|
| padInt Dictionary | Estrutura que mapeia <i>uid</i> em <i>PadInt</i> |
| Im alive timer | Timer usado para o envio e recepção de mensagens <i>Im alive</i> |
| Pair server reference | Referência para o outro servidor do par |
| Pair server address | Endereço do outro servidor do par |

Table 3. Atributos da classe Server State

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 <i>PadInt</i> que representa |
| Actual value | Valor no momento actual da transacção |
| Original value | Valor no início da transacção |
| Lock type | Tipo do lock (leitura ou escrita) atribuído actualmente |
| Readers | Lista de transacções com locks de leitura atribuídos |
| Writer | Transacção com lock de escrita atribuído |

Table 4. Atributos da classe PadInt

3.4 Stub do PadInt

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

| Variável | Descrição |
|----------|---|
| UID | Identificador do <i>PadInt</i> que representa |
| TID | Identificador da transacção |
| ServerID | Identificador do <i>Server</i> onde o <i>PadInt</i> está guardado |
| Address | Endereço do <i>Server</i> onde o <i>PadInt</i> está guardado |
| Cache | Referência para a <i>Cache</i> da <i>Library</i> |

Table 5. Atributos da classe Stub do PadInt

Esta classe disponibiliza os seguintes métodos:

- *int Read()*: lê o valor guardado na *Cache* se este estiver disponível. Isto é, já foi feito anteriormente um pedido de leitura ou escrita ao *Server* e foi obtido o *lock* de leitura ou escrita, respectivamente. Caso contrário, efectua a leitura no *Server* e guarda o valor em *Cache*.
- *void Write(int value)*: escreve o valor na *Cache* se este estiver disponível. Isto é, já foi feito anteriormente um pedido de escrita ao *Server* e foi obtido o *lock* de escrita. Caso contrário, efectua a escrita no *Server* e guarda o valor em *Cache*.

3.5 Biblioteca

A biblioteca usada pelos clientes para comunicar com o sistema de memória distribuída encontra-se representada na classe *Library*. Esta classe tem a seguinte estrutura:

| Variável | Descrição |
|---------------|--|
| Master server | Referência para o <i>Master</i> |
| Actual tid | Identificador da transacção atribuído pelo <i>Master</i> |
| Cache | Cache usada para guardar os valores temporários dos <i>PadInts</i> |
| Channel | Canal usado pela biblioteca |

Table 6. Atributos da classe Library

De seguida apresentam-se alguns métodos da Biblioteca:

- *bool init()*: cria o canal a ser usado pela *Library* e inicializa a variável *master server*. Quando o canal é criado é também definido um valor de *timeout* ao fim do qual a *Library* deixa de estar bloqueada à espera de resposta a pedido que efectuou;
- *bool TxBegin()*: a Biblioteca pede ao *Master* para criar um novo *TID* para a transacção e regista-o. É criada uma nova *Cache*;
- *bool TxCommit()*: descrito na secção 4.2.1;
- *bool TxAbort()*: descrito na secção 4.2.2;
- *PadInt CreatePadInt(int UID)*: a *Library* pede ao *Master* para registar o *PadInt* e se este não tiver já sido registado anteriormente o *Master* retorna um tuplo contendo o identificador e endereço do *Server* onde esse *PadInt* deverá ser criado. De seguida, o *Server* primário, em resposta ao pedido da *Library*, cria um *PadInt* inicializado a zero, sem *locks* e pede ao secundário para fazer o mesmo, só respondendo à *Library*, com um *ack*, depois de ter recebido o *ack* do

secundário. Por fim, a *Library* insere o *PadIntRegistry* (classe usada para guardar o *PadInt* na *Cache*), relativo ao *PadInt*, na cache, criando de seguida o *Stub* do *PadInt* para retornar ao cliente;

- *PadInt AccessPadInt(int UID)*: a *Library* pergunta ao *Master* qual é o *Server* onde está o *PadInt*. Caso o *PadInt* exista, o *Master* retorna um tuplo contendo o identificador e endereço do *Server* onde esse *PadInt* se encontra. De seguida a *Library* pergunta ao *Server* se tem o *PadInt*. Caso a resposta seja afirmativa, a *Library* insere o *PadIntRegistry*, relativo ao *PadInt*, na cache, e retorna ao cliente uma nova instância do *Stub* do *PadInt*. Caso contrário é lançada uma excepção;

3.5.1 Cache

Com o objectivo de melhorar o desempenho relativo a leituras e escritas criámos a classe *Cache*. Esta classe permite guardar valores temporários relativos a cada *PadInt* usado pelo cliente. A ideia base do funcionamento das leituras e escritas usando a *Cache* foi já descrita anteriormente na secção 3.4. Para tornar definitivos os valores escritos na *Cache* antes de ser efectuado o *commit*, descrito na secção 4.2.1, todos os *PadInt* acedidos para escrita são escritos no *Server* que os armazena.

Uma das vantagens obtidas com o uso da *Cache* é que se reduz drasticamente o número de pedidos efectuados ao *Server*, reduzindo assim a carga a que este está sujeito. Outra vantagem é reduzir o tempo consumido nas transacções, pois as leituras e escritas são mais rápidas devido a serem realizadas localmente.

4 Algoritmos propostos

4.1 Locking

Para a obtenção de locks a classe *PadInt*, referente a um dado *PadInt* identificado por *UID*, disponibiliza um conjunto de métodos cujo funcionamento será explicado de seguida. Estes métodos são utilizados pela classe *Server* para responder a pedidos de leitura ou escrita.

4.1.1 GetReadLock(TID)

Quando este método é invocado começa-se por verificar se o lock de leitura pretendido pela transacção, identificada por *TID*, já lhe foi atribuído ou se já lhe foi atribuído um lock de escrita. Nesse caso, retorna-se *true*. Caso contrário, é chamado o método *AcquireLock*, descrito na secção 4.1.3, com o tipo de lock pretendido, neste caso leitura.

4.1.2 GetWriteLock(TID)

Primeiro verifica-se se o lock de escrita pretendido pela transacção, identificada por *TID*, já lhe foi atribuído. Nesse caso, retorna-se *true*. Caso contrário, é chamado o método *AcquireLock*, descrito na secção 4.1.3, com o tipo de lock pretendido, neste caso escrita.

4.1.3 AcquireLock(TID, requiredLockType)

Este método é invocado para obter locks de leitura ou escrita, consoante o valor do argumento *requiredLockType*. O argumento *TID* identifica a transacção que está a tentar obter o lock.

Se for possível obter o lock do tipo pedido, a variável *lockType* da classe *PadInt* é actualizada para o novo tipo de lock, *requiredLockType*. De seguida, caso o tipo de lock pedido seja leitura, o *TID* é adicionado à variável *readers*. Caso contrário, isto é o tipo de lock pedido é de escrita, é verificado se se trata de uma promoção, isto é, a transacção já tinha o lock de leitura e pretende obter o lock de escrita e nesse caso, o *TID* é removido da variável *readers*. Por fim, é atribuído à variável *writer* o valor *TID*.

Quando não é possível obter o lock o pedido é posto em espera no máximo durante um determinado intervalo de tempo, usando o método *Wait* da classe *Monitor* disponibilizado pela linguagem C#. Assim que é efectuado um *commit* ou *abort*, descrito nas secções 4.2.1 e 4.2.2, todos os pedidos são retirados da fila de espera e aos que for possível tentar adquirir o lock este ser-lhes-á atribuído. Os restantes voltarão a ficar em espera. Se ao fim do intervalo de tempo máximo estipulado um pedido ainda se encontra em espera então foi detectado um *deadlock* e esse pedido recebe uma excepção a indicar que tem que abortar.

Um pedido fica em espera num de dois casos. O primeiro é quando o tipo de lock actualmente atribuído e o lock pedido são ambos do tipo escrita. O segundo ocorre quando o tipo de lock actualmente atribuído e o pedido são diferentes e a transacção não está a tentar realizar uma promoção possível, isto é, se o tipo de lock atribuído é de leitura e a transacção está a tentar obter um lock de escrita e não é possível realizar a promoção porque não é só esta a transacção que possui o lock de leitura.

4.1.4 FreeWriteLock(TID)

Remove o lock de escrita da transacção identificada por *TID*, associado ao *PadInt*. Neste passo basta apenas inutilizar a variável *writer*, atribuindo-lhe um valor que não identifique numa transacção.

4.1.5 FreeReadLock(TID)

É removido o lock de leitura, da transacção identificada pelo *TID*, associado ao *PadInt* identificado por *UID*. Neste passo basta apenas remover o *TID* da variável *readers*.

4.2 Commit e abort

Antes de explicarmos como funciona o commit e o abort explicaremos uma opção tomada em relação ao uso de two-phase-commit (2PC). Tendo em conta o âmbito deste projecto e a abordagem pessimista que decidimos seguir, tal como descrevemos na secção 2.1, não precisamos de usar 2PC.

Esta conclusão é baseada no facto que quando um cliente deseja realizar commit ou abort este já obteve previamente todos os locks que necessitou e ainda não os libertou. Devido a isto quando um cliente deseja realizar commit ou abort, mesmo que tenha efectuado vários pedidos a *Servers* diferentes basta que faça commit ou abort em cada um desses *Servers* de forma a libertar os locks só passando ao *Server* seguinte quando obtiver a confirmação que o commit ou abort num dado *Server* foi bem sucedido.

Mesmo no caso em que um *Server* falha ou os *PadInts* são redistribuídos para outro *Server* em consequência de ter sido criado um novo par (primário, secundário) o cliente quando recebe uma excepção a informar que o que o servidor não respondeu ou que o *PadInt* não foi encontrado, respectivamente, este pergunta novamente ao *Master* qual é o *Server* que guarda actualmente o *PadInt*. Depois disto volta a re-enviar o pedido e irá obter uma resposta, pois ou o *Server* foi recuperado devido à replicação passiva ou o cliente agora sim enviou o pedido para o *Server* que guarda o *PadInt*.

4.2.1 Commit

Quando é invocado o método *TxCommit* da *Library*, caso tenham sido criados ou acedidos *PadInts*, são efectuados dois passos. O primeiro passo corresponde à escrita de todos os valores escritos em *Cache* e encontra-se descrito na secção 3.5.1. No segundo passo, é enviado a cada *Server* um pedido de commit contendo todos os *UID* dos *PadInts* acedidos nesse *Server* e a *Cache* é re-iniciada.

O *Server* ao receber este pedido verifica se guarda todos os *PadInts* identificados pelos *UID* recebidos. Caso não guarde algum deles lança uma excepção a indicar que não guarda o *PadInt*. Caso contrário, usando os métodos da classe *PadInt* descritos nas secções 4.1.5 e 4.1.4 são libertados os locks de leitura ou escrita atribuídos à transacção que está a efectuar o commit.

4.2.2 Abort

O método *TxAbort* da *Library* é em tudo semelhante ao método *TxCommit* excepto em dois pontos. A primeira diferença é que antes ser libertado 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, isto é, é reposto o valor do último commit realizado com sucesso. A segunda diferença está relacionada com a *Cache*, pois não existe a escrita dos valores nos *Servers*, devido à primeira diferença enunciada.

5 Tolerância a Faltas - Fail, Freeze e Recover

De forma a existir tolerância a faltas cada *Server* tem um estado, podendo assim suportar diferentes comportamentos, tal como referido anteriormente na secção 3.2.1. Se o estado do *Server* for *PrimaryServer*, isto é assume o papel de servidor primário, então envia uma mensagem de *I'm alive* ao respectivo servidor secundário (servidor com estado *BackupServer*) a cada 10 segundos. Caso o servidor secundário não receba a mensagem após o tempo limite (15 segundos), este regista-se no Master como primário e cria uma nova instância de secundário.

A detecção de falha do servidor secundário por parte do servidor primário é também feita através de um temporizador. Este temporizador é iniciado quando na sequência de um pedido feito ao servidor primário este tenta replicar o pedido no servidor secundário, de forma a actualizá-lo. Se o servidor secundário não responder após o tempo limite de 15 segundos o servidor primário cria uma nova instância de secundário.

De forma a suportar os pedidos de *Fail* e *Freeze* existem os estados *FailedState* e *FrozeState*, respectivamente. Quando um *Server* recebe o pedido de *Fail* passa a ter o estado *FailedState* e é desconectado do canal, no entanto, no âmbito deste projecto, para permitir a recriação de um *Server* a máquina (*ServerMachine* referida na secção 3.2) continua disponível de forma a que o outro *Server* do par (primário, secundário) possa criar uma nova instância de um *Server* na máquina onde o servidor que recebeu o pedido *Fail* estava a correr. Ainda no âmbito deste projecto assumimos que quando um *Server* recebe um pedido de *Fail* a instância de *Server* é removida e não pode ser recuperada usando o pedido *Recover*.

Um *Server* ao passar para o estado *FrozeState*, devido ao pedido *Freeze*, faz com que todos os pedidos enviados pelo cliente ou por outro *Server* fiquem bloqueados. Caso o antigo estado do *Server* fosse *PrimaryServer* deixará também de enviar mensagem de *I'm alive* ao respectivo servidor secundário. Existem duas formas para o *Server* abandonar este estado. Uma está relacionada com o pe-

dido de *Recover* que um cliente pode enviar, nesse caso o servidor voltará ao estado antigo (ao que tinha antes de receber o pedido de *Freeze*) e responde a todos os pedidos que estavam pendentes. A outra forma foi já referida anteriormente no início desta secção e está relacionada com a detecção de falha do outro *Server* do par (primário, secundário).

6. Avaliação

De forma a realçar as vantagens da nossa solução apresentamos de seguida comparações entre o que a nossa solução permite atingir e o que aconteceria se outras opções tivessem sido tomadas.

6.1 Cache

Para demonstrar o efeito positivo que a *Cache* tem na nossa solução apresentamos de seguida duas tabelas que mostram as diferenças de usar ou não a *Cache*. Este exemplo pressupõe a existência de um cliente que acede a dois *PadInts* guardados no mesmo *Server* e efectua uma sequência de 300 leituras e escritas a intercaladas a cada um dos *PadInts*.

Os dados apresentados nas tabelas confirmam que o número de pedidos que o *Server* recebe com o uso da *Cache* é muito menor do que quando não é utilizada uma *Cache*. Para além destes dados foi também possível verificar que ao usar a *Cache* o tempo consumido pelo cliente foi menor relativamente ao caso em que a *Cache* não foi usada.

É importante referir que estes efeitos se verificam sempre para leituras. Em relação às escritas estes resultados apenas se verificam para um número mínimo de duas escritas. Quando o cliente faz apenas uma escrita devido ao uso da *Cache* serão efectuadas duas escritas, uma para obter o lock de escrita e outra antes de efectuar o commit.

| UID do <i>PadInt</i> | Leituras no <i>Server</i> | Escritas no <i>Server</i> | Total |
|----------------------|---------------------------|---------------------------|-------|
| 1 | 300 | 300 | 600 |
| 2 | 300 | 300 | 600 |
| Total | 600 | 600 | 1200 |

Table 7. Número de pedidos ao *Server* sem uso de *Cache*

| UID do <i>PadInt</i> | Leituras no <i>Server</i> | Escritas no <i>Server</i> | Total |
|----------------------|---------------------------|---------------------------|-------|
| 1 | 1 | 2 | 3 |
| 2 | 1 | 2 | 3 |
| Total | 2 | 4 | 6 |

Table 8. Número de pedidos ao Server com uso de Cache

6.2 Redistribuição de *PadInts*

De forma a demonstrar a redistribuição da carga quando são criados novos pares (primário, secundário) de *Servers* realizámos um teste. No início do teste para além do *Master* é criado um par (primário, secundário) de *Servers*. Existem dois clientes A e B. O cliente A cria 30 *PadInts* que serão guardados pelo único par de *Servers* que existe. De seguida o cliente B é executado, sendo que este acede aos *PadInts* criados pelo cliente A para realizar uma escrita em cada um deles. Por fim, repete-se duas vezes a criação de um novo par de *Servers* e a execução do cliente B.

| Número de pares de <i>Servers</i> | Número de <i>PadInt</i> no <i>Server</i> | Escritas por <i>Server</i> |
|-----------------------------------|--|----------------------------|
| 1 | 30 | 30 |
| 2 | 15 | 15 |
| 3 | 10 | 10 |

Table 9. Distribuição da carga pelos servers

Como se pode verificar pelos resultados obtidos a carga vai sendo distribuída pelos *Servers* à medida que novos pares (primário, secundário) de *Servers* são criados.

6.3 Aborts

Para identificar a taxa de aborts de transações submetemos a nossa solução a três testes. Antes de mais salientamos que a forma como estes testes foram realizados não tiram partido da utilização da *Cache*, pois nunca são feitas duas ou mais leituras ou escritas no mesmo *PadInt* durante uma transação. No primeiro destes testes os clientes realizaram 3 leituras e 3 escritas por iteração em três *PadInts*, durante 300 iterações (por iteração entende-se iniciar a transação, realizar as leituras e escritas e realizar o commit no final). Durante este teste variámos o número de clientes e pares de *Servers* (primário, secundário) de forma a cobrir vários casos. Os resultados são apresentados na tabela 10. Como se pode verificar o número de aborts é bastante menor no caso em que existem apenas dois clientes e aumenta conforme o crescimento do número de clientes. Salientamos

que neste teste todos os clientes efectuem leituras e escritas nos mesmos *PadInts*, sendo por isso normal o número de aborts observados.

| N. de Pares de Servers | N. de Clientes | | | | |
|------------------------|----------------|----|-----|-----|-----|
| | 2 | | 3 | | |
| | C1 | C2 | C1 | C2 | C3 |
| 1 | 6 | 6 | 136 | 125 | 131 |
| 2 | - | - | 153 | 154 | 164 |

Table 10. Número de aborts tendo 3 leituras e 3 escritas por iteração

Durante o segundo teste foram realizadas duas leituras e oito escritas por cada iteração, no total de dez. No terceiro teste foram realizadas oito leituras e duas escritas também no total de dez iterações. Os resultados apresentados nas tabelas 11 e 12 confirmam as conclusões suportadas pelo primeiro teste.

| N. de Pares de Servers | N. de Clientes | | | | |
|------------------------|----------------|----|----|----|----|
| | 2 | | 3 | | |
| | C1 | C2 | C1 | C2 | C3 |
| 1 | 0 | 0 | 4 | 5 | |

Table 11. Número de aborts tendo 2 leituras e 8 escritas por iteração

| N. de Pares de Servers | N. de Clientes | | | | |
|------------------------|----------------|----|----|----|----|
| | 2 | | 3 | | |
| | C1 | C2 | C1 | C2 | C3 |
| 1 | 1 | 0 | 4 | 5 | 2 |

Table 12. Número de aborts tendo 8 leituras e 2 escritas por iteração

7 Conclusão

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

Podemos concluir também que o uso de cache e a redistribuição dos *PadInts*, aquando da criação de um novo par (primário, secundário) de *Servers* permite reduzir drasticamente o número de pedidos efectuados ao *Server*, reduzindo assim a carga a que este está sujeito. Outra vantagem é reduzir o tempo consumido nas transações, pois as leituras e escritas são mais rápidas devido a serem realizadas localmente.