Um Mecanismo de *Caching* para o Protocolo SCORe*

João A. Silva, João M. Lourenço, and Hervé Paulino

CITI—Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
{jaa.silva}@campus.fct.unl.pt
{joao.lourenco,herve.paulino}@fct.unl.pt

Resumo Os protocolos de replicação parcial de dados apresentam um grande potencial de escalabilidade. O SCORe é um protocolo para replicação parcial proposto recentemente que faz uso de controlo de concorrência multi-versão. Neste artigo abordamos um dos principais problemas que afeta o desempenho deste tipo de protocolos: a localidade dos dados, i.e., pode-se dar o caso do nó local não ter uma cópia dos dados a que pretende aceder, e assim ser necessário realizar uma ou mais operações de leitura remota. Assim, a não ser que se empreguem técnicas para melhorar a localidade no acesso aos dados, o número de operações de leitura remota aumenta com o tamanho do sistema, acabando por afetar o desempenho do mesmo. Nesse sentido, introduzimos um mecanismo de *caching* que permite replicar cópias de dados remotos de maneira a que seja possível servir localmente dados remotos enquanto que se mantém a consistência dos mesmos e a escalabilidade oferecida pelo protocolo. Avaliamos o mecanismo de *caching* com um *benchmark* conhecido da literatura e os resultados experimentais mostram resultados animadores com algum aumento no desempenho do sistema e uma redução considerável da quantidade de operações de leitura remota.

Palavras Chave: Cache; Replicação Parcial de Dados; Memória Transacional Distribuída; Controlo de Concorrência, Sistemas Distribuídos

1 Introdução

Uma das estratégias possíveis para abordar a questão da disponibilidade e escalabilidade das soluções de software passa pelo recurso à distribuição do código e dos dados por vários nós computacionais. Muitos dos sistemas que suportam esta estratégia requerem a partilha de dados entre processos residentes em diferentes nós computacionais, o que tem implicações não negligenciáveis no desenho

* Este trabalho foi parcialmente financiado pela Fundação para a Ciência e Tecnologia (FCT/MEC) no contexto do projeto de investigação PTDC/EIA-EIA/113613/2009 e PEst-OE/EEI/UI0527/2011, e do projecto estratégico PEst-OE/EEI/UI0527/2014—040402112.

da solução e no desempenho do sistema que a implementa. Neste contexto, a memória transacional distribuida (MTD) [10] apresenta-se como um mecanismo de controlo de concorrência de alto nível que oferece a semântica transacional sobre um espaço de endereçamento distribuído virtualmente partilhado (*distributed shared memory addressing space*). A MTD traz a simplicidade do modelo de memória transacional (MT) [9, 21] para o contexto distribuído,

A maioria dos sistemas correntes de MTD fazem uso ou de *replicação total de dados* [4, 5, 24] ou de algum tipo de *distribuição* [12, 17]. Nenhum dos trabalhos anteriores endereça a questão da replicação parcial de dados [1] em MTD, onde cada nó apenas replica um subconjunto dos dados existentes no sistema. Kim et al. [11] usa replicação parcial de dados em MTD, mas este não é o foco do trabalho, que aborda o escalonamento de transações em memória.

Uma abordagem baseada em replicação parcial permite uma melhor utilização dos recursos de armazenamento de dados do sistema (disco e/ou memória central), enquanto que simultaneamente disponibiliza ainda algum suporte (parametrizável) para tolerância a falhas. Adicionalmente, esta estratégia reduz o nível de coordenação entre os nós do sistema no momento de confirmação (*commit*) das transações: apenas os nós que replicam os dados manipulados pela transação necessitam de ser envolvidos no processo de validação e confirmação da mesma.

A questão da replicação parcial de dados foi endereçada nos sistemas de bases de dados distribuídas [18, 23], incluindo repositórios de dados do tipo chave/valor [15, 19]. Apesar de alguns sistemas de bases de dados utilizarem a memória central (RAM) como repositório de dados (mais frequente ainda no caso dos sistemas chave/valor), os requisitos e desafios levantados pela replicação parcial de dados diferem consideravelmente daqueles encontrados nos sistemas de MTD para linguagens de programação de propósito geral. A maior diferença encontra-se na forma como os dados são acedidos e manipulados: enquanto que os sistemas chave/valor são operados através de uma interface muito simples `put/get/delete`, numa linguagem de programação de propósito geral os dados são acedidos e manipulados através de referências à memória. Neste contexto de distribuição parcial de dados em MTD, colocam-se as seguintes questões: (i) que dados deverão ser replicados parcialmente?; (ii) como expressar qual o nível de replicação desejado numa linguagem de programação de propósito geral?; (iii) como replicar parcialmente dados (objetos) cujas relações formam grafos complexos?; e (iv) como assegurar o acesso eficiente a dados remotos?, i.e., a dados que não estão replicados no nó local.

Trabalho anterior [22] endereçou alguns destes desafios estendendo uma infraestrutura de MTD, a TribuDSTM [24], de formar a suportar replicação parcial de dados. Esta extensão incluiu uma anotação Java `@Partial` que permite ao programador especificar que dados deverão ser replicados parcialmente. Construiu-se assim um sistema que permite um grau configurável de replicação de dados, que vai da distribuição à replicação total.

Embora a replicação parcial de dados possua alto potencial de escalabilidade, a sua eficiência pode ser seriamente afetada se os acessos aos dados não

mostrarem algum nível de localidade. Neste contexto, por vezes os nós não têm uma cópia local dos dados a que se pretende aceder e é necessário realizar uma ou mais operações de leitura de objetos remotos. Este tipo de operações requer comunicação através da rede, cujo *overhead* não é negligenciável. Por outras palavras, a não ser que se empreguem técnicas para melhorar a localidade no acesso aos dados, o número de operações de leitura remota aumenta com o tamanho do sistema, acabando por afetar o desempenho do mesmo.

Uma estratégia possível para maximizar a eficiência da replicação parcial é recorrer a técnicas de *caching*, que replicam dados remotos que são acedidos frequentemente, de forma a minimizar a comunicação inter-nós. Este tipo de técnicas pode melhorar significativamente o tempo de acesso a objetos remotos, particularmente em *workloads* dominados por leituras. No entanto, integrar um mecanismo de *caching* num sistema que faz uso de replicação parcial e oferece um modelo de consistência de dados forte está longe de ser uma tarefa óbvia.

Assim, este artigo endereça o último dos desafios apresentados acima (*como assegurar o acesso eficiente a dados remotos?, i.e., a dados que não estão replicados no nó local*), propondo e avaliando um mecanismo de *caching* de objetos remotos, específico para o protocolo de confirmação SCORe [14]. O SCORe é um protocolo distribuído de confirmação de transações para ambientes de replicação parcial de dados. É baseado no protocolo de confirmação em duas fases (2PC) e oferece *1-copy-serializability* (1CS) como modelo de consistência de dados.

As contribuições deste artigo são portanto: (1) o desenho de um mecanismo de *caching* para o protocolo SCORe, inspirado no trabalho de Pimentel et al. [16] (para o protocolo de replicação parcial GMU [15]), e (2) a sua implementação no contexto da plataforma TribuDSTM, adaptando as estratégias de disseminação da invalidação de dados na cache ao esquema de particionamento dos dados suportado pela plataforma.

O resto deste documento está organizado da seguinte maneira. A Secção 2 apresenta o modelo do sistema e uma visão geral do protocolo SCORe. O mecanismo de *caching* proposto é apresentado na Secção 3. Os resultados da avaliação experimental são apresentados e discutidos na Secção 4. A Secção 5 discute o trabalho relacionado. Finalmente, a Secção 6 conclui este artigo.

2 Modelo do Sistema e Visão Geral do SCORe

Consideramos um modelo de sistema distribuído assíncrono clássico composto por $\Pi = \{n_1, \ldots, n_k\}$ nós. Os nós comunicam por troca de mensagens e não têm acesso a nenhum tipo de memória partilhada nem a relógios globais. As mensagens podem sofrer atrasos arbitrariamente longos (mas finitos) e não assumimos limites quanto ao poder de computação dos nós nem quanto ao desvio dos seus relógios. Consideramos ainda o modelo de falhas clássico *crash-stop*: os nós podem falhar por *crash* mas não manifestam comportamentos maliciosos.

Cada nó n_i replica um subconjunto dos dados do sistema, os quais são identificados univocamente por um identificador único. Cada item de dados d é representado por um tuplo $\langle k, val, ver \rangle$, onde k é o seu identificador único, val é

o seu valor e *ver* é uma estampilha temporal que aumenta monotonicamente e identifica (e ordena totalmente) as versões do item d.

Abstraímo-nos da colocação dos dados assumindo que estes são divididos por p partições de dados, e que cada partição é replicada por r nós, i.e., r representa o fator de replicação de cada item de dados. Representamos por $\Gamma = \{g_1, \ldots, g_p\}$ o conjunto de grupos de nós g_j que replica a partição de dados j. Dizemos que um grupo g_j é dono da partição de dados j e assumimos que para cada partição existe um nó chamado *mestre*, representado por $master(g_j)$. Cada grupo é composto por exatamente r nós (para garantir o fator de replicação desejado), dos quais pelo menos um é assumido como correto. De maneira a maximizar a flexibilidade da estratégia de colocação de dados, não é necessário que os grupos sejam disjuntos (i.e., podem conter nós em comum), e assumimos que um nó possa participar em vários grupos, desde que $\bigcup_{j=1\ldots p} g_j = \Pi$. Dado um grupo g_j, representamos como $data(g_j)$ o conjunto de itens de dados replicados pelos nós que constituem o grupo g_j. Salientamos que este modelo permite capturar um largo espetro de estratégias de partição de dados.

Modelamos transações como sequências de operações de leitura e escrita em itens de dados encapsuladas num bloco atómico. Cada transação é originária num nó $n_i \in \Pi$, e pode ler/escrever dados pertencentes a qualquer partição (mesmo que não replicada localmente). Também, não é assumido qualquer conhecimento *a priori* quanto aos itens de dados lidos ou escritos pelas transações. Dado um item de dados d, representamos como $replicas(d)$ o conjunto de nós que replicam d (i.e., os nós do grupo g_j que replicam a partição de dados que contém d). Dada uma transação T, representamos como $participants(T)$ o conjunto $\bigcup_{d \in \mathcal{F}} replicas(d)$, onde $\mathcal{F} = readSet(T) \cup writeSet(T)$ (i.e., o conjunto de itens de dados lidos ou escritos por T, respetivamente).

2.1 Visão Geral do SCORe

Antes de apresentarmos o mecanismo de *caching* proposto, fornecemos uma breve visão geral do protocolo base sobre o qual este foi construído: o protocolo SCORe. Direcionamos o leitor para [14] onde é apresentado o protocolo SCORe de uma maneira mais extensa e fornecemos, de seguida, uma descrição sucinta do mesmo destinada a permitir o fácil entendimento do funcionamento da nossa extensão ao protocolo.

Tal como nos sistemas de controlo de concorrência multi-versão clássicos, no SCORe [14] cada nó n_i mantém para cada item de dados uma lista com o histórico de versões desse mesmo item, onde cada versão é marcada com um número de versão (estampilha temporal) usado para ordenar totalmente as operações de confirmação de transações que modificaram o respetivo item. No seu núcleo, o SCORe gere um esquema de estampilhas temporais distribuído que permite estabelecer que versões estão visíveis para uma certa transação e qual a ordem de serialização das transações de escrita.

A operação de confirmação de uma transação T atribui a $T.ts^C$ uma estampilha temporal que representa o seu ponto de serialização. As versões que são visíveis por T são determinadas via uma estampilha temporal associada a T que

representa o seu *snapshot* (chamada ts^S), a qual é estabelecida no momento da primeira leitura de T. A partir desse momento, qualquer operação de leitura executada por T pode observar a versão mais recente do item de dados com número de versão menor ou igual a ts^S, como em controlo de concorrência multi-versão clássico.

Para chegar a um consenso acerca do destino de T, o SCORe baseia-se no 2PC. Após receber a mensagem de preparação, cada participante n_i (representados pelo conjunto $participants(T)$) adquire os trincos de leitura/escrita sobre os itens de dados lidos/escritos por T, para os quais n_i tem uma cópia local. Em seguida, cada participante valida o conjunto de leitura de T e envia o seu voto para o coordenador. Se T for validada com sucesso por n_i, então n_i propõe um ponto de serialização para T (o valor de $T.ts^C$). O valor da estampilha temporal proposta é um mais recente do que qualquer valor observado até então por n_i (guardado pelo SCORe na variável $commitId$). O coordenador coleciona os votos de todos os participantes e escolhe o máximo $T.ts^C$ proposto para T, ou aborta T caso algum voto tenha sido negativo. Por último, a mensagem de confirmação é enviada para todos os participantes. Quando é recebida, cada participante escreve o conjunto de escrita de T para as listas de versões e torna-as disponíveis para novas transações lerem.

As operações de leitura requerem a determinação de qual versão, de entre as existentes, deve ser visível para uma dada transação. Isto é conseguido usando as seguintes regras:

R1 Limite inferior de *snapshot* Em cada operação de leitura num nó n_i, o SCORe verifica se n_i está suficientemente atualizado para servir a transação T, i.e., se n_i já finalizou todas as transações que foram serializadas antes de T de acordo com $T.ts^S$. Isto é conseguido através do bloqueio de T até $T.ts^S$ ser maior ou igual do que a transação mais recentemente confirmada em n_i;

R2 Limite superior de *snapshot* A fim de maximizar a frescura dos dados acedidos, no momento da primeira operação de leitura da transação T, $T.ts^S$ é afetado com a estampilha temporal mais recente observada até ao momento por n_i, estabelecendo um limite superior sobre a frescura do *snapshot* que pode ser observado por T; e

R3 Seleção da versão Tal como nos sistemas de controlo de concorrência multi-versão clássicos, sempre que existem várias versões de um determinado item de dados, a versão selecionada é a mais recente com número de versão menor ou igual a $T.ts^S$.

Na Figura 1 podemos ver um exemplo de utilização das regras de leitura já descritas. Esta representa um sistema com três nós (N_0, N_1 e N_2), sendo que N_1 replica o item de dados A e N_2 replica o item de dados B. Todos os nós têm $commitId = 1$ quando a transação T_0 é iniciada no nó N_1. Esta começa com $ts^S = 1$ e é confirmada com $ts^C = 2$ criando novas versões para ambos os itens de dados A e B. Depois do *commit* de T_0 em N_1, T_1 é iniciada em N_0 onde a sua primeira operação é a leitura de A (replicado apenas por N_1). Aplicando a regra R2, depois da leitura de $A(2)$ em N_1, $T_1.ts^S$ é afetado com a estampilha temporal mais recente observada até ao momento, que neste caso é o valor 2

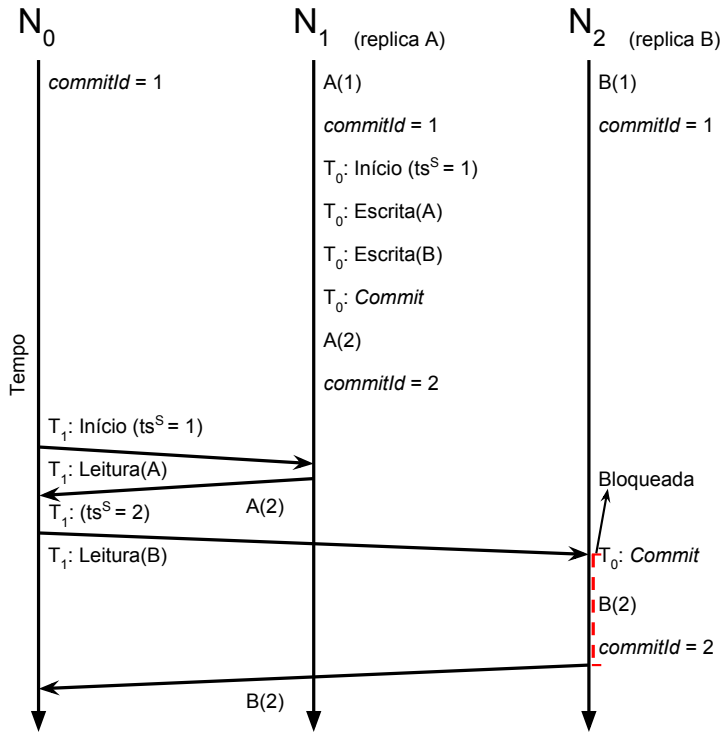


Figura 1: Aplicação das regras de leitura no protocolo SCORe.

(dois). Depois de T_1 ler $A(2)$, existe uma leitura de B (que é replicado por N_2). No entanto, uma vez que o *commit* de T_0 demorou mais tempo em N_2 do que em N_1, a nova versão de B ainda não é visível por T_1. Pela aplicação da regra R1, T_1 fica bloqueada esperando pelo *commit* de T_0 em N_2, e só depois é que retorna a nova versão de B, $B(2)$. Em ambas as leituras a regra R3 também é aplicada porque A e B são as versões mais recentes em ambos os nós e são visíveis por T_1.

3 *Caching* no Protocolo SCORe

Fazer cache de objetos remotos levanta dois grandes problemas: (i) é necessário garantir que do acesso a dados em cache não resultam violações das garantias de consistência oferecidas; e (ii) é importante que a utilização do mecanismo de *caching* não tenha um impacto negativo sobre a frescura dos dados observados pelas transações (ou pode acabar por piorar o desempenho do sistema, ao invés de melhorá-lo).

Nesta secção endereçamos estes problemas e descrevemos como os resolvemos no mecanismo de *caching* proposto. Este mecanismo é constituído por dois com-

ponentes: (i) um algoritmo de consistência da cache, que permite determinar se uma transação pode ler os dados em cache preservando as garantias de consistência oferecidas pelo SCORe; e (ii) um mecanismo de invalidação que pode utilizar diferente estratégias de disseminação com vista à maximização da frescura dos dados mantidos em cache e, consequentemente, a melhoria da eficácia do mecanismo de *caching*.

3.1 Assegurar a Consistência dos Dados

Para facilitar as operações de leitura, os dados em cache são mantidos da mesma forma que os dados normais (i.e., não armazenados em cache) num *container* de dados multi-versão. No entanto, ao contrário dos dados normais, cada item de dados d em cache é representado por uma sequência de versões

$$\langle k, val, version, validity \rangle$$

onde k é o identificador único de d; val é o valor desta versão de d; $version$ é uma estampilha temporal com o número de versão deste valor de d; e $validity$ é uma estampilha temporal com o número de versão até à qual esta versão é válida (i.e., representa o limite inferior sobre o *snapshot* mais recente em que este valor é o mais fresco para d). A informação dada por $version$ e $validity$ permite que o mecanismo de *caching* siga as regras que determinam a visibilidade das versões no SCORe.

O pseudo código presente no Algoritmo 1 descreve o comportamento de uma operação de leitura no nó local.

Quando uma transação T tenta ler um item de dados, ela primeiro verifica a localidade desse mesmo item. Se o item de dados d é local, a operação de leitura pode ser executada localmente. Caso contrário, d é considerado remoto. Neste caso, com a introdução do mecanismo de *caching*, agora T primeiro inquire a cache acerca de d e só então, se d não for encontrado, T executa uma operação de leitura remota.

Se o item de dados d for encontrado (Linha 10), então o mecanismo de *caching* necessita de verificar se existe alguma versão v (de d) que foi criada antes de T. Isto é conseguido verificando simplesmente o valor de $v.version$ e de $T.ts^S$, e se T já leu algum outro item de dados. Se esta é a primeira operação de leitura feita por T (Linha 12), a versão mais recente pode ser lida com segurança (regra R2, Secção 2.1). Caso não seja, tal como em controlo de concorrência multi-versão clássico, é selecionada a versão mais recente com o valor de $version$ menor ou igual a $T.ts^S$ (Linhas 14-18), como dita a regra R3 da Secção 2.1.

Quando v é encontrada (Linha 3), ainda é necessário fazer uma verificação adicional para garantir que é seguro para T ler v. O valor de $T.ts^S$ é comparado com o valor de $v.validity$ (Linha 5) e se esta verificação falhar, v é considerada demasiado velha, indicando que pode existir uma versão mais recente deste item de dados nos nós remotos (que replicam d) e que ainda não é conhecida pelo nó local. Caso contrário, T pode ler v em segurança.

Se alguma das verificações falhar é forçado um *cache miss* (retornando um valor nulo) e é necessário executar uma operação de leitura remota.

Algorithm 1 Operação de leitura no nó local.

```
1: function readCache(Key k, Timestamp ts, boolean firstRead)
2:     Version v ← getVisibleVersion(k, ts, firstRead)
3:     if v ≠ null then
4:         Timestamp val ← v.validity
5:         if ts < val then
6:             return v
7:     return null

8: function getVisibleVersion(Key k, Timestamp ts, boolean firstRead)
9:     Versions vers ← cache.getVersions(k)
10:    if vers ≠ null then
11:        Version v ← vers.mostRecent
12:        if firstRead then
13:            return v
14:        while v ≠ null do
15:            if v.version ≤ ts then
16:                return v
17:            else
18:                v ← v.prev
19:    return null
```

O mecanismo de *caching* introduziu também algumas operações que têm de ser levadas a cabo pelos nós remotos aquando da resposta a pedidos de leitura remota. Agora, além de enviar o item de dados solicitado, os nós remotos também enviam o respectivo valor de *validity* (para a versão enviada).

Se a versão respondida v for a mais recente, a sua validade é o valor da estampilha temporal da última transação de escrita que foi confirmada nesse nó. Caso contrário, $v.validity$ é afetada com o valor de $v.next.version$, i.e., $v.validity$ é afetada com a estampilha temporal da versão mais recente a seguir a v.

3.2 Maximizar a Eficácia da Cache

De acordo com o Algoritmo 1, uma transação T pode ler uma versão v com segurança da cache apenas se $v.validity$ garantir que v é suficientemente fresca dado $T.ts^S$, i.e., se $T.ts^S$ for menor do que $v.validity$. Caso contrário, é necessário forçar um *cache miss* de maneira a garantir que nenhuma versão mais fresca foi escrita por alguma transação que devia ser serializada antes de T e cujas modificações T devia observar.

Por outro lado, a fim de garantir que as transações observam as versões mais recentes, o valor de ts^S das transações é avançado em dois casos: (i) no início de uma transação, o seu valor de ts^S é afetado com a estampilha temporal da última transação de escrita que foi confirmada (no nó); e (ii) por ocasião da primeira operação de leitura de uma transação.

Uma vez que o valor de ts^S das transações é constantemente atualizado a fim de maximizar a frescura dos dados observados pelas transações, o valor de *validity* das versões em cache também necessita de ser atualizado se quisermos maximizar a chance da cache ser capaz de atender aos pedidos de leitura das transações. Isto é conseguido atraves do mecanismo de invalidação da cache descrito nos Algoritmos 2 e 3.

A função getInvalidationSet (Algoritmo 2) descreve as operações realizadas por um nó para construir um conjunto de invalidação ($iSet$), i.e., um conjunto que contém os identificadores de todos os itens de dados que o nó remetente replica localmente e que foram atualizados desde a última vez que um $iSet$ foi construído e enviado.

Ao executar a função getInvalidationSet, o nó recolhe duas estampilhas temporais (Linhas 2 e 3): (i) a variável $mostRecent$ é afetada com a estampilha temporal da última transação de escrita que foi confirmada no nó ($commitId$); e (ii) a variável $lastSent$ é afetada com a estampilha temporal do momento quando o último $iSet$ foi enviado (que é guardada pela cache). A cache mantém o registo de que itens de dados foram modificados desde o envio do último $iSet$ (o conjunto $committedItems$, na Linha 6). E o $iSet$ é composto apenas pelos itens de dados que são replicados pelo grupo g_j ao qual o respetivo nó pertence, i.e., $\bigcup_{d \in data(g_j)} d \in committedItems$.

Algorithm 2 Operação de invalidação no nó remetente.

```
1:  function getInvalidationSet( )
2:      Timestamp mostRecent ← commitId
3:      Timestamp lastSent ← cache.lastSent
4:      Set iSet ← ∅
5:      if mostRecent > lastSent then
6:          Set committedItems ← cache.committedItems
7:          for all item ∈ committedItems do
8:              if isLocal(item) then
9:                  iSet ← iSet ∪ {item}
10:         return [iSet, mostRecent, groupId]
11:         cache.lastSent ← mostRecent
12:     return null
```

Os conjuntos de invalidação podem ser enviados/disseminados de uma maneira assíncrona, usando diferentes estratégias de disseminação. Nós implementámos três estratégias de disseminação diferentes:

Eager um $iSet$ é disseminado sempre que uma transação é confirmada;
Batch um $iSet$ é disseminado com uma frequência fixa (configurável); e
Lazy um $iSet$ é enviado juntamente com a resposta a um pedido de leitura remota (apenas para o nó que fez o pedido).

Finalmente, o Algoritmo 3 apresenta a lógica de invalidação executada quando um nó recebe uma mensagem de invalidação. Uma mensagem de invalidação contém um $iSet$, a estampilha temporal $mostRecent$ da transação mais recentemente confirmada no momento em que o $iSet$ foi construído, e o identificador $group$ do grupo ao qual o nó remetente pertence.

Quando é recebida uma mensagem de invalidação, esta desencadeia a execução da função invalidateCache($iSet, mostRecent, group$). Invalidar os itens de dados contidos no $iSet$ significa informalmente que as validades de todas as versões (mais recentes) de itens em cache, replicados pelo grupo $group$, que não

estão incluídos no $iSet$ (i.e., que não foram atualizados desde a receção do último $iSet$) têm de ser extendidas.

Esta tarefa pode ser feita iterando sobre todas as versões em cache e identificando quais as validades que devem ser atualizadas. Mas, de modo a ser mais eficiente, a lógica de invalidação associa uma única validade partilhada, representada por $mostRecentValidities[group]$, a todos os itens de dados replicados pelo grupo $group$ cujas versões em cache são reconhecidas como estando atualizadas aquando da construção do $iSet$.

Ao executar a função INVALIDATECACHE($iSet, mostRecent, group$) são realizadas duas operações: (i) todos os itens de dados contidos no $iSet$ são separados da validade partilhada (Linhas 2-7), clonando o seu valor para uma validade privada; e (ii) o valor de $mostRecentValidities[group]$ é atualizado para $mostRecent$ (Linhas 8-12).

Algorithm 3 Operação de invalidação no nó recetor.

```
1:  function invalidateCache(Set iSet, Timestamp mostRecent, int group)
2:      for invalidItem ∈ iSet do
3:          Versions vers ← cache.getVersions(invalidItem)
4:          if vers ≠ null then
5:              Version v ← vers.mostRecent
6:              if v.validity.isShared( ) then
7:                  v.validity ← [v.validity.validity, false]
8:      Validity mrv ← mostRecentValidities[group]
9:      if mrv = null then
10:         mostRecentValidities[group] ← [mostRecent, true]
11:     else
12:         mrv.validity ← mostRecent
```

4 Resultados Experimentais

Nesta secção apresentamos alguns resultados de um estudo experimental feito ao mecanismo de *caching* proposto para o protocolo SCORe, que foi integrado na infraestrutura TribuDSTM. Estes resultados endereçam duas questões: (1) qual é o impacto do mecanismo de *caching* no desempenho do sistema?, e (2) na quantidade de operações de leitura remota?

4.1 Instalação Experimental

A avaliação foi realizada num sistema com 6 nós computacionais heterogéneos. Três dos nós têm a seguinte especificação: 2 × Quad-Core AMD Opteron 2376 2.3 GHz e 16 GB de RAM. Os outros três: 1 × Quad-Core Intel Xeon X3450 2.66 GHz (com *Hyper-Threading*) e 8 GB de RAM. Todos os nós executam Linux 2.6.26-2-amd64 (Debian 5.0.10) e estão ligados em rede via *Ethernet Gigabit* privada. A plataforma Java utilizada foi OpenJDK 6 (IcedTea 1.8.10). No que toca ao sistema de comunicação em grupo foi utilizado o JGroups versão 3.4.1 final [2].

4.2 *Benchmarks*

Para avaliar o mecanismo de *caching* proposto utilizámos um *benchmark* conhecido da literatura.

Árvore Vermelha-Preta. Este *benchmark* é composto por três transações que operam sobre uma árvore binária vermelha-preta: (i) inserções, que adicionam um elemento novo na árvore (se ainda não presente); (ii) remoções, que removem um elemento da árvore (se este existir); e (iii) pesquisas, que procuram na árvore por um elemento específico. As inserções e remoções são consideradas transações de *escrita*. O *benchmark* é caracterizado por transações muito curtas e rápidas e que realizam pouco trabalho e apresenta baixa contenção. Todas as inserções, remoções e procuras são feitas com elementos aleatórios, por isso este *benchmark* não apresenta nenhuma localidade nos acessos aos dados.

4.3 Considerações sobre a Implementação

Como referido na Secção 1, o mecanismo de *caching* proposto foi integrado numa infraestrutura de MTD [22] para uma linguagem de programação de propósito geral (Java), na qual foi implementado o protocolo SCORe. Na implementação de uma memória transacional por software (MTS) replicada parcialmente, optou-se por um esquema simples onde cada nó apenas pertence a um único grupo e cada item de dados é replicado por um único grupo de nós.

O trabalho de Pimentel et al. [16] implementa um mecanismo de *caching* para o protocolo GMU [15]. O nosso trabalho inspira-se nesse trabalho e adapta os algoritmos existentes em [16], na medida em que o GMU usa um esquema de controlo de concorrência baseado em relógios vetoriais enquanto que o SCORe faz uso de relógios escalares.

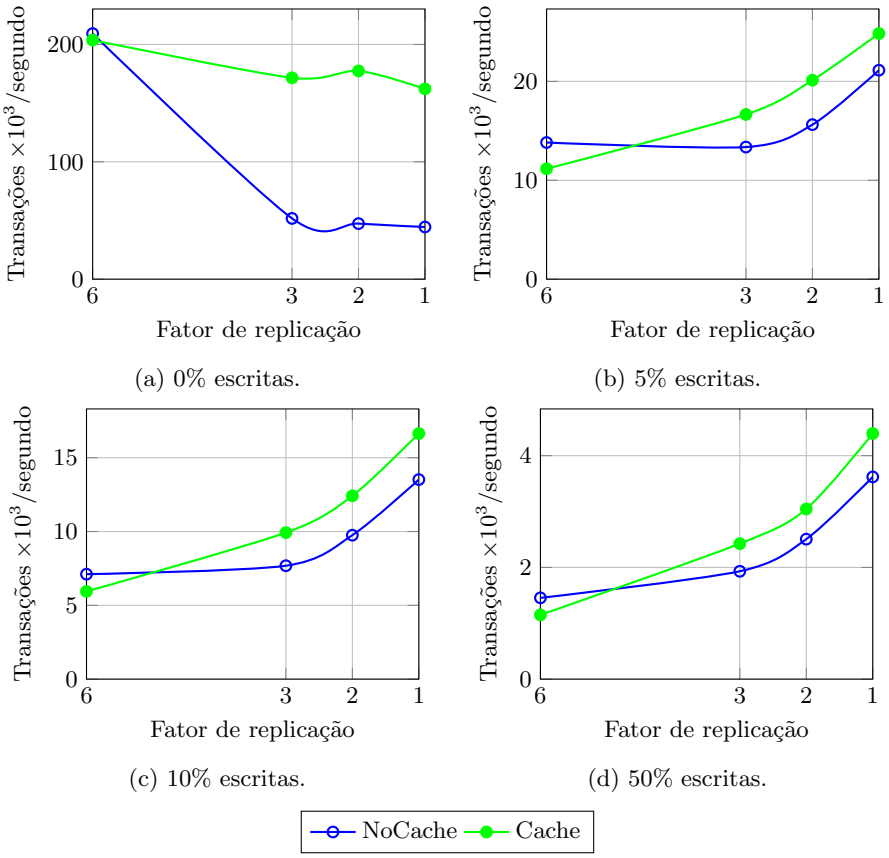
Ainda em [16], o mecanismo de invalidação é executado em todos os nós. Mas, tendo em conta o esquema implementado na nossa infraestrutura, isso iria fazer com que fossem enviadas várias mensagens iguais. Uma vez que cada grupo replica os mesmo itens de dados, todos os nós pertencentes a um mesmo grupo enviariam os mesmos conjuntos de invalidação (já que todos os nós de um mesmo grupo "vêem" as mesmas transações). Por isso, apenas o nó mestre de cada grupo g_j, $master(g_j)$, dissemina os conjuntos de invalidação nas estratégias *eager* e *batch*. Já na estratégia *lazy* todos os nós enviam os conjuntos de invalidação, uma vez que os conjuntos são enviados juntamente com a resposta a um pedido de leitura remota apenas para o nó que fez esse mesmo pedido de leitura.

4.4 Resultados

Ambas as Figuras 2 e 3 mostram resultados experimentais para o *benchmark* da Árvore Vermelha-Preta e todos os pontos dos gráficos referentes à linha Cache são para a estratégia de disseminação *batch*.

A Figura 2 mostra o desempenho do sistema com a cache ativada e desativada (Cache e NoCache nas figuras, respetivamente).

Um *workload* com 0% de escritas (Figura 2a) é o cenário perfeito para um mecanismo de *caching*, uma vez que os dados não são modificados. Assim, quando os dados entram na cache todos os acessos de leitura a esses mesmos dados podem ser executados localmente, como se fossem replicados no nó local. Ao aumentar a percentagem de escritas, começamos a ver que o mecanismo de *caching* passa ser menos rentável, mas continua a melhorar o desempenho geral do sistema (que começa a ser afetado pelas regras de leitura de maneira a manter a consistência dos dados).



(a) 0% escritas.

(b) 5% escritas.

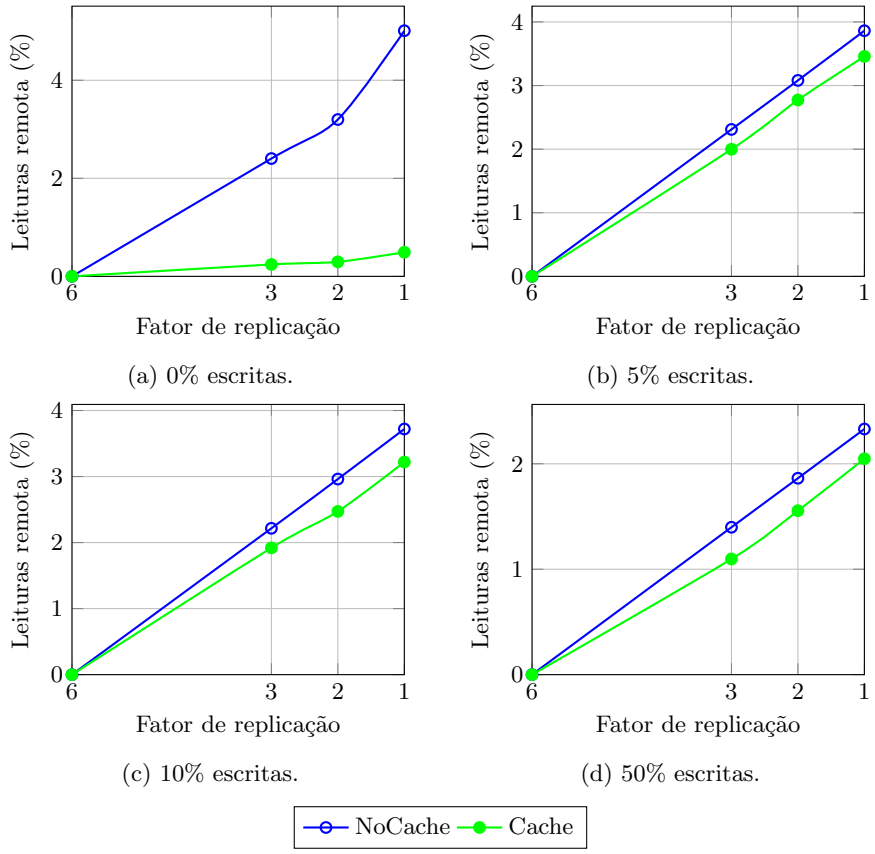
(c) 10% escritas.

(d) 50% escritas.

Figura 2: Desempenho no *benchmark* da Árvore Vermelha-Preta

Em todos os gráficos da Figura 2, com um fator de replicação igual a 6 (seis) a linha do desempenho do sistema sem cache supera o desempenho do sistema com cache. Uma vez que todos os nós replicam todos os dados (i.e., replicação total), a cache só introduz *overhead* nas operações de leitura.

A Figura 3 mostra a percentagem de operações de leitura remota do sistema para a mesma experiência que na Figura 2. E como mostrado na Figura 2a, sem escritas o mecanismo de *caching* reduz drasticamente a quantidade de pedidos de leituras remotas feitos pelos nós (Figura 3a). Com o aumento da percentagem de escritas continuamos a ver uma redução na quantidade de pedidos de leituras remotas, mas cada vez mais pequena.



(a) 0% escritas.

(b) 5% escritas.

(c) 10% escritas.

(d) 50% escritas.

Figura 3: Percentagem de operações de leitura remota no *benchmark* da Árvore Vermelha-Preta.

5 Trabalho Relacionado

É possível identificar um conjunto de variáveis que condicionam os estudos realizados no contexto da replicação de sistemas transacionais, tais como: a natureza

do suporte físico para os dados, o nível de replicação e o nível de consistência entre réplicas.

Considerando a natureza do suporte físico, é possível encontrar estudos dirigidos a sistemas de bases de dados (SBD) em disco [8], a SBD em memória [6] e também a sistemas de memória transacional [5, 17].

O nível de replicação pode variar entre a replicação total dos dados, onde todos os nós gerem uma cópia integral dos dados, e a distribuição (total) dos dados, onde cada item de dados reside num único nó do sistema. No intermédio encontram-se as soluções que exploram a replicação parcial de dados, onde cada nó gere um sub-conjunto próprio dos dados. Estas soluções podem ainda ser diferenciadas de acordo com os nós que são envolvidos na confirmação das transações, sendo apelidados de não-genuínas quando todos os nós do sistema são necessariamente envolvidos e de genuínas quando envolvem apenas os nós que contém réplicas dos dados manipulados pela transação a ser confirmada.

O nível de consistência mais forte, que assegura que o sistema com várias réplicas se comporta como um sistema centralizado (não replicado) é 1CS. No entanto, quer seja para optimização do desempenho quer seja pelas implicações naturais do teorema CAP [3], é frequente que os sistemas transacionais com replicação de dados recorram a níveis de consistência mais fracos, como por exemplo *snapshot isolation* [20] ou consistência eventual [6].

No que respeita às soluções envolvendo replicação parcial de dados, as propostas presentes na literatura podem ser classificadas de acordo com (i) que nós são envolvidos na confirmação das transações (protocolo genuíno ou não-genuíno); e (ii) que garantias de consistência são fornecidas.

No contexto dos SBD, Serrano et al. em [20] argumenta que 1CS impõe fortes limitações à escalabilidade das soluções de replicação e propõe um protocolo não-genuíno um nível de consistência alternativo denominado de *1-copy-snapshot-isolation*, que explora do modelo de *snapshot isolation* para a gestão da consistência entre réplicas. Schiper et al. em [19] propõe uma solução eficiente que garante 1CS para SBD, propondo o primeiro protocolo genuíno. Mais recentemente, foram propostos alguns protocolos genuínos. O GMU [15] foi a primeira proposta de um protocolo genuíno para replicação parcial a garantir que transações de leitura nunca são abortadas ou forçadas a uma confirmação distribuída. O SCORe [14] é em muito semelhante ao GMU e pode até ser visto como uma evolução deste mas que oferece 1CS como modelo de consistência (em vez de *extended update serializability* (EUS) oferecido pelo GMU).

Fazer *caching* de dados remotos que são acedido frequentemente é uma abordagem ortogonal a todos estes sistemas, e que pode ser adotada para maximizar a eficiência das estratégias de particionamento dos dados. Aqui, o principal desafio de implementar um mecanismo de *caching* para um sistema transacional é como preservar a consistência quando são lidos dados da cache (que é replicada assincronamente).

Finalmente, existem ainda outras técnicas que estão relacionadas com cache, tais como o Tashkent [7] ou o AutoPlacer [13], que tentam dinamicamente afinar o mapeamento dos dados para os nós do sistema de maneira a minimi-

zar a frequência dos acessos a dados remotos. Estas técnicas são ortogonais aos mecanismos de *caching* e podem ser usadas em conjunto com estes.

6 Conclusões

Os sistemas que usam abordagens baseadas em replicação parcial de dados podem ser severamente afetados pela ineficiente colocação dos dados, porque esta pode gerar padrões de acesso com baixa localidade no acesso aos dados. Uma solução no caminho para atenuar este problema é o uso de técnicas de *caching* para replicar remotamente dados que são acedidos mais frequentemente.

Neste trabalho, introduzimos uma mecanismo de *caching* no protocolo SCORe. A avaliação mostra algumas melhorias no desempenho do sistema e redução na quantidade de leituras remotas, principalmente em *workloads* dominados por leituras.

Como trabalho futuro, seria importante conduzir um extensivo estudo experimental de maneira a ser possível analisar, através de vários *benchmarks*, qual o impacto das diferentes estratégias de disseminação no desempenho do sistema com diferentes *workloads* (tanto ao nível de leituras e escritas, como com diferentes padrões de acesso/localidade).

Referências

1. G. Alonso. Partial database replication and group communication primitives. In *European Research Seminar on Advances in Distributed Systems*, 1997.
2. Bela Ban. JGroups - A Toolkit for Reliable Multicast Communication. `http://www.jgroups.org`, June 2013.
3. E. A. Brewer. Towards robust distributed systems. In *PODC*, 2000.
4. N. Carvalho, P. Romano, and L. Rodrigues. A generic framework for replicated software transactional memories. In *NCA*, 2011.
5. M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *PRDC*, 2009.
6. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS*, Oct. 2007.
7. S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, 2006.
8. J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD*, 1996.
9. M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.
10. M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. In *DISC*, 2005.
11. J. Kim et al. Scheduling transactions in replicated distributed software transactional memory. In *CCGRID*, 2013.
12. C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP*, 2008.

13. J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. In *ICAC*, 2013.
14. S. Peluso, P. Romano, and F. Quaglia. SCORe: A scalable one-copy serializable partial replication protocol. In *Middleware*. 2012.
15. S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, 2012.
16. H. Pimentel, P. Romano, S. Peluso, and P. Ruivo. Enhancing locality via caching in the GMU protocol. In *CCGRID*, 2014.
17. M. M. Saad and B. Ravindran. Hyflow: A high performance distributed software transactional memory framework. In *HPDC*, 2011.
18. N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. In *Principles of Distributed Systems*. 2006.
19. N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *SRDS*, 2010.
20. D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *PRDC*, 2007.
21. N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 1995.
22. J. A. Silva, T. M. Vale, J. M. Lourenço, and H. Paulino. Replicação parcial com memória transacional distribuída. In *INForum*, 2013.
23. A. Sousa, R. Oliveira, F. Moura, and F. Pedone. Partial replication in the database state machine. In *NCA*, 2001.
24. T. M. Vale, R. J. Dias, and J. M. Lourenço. Uma infraestrutura para suporte de memória transacional distribuída. In *INForum*, 2012.