INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Replicação Parcial para Sistemas de Memória Transaccional por *Software*

Pedro Miguel Pereira Ruivo

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Prof. Doutor João Emílio Pavão Martins
Orientador:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Vogal:	Prof. Doutor Hervé Miguel Cordeiro Paulino

Outubro 2011

Agradecimentos

Gostava de começar por agradecer ao meu orientador, Prof. Luís Rodrigues, pela oportunidade e apoio prestado durante a elaboração desta Dissertação.

Quero também agradecer a toda a equipa do Grupo de Sistemas Distribuídos, nomeadamente ao Dr. Paolo Romano, João Fernandes, Maria Couceiro, Nuno Carvalho, Sebastiano Peluso e Diego Didona, pelo contínuo apoio que me deram, discussão e comentários produtivos durante o desenvolvimento deste trabalho.

Finalmente quero agradecer também aos amigos e colegas, assim como aos meus pais, pela paciência, apoio e encorajamento dado ao longo deste trabalho.

Este trabalho foi parcialmente financiado pela comissão europeia através do projecto "Cloud-TM" (257784), pela FCT através do projecto ARISTOS (PTDC/EIA- EIA/102496/2008), e pelo financiamento plurianual do INESC-ID através de fundos do programa PIDDAC.

Lisboa, Outubro 2011

Pedro Miguel Pereira Ruivo

Para os meus pais.

Luís e Elvira.

Resumo

Actualmente, os sistemas de armazenamento em memória transaccional e distribuída são cada vez mais utilizados como forma de aumentar o desempenho de aplicações com acesso intensivo a grandes quantidades de dados. Neste contexto, a replicação dos dados na memória de múltiplos nós possui duas grandes vantagens: por um lado, permite distribuir a carga das operações de leitura por vários nós; por outro, assegura a sobrevivência dos dados na eventualidade da falha de um dos nós. Estas vantagens necessitam de ser ponderadas contra os custos envolvidos na replicação: os dados consomem memória e quanto maior o número de nós maior será o custo dos manter coerentes. Desta forma, justifica-se a utilização de replicação parcial.

Esta dissertação aborda a aplicação de técnicas de replicação parcial em sistemas de armazenamento em memória transaccional e distribuída. Apesar destas técnicas já terem sido experimentadas em sistemas de bases de dados distribuídas, existem diferenças significativas na caracterização das cargas impostas ao sistema, e no tipo de processamento que é feito na sua execução, que justificam o nosso trabalho. Em particular, os sistemas de gestão de bases de dados são obrigados a executar diversas fases (processamento do SQL, persistência síncrona, etc.) que acarretam custos que não existem nos sistemas de armazenamento em memória. Assim, os custos de coordenação, associados à replicação parcial, são em termos relativos, amplificados. Este trabalho pretende contribuir para aferir em que medida a replicação parcial é viável e eficaz neste contexto.

A dissertação apresenta as seguintes contribuições: descreve um conjunto de algoritmos para a concretização de replicação parcial em sistemas de armazenamento em memória distribuída e transaccional; faz uma avaliação experimental desses algoritmos com base numa adaptação do *Infinispan*, uma cache distribuída de código aberto da RedHat. Ao contrário da solução nativa, baseada no protocolo de confirmação em duas fases, a solução aqui proposta evita o interbloqueio. Resultados experimentais mostram que os algoritmos propostos permitem obter ganhos significativos no desempenho do sistema.

Abstract

Today, transactional in-memory distributed storage systems are widely used as a mean to increase the performance of applications that need to access frequently large quantities of shared data. In this context, data replication has two main advantages: it supports load balancing and fault-tolerance. However, these advantages need to be weighted against the costs of replication, namely memory consumption and coordination costs. This motivates the use of partial replication.

This dissertation addresses the problem of supporting partial replication in distributed in-memory storage systems. Even if there are many proposed solutions for implementing partial replication in database systems, they are not well suited for distributed transactional in-memory systems: in the latter the replicas coordination has a much higher impact on the whole transaction execution time due to a reduced local execution time and the avoidance of queries processing and data durability.

In this context, my thesis proposes a set of solutions specifically tailored for implementing partial replication in distributed in-memory transactional systems and it assesses the efficiency of these solutions compared to the existing ones.

The dissertation presents the following contributions: it proposes a set of algorithms to support partial replication in transactional in-memory distributed storage systems; and presents an experimental evaluation of these algorithms.The experimental evaluation is performed using a prototype implementation that has been integrated within *Infinispan*, an open-source in-memory transactional data-grid by JBoss RedHat. By the results we show that our protocols contribute to a significant performance improvement with respect to the native implementations, with a surplus value of avoiding deadlock scenarios that affects the efficiency of the existing Two-Phase Commit based protocols.

Palavras Chave
Keywords

Palavras Chave

Replicação Parcial

Memória Distribuída

Memória Transaccional

Difusão Atómica

Keywords

Partial Replication

Distributed Memory

Transactional Memory

Atomic Multicast

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos

1CS *1-Copy Serializability*

1CSI *1-Copy Snapshot Isolation*

2PC *Two-Phase Commit*

3PC *Three-Phase Commit*

ABcast Difusão Atómica (do Inglês *Atomic Broadcast*, também conhecida por *Total Order Broadcast*)

ACID Atomicidade, Coerência, Isolamento e Durabilidade

AGGRO *AGGRessively Optimistic*

AMcast Difusão Atómica Selectiva (do Inglês *Atomic Multicast*)

CT Estampilha temporal de confirmação (do Inglês *Commit Timestamp*)

D2STM *Dependable Distributed Software Transactional Memory*

DiSTM *Distributed Software Transactional Memory*

FIFO *First In First Out*

HTM Memória Transaccional por *Hardware* (do Inglês *Hardware Transactionl Memory*)

IP *Internet Protocol*

JTA *Java Transaction API*

JVSTM *Java Versioned Software Transactional Memory*

LAN *Local Area Network*

OABcast Difusão Atómica Optimista (do Inglês *Optimistic Atomic Broadcast*)

OLTP Processamento On-Line de Transacções (do Inglês *On-Line Transaction Processing*)

RAC *Resilient Atomic Commit*

RAM *Random Access Memory*

RBcast Difusão Fiável (do Inglês *Reliable Broadcast*)

RC *Read Committed*

RR *Repeatable Read*

RR+WS *Repeatable Read* com verificação *Write Skew*

RS Conjunto de Leitura (do Inglês, *Read Set*)

SCG Sistema de Comunicação em Grupo

SGBD Sistema de Gestão de Base de Dados

SQL *Structured Query Language*

ST Estampilha temporal de início (do Inglês *Start Timestamp*)

STM Memória Transaccional por Software (do Inglês *Software Transactional Memory*)

TCC *Transactional Coherence and Consistency*

TCP *Transmission Control Protocol*

TPC *Transaction Processing Performance Council*

UDP *User Datagram Protocol*

WS Conjunto de Escrita (do Inglês, *Write Set*

1 Introdução

Esta dissertação aborda a aplicação de técnicas de replicação parcial em sistemas de armazenamento em memória transaccional e distribuída. Embora estas técnicas já tinham sido experimentadas em sistemas de bases de dados distribuídas, existem diferenças significativas na caracterização das cargas impostas ao sistema, e no tipo de processamento que é feito na sua execução, que justificam o nosso trabalho. Em particular, os sistemas de gestão de bases de dados são obrigados a executar diversas etapas (processamento do SQL, persistência síncrona, etc.) que acarretam custos que não existem nos sistemas de armazenamento em memória. Assim, os custos de coordenação relacionados à replicação parcial, são em termos relativos, amplificados. Este trabalho pretende contribuir para aferir em que medida a replicação parcial é viável e eficaz neste contexto.

1.1 Motivação

Recentemente, sistemas de armazenamento em memória transaccional e distribuída são cada vez mais utilizados como forma de aumentar o desempenho de aplicações com acesso intensivo a grandes quantidades de dados. O *Youtube*, *Wikipedia*, *Twitter*, *Facebook* são alguns exemplos de aplicações que fazem uso deste tipo de tecnologia. Uma das principais motivações para a utilização de armazenamento em memória é o aumento do desempenho das aplicações, uma vez que o acesso a informação em memória, mesmo que numa máquina remota, é substancialmente mais rápido do que o acesso ao disco. Estes sistemas não se limitam a suportar operações de leitura; as operações de escrita são também realizadas em memória, sendo a transferência para disco realizada nos "bastidores", de forma diferida.

Para além disso, muitas destas aplicações não necessitam de usar SQL no acesso aos dados, conseguindo operar através de uma interface mais simples, como por exemplo a oferecida por uma tabela de dispersão, o que desencoraja a utilização de soluções baseadas em bases de

dados tradicionais. No entanto, a possibilidade de realizar um conjunto de operações de forma atómica continua a ser útil em muitos padrões de utilização, pelo que sistemas de armazenamento em memória mais recentes oferecem algum suporte para a execução de *transacções em memória* (Aguilera, Merchant, Shah, Veitch, & Karamanolis 2007; Ports, Clements, Zhang, Madden, & Liskov 2010).

Neste contexto, a replicação dos dados armazenados no sistema e distribuídos pela memória de múltiplos nós possui duas grandes vantagens: por um lado, permite distribuir a carga das operações de leitura por vários nós; por outro, assegura a sobrevivência dos dados na eventualidade da falha de um dos nós. Este último aspecto é de particular relevância, atendendo ao facto de os dados serem primeiro armazenados em memória volátil (cujo conteúdo se perde no caso de falha) e tornados persistentes de forma assíncrona.

As vantagens acima referidas necessitam de ser pesadas contra os custos envolvidos na replicação. Por um lado, os dados consomem memória, reduzindo a quantidade de informação que pode ser armazenada no sistema, podendo assim obrigar a um acesso mais frequente ao disco; por outro, quanto maior o número de nós, maior poderá ser o custo de manter os mesmos coerentes.

Desta forma, justifica-se a utilização de *replicação parcial*, isto é, uma configuração em que cada item é replicado num subconjunto dos nós do sistema, sendo que nenhum nó possui todos os dados. Num sistema com estas características, uma transacção envolve necessariamente vários nós, mais precisamente, os nós que possuem uma cópia dos dados acedidos pela transacção. Como transacções diferentes podem aceder a dados diferentes, os conjuntos de nós envolvidas em cada transacção não são necessariamente os mesmos, embora possam também não ser disjuntos. Assegurar o controlo de concorrência e a gestão da replicação neste cenário e fazê-lo de forma eficiente são desafios significativos.

1.2 Contribuições

A dissertação apresenta as seguintes contribuições:

- descreve um conjunto de algoritmos para a concretização de replicação parcial em sistemas de armazenamento em memória distribuída e transaccional;

- identifica as principais vantagens e desvantagens da utilização de replicação parcial neste contexto, através de uma avaliação experimental.

1.3 Resultados

Os resultados produzidos por esta tese podem ser enumerados da seguinte forma:

- Um protótipo construído com base numa adaptação do *Infinispan*, uma cache de memória transaccional distribuída e replicada de código aberto da RedHat.
- A concretização dos algoritmos propostos no protótipo acima, incluindo dois algoritmos de certificação diferentes para replicação parcial de dados e um terceiro para replicação total, todos baseadas no uso de uma primitiva de difusão atómica. Este último tem como objectivo mostrar a vantagem do uso de primitivas de difusão atómica, especialmente em cenário de elevada contenção.
- Uma avaliação experimental da implementação dos algoritmos com recurso a dois *benchmarks*, um deles sintético e outro mais complexo.

1.4 Evolução do Trabalho

Este trabalho começou com o objectivo de estudar os mecanismos de replicação parcial para sistemas de memória transaccional distribuída por software, estando ainda em aberto qual a plataforma para a qual os mecanismos seriam desenvolvidos. Com a evolução não só deste trabalho, mas também dos projectos em que o grupo de investigação se encontra inserido, considerou-se interessante centrar o trabalho em sistemas de armazenamento em memória do tipo *"grelha-de-dados"* (*data-grid*), e em particular no sistema *Infinispan* da RedHat.

Este trabalho foi parcialmente financiado pela comissão europeia através do projecto "Cloud-TM" (257784), pela FCT através do projecto ARISTOS (PTDC/EIA- EIA/102496/2008), e pelo financiamento plurianual do INESC-ID através de fundos do programa PIDDAC. Durante o meu trabalho, beneficiei da colaboração de vários membros do Grupo de Sistemas Distribuídos do INESC-ID, nomeadamente; Paolo Romano, Nuno Carvalho, Maria Couceiro, Sebastiano Peluso, e Diego Didona.

Parte do trabalho apresentado nesta dissertação foi publicado em (Ruivo, Couceiro, Romano, & Rodrigues 2011a) e (Ruivo, Couceiro, Romano, & Rodrigues 2011b)

1.5 Estrutura do Documento

O remanescente deste relatório está organizado da seguinte forma. O Capítulo 2 expõe o trabalho efectuado nas diversas áreas relacionados com esta tese. O Capítulo 3 introduz o *Infinispan*, uma cache de memória transaccional distribuída e replicada, desenvolvida pela RedHat em código aberto, e descreve com mais detalhe os algoritmos concretizados. O Capítulo 4 apresenta os resultados experimentais das técnicas em estudo. Finalmente, o Capítulo 5 conclui este documento e apresenta os pontos principais e o trabalho futuro.

2 Trabalho Relacionado

2.1 Introdução

Existem três grandes classes de sistemas relevantes para o trabalho descrito nesta dissertação, sobre os quais o presente capítulo oferece uma panorâmica:

- Sistemas de memória transaccional por software, inicialmente desenvolvidos para resolver os problemas do controlo de concorrência em sistemas não distribuídos, mas que tem vindo a ser enriquecidos com suporte para distribuição e replicação;
- Sistemas de armazenamento em memória, inicialmente desenvolvidos como forma de aumentar o desempenho de cargas maioritariamente de leitura, mas que tem vindo a ser enriquecidos com suporte para transacções de leitura e escrita, assim como para replicação de dados;
- Sistemas de suporte à replicação de dados, mesmo em ambientes mais clássicos, como os sistemas de gestão de bases de dados.

Cada uma destas classes é abordada separadamente nas secções seguintes, concluindo-se o capítulo com uma breve discussão dos trabalhos apresentados.

2.2 Memória Transaccional por Software

Um dos principais problemas inerentes à partilha de dados é o controlo de concorrência. Uma zona do código que acede a dados partilhados, e que deve ser executada em exclusão mútua, é tipicamente designada por secção crítica. A execução concorrente do código de uma secção crítica por dois ou mais processos ou fios de execução pode deixar os dados num estado incoerente. Os

mecanismos de controlo de concorrência tem como objectivo resolver este problema e podem ser categorizados como explícitos ou implícitos.

A forma mais básica para proteger uma secção crítica consiste na utilização de *trincos*. Os trincos funcionam da seguinte forma: a cada secção crítica é associado um trinco. Um processo, antes de entrar numa secção crítica, adquire o trinco dessa secção, ou fica bloqueado à espera da sua libertação caso outro processo já possua o trinco. Após terminar o acesso à secção crítica o processo liberta o trinco, libertando assim um dos processos em espera (caso exista). De modo a tornar este mecanismo eficiente, é importante identificar com exactidão qual a secção crítica, evitando executar desnecessariamente código em exclusão mútua, o que limita o desempenho do sistema.

Infelizmente, os trincos são bastante difíceis de utilizar pelos programadores. Uma incorrecta identificação das regiões críticas pode deixar dados partilhados serem acedidos concorrentemente sem a sincronização adequada. A existência de múltiplas regiões críticas no programa leva à geração de código de elevada complexidade onde os trincos são adquiridos e libertados em diferentes pontos, podendo originar situações de interbloqueios (*deadlock*) ou míngua (*starvation*). Outros mecanismos de sincronização explícita, como semáforos, monitores, etc., apesar de serem mais potentes, sofrem dos mesmos problemas.

Uma alternativa para evitar estes problemas consiste em utilizar um mecanismo de concorrência implícito, tipicamente oferecido por sistemas baseados em *transacções*, tal como se descreve nos parágrafos seguintes.

2.2.1 Memória Transaccional

Originalmente proposta no contexto das aplicações de bases de dados, uma transacção é definida como uma sequência de operações de leitura e escrita em dados partilhados, que deve ser executada de forma isolada, sem entrelaçamento com outras transacções que acedam aos mesmos dados. Para além disso, todas as operações de uma transacção são aplicadas de forma atómica, isto é, ou todas as actualizações são aplicadas ou não é nenhuma (neste último caso, diz-se que a transacção foi cancelada). Note-se que, para assegurar o isolamento, o sistema de suporte à execução das transacções pode recorrer ao uso de trincos; no entanto, estes são geridos automaticamente, sem a intervenção directa do programador (Zhang, Chen, Tian, &

Zheng 2008).

Quando a transacção termina, é executado um procedimento com o objectivo de realizar a confirmação (*commit*) da transacção. Caso não se verifiquem erros ou violações do modelo de coerência dos dados devido ao acesso concorrente aos mesmos, a transacção é confirmada e todas as escritas realizadas no seu contexto se tornam visíveis. Caso contrário, a transacção é cancelada e todas as actualizações aos dados são descartadas.

As transacções são usadas de forma generalizada no contexto dos sistemas de gestão de bases de dados (SGBD), onde asseguram as garantias ACID que, de uma forma informal, podem ser descritas da seguinte forma:

- *Atomicidade:* garante que todas ou nenhumas operações são executadas;
- *Coerência:* a execução das transacções não viola a coerência dos dados;
- *Isolamento*: a execução de uma transacção não é afectada pela execução concorrente de outras transacções;
- *Durabilidade:* as modificações, depois de confirmadas, tornam-se persistentes.

Os sistemas de memória transaccional são inspirados nos SGBDs embora com algumas características específicas. Em muitos casos, a persistência dos dados não é um requisito. As transacções podem ser bastante curtas e o modelo de coerência de dados pode ser mais exigente, uma vez que mesmo as transacções que cancelam devem ser protegidas de observar dados incoerentes (uma propriedade designada por opacidade (Guerraoui & Kapalka 2008)).

De forma a realizar o controlo de concorrência, é geralmente necessário recolher informação acerca de quais os dados acedidos por uma transacção (designado por conjunto de dados ou *dataset*). O *dataset* está dividido no conjunto de dados que são lidos (conjunto de leitura ou *readset* – RS) e no conjunto de dados que são actualizados (conjunto de escrita ou *writeset* – WS). Estes dois conjuntos não são geralmente conhecidos à partida, caso em que as transacções se designam por dinâmicas. Quando o *dataset* é conhecido à partida as transacções designam-se por estáticas.

O controlo de concorrência pode seguir uma abordagem optimista ou pessimista. Na abordagem optimista (Bernstein, Hadzilacos, & Goodman 1986), a gestão de conflitos é adiada o

mais possível. Quando começa uma transacção, assume-se que a transacção poderá ser confirmada e que não existirão conflitos com outras transacções. Nos acessos à memória, a transacção não adquire os respectivos trincos e as escritas e leituras são colocadas num histórico local à transacção (*log*). Quando a transacção termina, verifica-se se os dados lidos pela transacção não foram entretanto escritos por outras transacções concorrentes. Caso os dados estejam no mesmo estado, isto significa que não ocorreu nenhum acesso concorrente e a transacção pode ser confirmada, actualizando-se o conjunto de escrita de forma atómica. Caso tenham ocorrido conflitos, a transacção é cancelada, perdendo-se o trabalho realizado durante a sua execução. Esta é a principal desvantagem desta abordagem.

Na abordagem pessimista (Shriraman, Dwarkadas, & Scott 2008), a gestão dos conflitos é efectuada o mais cedo possível. Associa-se um trinco a cada objecto de memória e quando a transacção faz um acesso à memória, o trinco correspondente é adquirido. Os conflitos são detectados mais cedo e o trabalho perdido é menor. No entanto, um modelo de execução pessimista pode originar interbloqueios entre as transacções. Assim, o sistema deve prevenir ou detectar (e resolver) essas situações. Em transacções estáticas é possível prevenir a ocorrência de interbloqueios se os trincos forem adquiridos por uma ordem determinista. No entanto, em transacções dinâmicas, é necessário executar algoritmos de detecção ou prevenção de interbloqueio. Para mais detalhes sobre a prevenção e detecção de interbloqueios, sugere-se a consulta de Ramakrishnan & Gehrke (2000), páginas 546 a 548.

A escolha entre a abordagem optimista e pessimista depende do tipo de sistema. Na abordagem optimista, tem-se o custo de fazer um única validação no final da transacção, enquanto que na abordagem pessimista tem-se o custo de adquirir/libertar os trincos juntamente com o de prevenir/detectar os interbloqueios. Desta forma, em sistemas com poucos conflitos, uma abordagem optimista é mais vantajosa.

Os mecanismos de concorrência (Shriraman, Dwarkadas, & Scott 2008) podem ser concretizados em *hardware*, (Ananian, Asanovic, Kuszmaul, Leiserson, & Lie 2006; Hammond, Wong, Chen, Carlstrom, Davis, Hertzberg, Prabhu, Wijaya, Kozyrakis, & Olukotun 2004; Herlihy & Moss 1993; Moore, Bobba, Moravan, Hill, & Wood 2006; Rajwar, Herlihy, & Lai 2005) *software*, (Dice, Shalev, & Shavit 2006; Fraser & Harris 2007; Herlihy, Luchangco, & Moir 2003; Marathe, Iii, & Scott 2005; Saha, Adl-Tabatabai, Hudson, Minh, & Hertzberg 2006) ou de uma forma híbrida (Damron, Fedorova, Lev, Luchangco, Moir, & Nussbaum 2006; Kumar, Chu,

Hughes, Kundu, & Nguyen 2006; Minh, Trautmann, Chung, McDonald, Bronson, Casper, Kozyrakis, & Olukotun 2007; Shriraman, Spear, Hossain, Marathe, Dwarkadas, & Scott 2007). Os mecanismos baseados em *hardware* (HTM) são mais rápidos, mas não são flexíveis e possuem escolhas fixas ou limitadas de concretização. O oposto acontece com os mecanismos por *software* (STM), que são mais lentos, mas oferecem uma maior flexibilidade. Neste documento, vamos focar-nos nos mecanismos concretizados em *software*.

2.2.2 Sistemas de Memória Transaccional por Software

Como supracitado, um sistema de memória transaccional pode ser concretizado em *software*. Um exemplo de um sistema STM é apresentado por Shavit & Touitou (1995). Esta solução oferece uma concretização não bloqueante de transacções estáticas. O modo de funcionamento baseia-se na declaração global dos dados que a transacção acede. Uma transacção que declara um conjunto de dados é denominada de *dono* desses dados e, para garantir actualizações atómicas necessita de garantir donos exclusivos para cada item de dados. A principal desvantagem é ter de possuir conhecimento *à-priori* dos dados acedidos pela transacção.

No entanto, um trabalho recente efectuado por Herlihy, Luchangco, & Moir (2003) têm-se focado em fornecer acessos concorrentes a dados dinâmicos, isto é, não são conhecidos *à-priori*, pois muitas das aplicação alocam e utilizam dados dinamicamente.

Para oferecer um sistema com elevado desempenho, o número de transacções canceladas deve ser inferior ao número total de transacções do sistema. Desse modo, conflitos entre transacções devem ser minimizados para reduzir o número de transacções canceladas. Um mecanismo de controlo de concorrência baseado em múltiplas versões foi proposto por Cachopo & Rito-Silva (2006). Para cada item do conjunto de dados, o sistema mantém um certo número de versões. Uma versão é associada à transacção quando se inicia, e o valor retornado pela leitura é baseado nessa versão. Desse modo, o mecanismo de controlo de concorrência garante que a transacção observa um estado coerente e, tem como consequência, transacções só de leitura que nunca são canceladas.

2.2.3 Memória Transaccional por Software Distribuída

Os primeiros sistemas STM foram desenvolvidos para máquinas multi-processador (Cachopo & Rito-Silva 2006; Dice, Shalev, & Shavit 2006). No entanto, quando usados em sistemas que são submetidos a cargas elevadas, deveria ser possível aumentar a capacidade de processamento, acrescentando múltiplos nós recorrendo a um operação distribuída da STM. Neste contexto, definimos capacidade de escala (Ye, Kerherve, v Bochmann, & Bourne 2002) como a capacidade de um sistema se adaptar ao aumento do número de nós. Como as STM e os SGBD partilham o conceito de transacção, as STM começaram a adoptar soluções existentes para SGBD (Romano, Carvalho, & Rodrigues 2008).

A distribuição consiste em dividir os dados pelos diversos nós do sistema sem recorrer à replicação. Neste caso, aumenta-se a quantidade de dados que é possível armazenar e divide-se a carga pelos nós do sistema. Normalmente, uma transacção não necessita de aceder a todos os dados. Assim, não é necessário que seja certificada por todos os nós. Isto permite que o sistema seja mais escalável mas, se um dos nós falhar, pode levar à perda dos dados armazenados nesse nó.

A maioria dos protocolos para suportar distribuição de dados foca-se em arquitecturas de memória partilhada, deixando o domínio dos *clusters* inexplorado. O comportamento destes protocolos é diferente em *clusters*, devido ao custo da troca de uma mensagem entre nós (Kotselidis, Ansari, Jarvis, Luján, Kirkham, & Watson 2008). Em *clusters* existem dois objectivos importantes: (i) reduzir (ou eliminar) as operações remotas (comunicação); (ii) assegurar que a concretização da STM permite agregar mensagens e explorar o acesso localizado.

Um exemplo de uma STM distribuída é o DiSTM (Kotselidis, Ansari, Jarvis, Luján, Kirkham, & Watson 2008). No DiSTM são apresentados três protocolos para manter a coerência dos dados. Todos assumem a existência de um nó especial — o mestre — com quem todos comunicam e que possui um comportamento diferente dos restantes. Esses protocolos são o *Transactional Coherence and Consistency*, TCC (Hammond, Wong, Chen, Carlstrom, Davis, Hertzberg, Prabhu, Wijaya, Kozyrakis, & Olukotun 2004), e dois protocolos baseados no conceito de *leases* (Gray & Cheriton 1989).

No TCC cada transacção adquire uma *senha* (*ticket*), mecanismo através do qual se define uma ordem global das transacções. A validação é efectuada no final da transacção, recorrendo à

distribuição do WS e do RS. Cada nó vai comparar o WS e o RS das transacções concorrentes e, caso seja detectado algum conflito, a decisão de cancelar depende da ordem de seriação, isto é, a transacção com a *senha* mais recente deve ser cancelada. A coerência é garantida através de uma aproximação pessimista no mestre. No caso de a transacção ser confirmada, as actualizações são tornadas públicas, quer localmente, quer globalmente no mestre, que possui todos os dados. Cada nó pode ter uma cópia (cache) de dados de outros nós. O mestre invalida estas cópias, cada vez que os dados são alterados.

Nos outros dois protocolos, cada transacção adquire um *lease*. O papel do *lease* é seriar as confirmações das transacções, evitando o custo de distribuir o *dataset*. Quando uma transacção possui o *lease*, e não foi cancelada, pode confirmar as suas modificações, pois é garantido que mais ninguém vai confirmar ou certificar outras transacções. As modificações são tornadas públicas no mestre, o que pode provocar o cancelamento de outras transacções. A diferença entre os dois protocolos baseados em *leases* é que um deles possui um único *lease*, enquanto o outro possui múltiplos *leases*. O uso de múltiplos *leases* tem a vantagem de confirmar várias transacções em simultâneo, mas com o custo de uma validação extra, efectuada no mestre, onde é verificado que não existem conflitos com transacções que possuem outros *leases*.

2.2.4 Memória Transaccional por Software vs SGBDs

Os Sistemas de Gestão de Base de Dados (SGBD) são conjuntos de programas que permitem armazenar, modificar e extrair grandes quantidades de dados a partir de uma base de dados. Uma base de dados é um conjunto estruturado de registos e a forma como estes estão estruturados é denominado o modelo da base de dados. Uma transacção é a unidade de trabalho efectuada no SGBD.

O sistema STM e o SGBD possuem várias características em comum, nomeadamente ambos suportam a abstracção de transacção e ambos controlam de forma automática o acesso concorrente aos dados. No entanto, existem também diferenças significativas entre estes dois tipos de sistemas, nomeadamente:

- o sistema STM não possui o requisito de assegurar a persistência dos dados;
- as transacções num sistema STM podem conter código arbitrário, que não está restrito pela sintaxe da linguagem usada para aceder a bases de dados, o SQL;

- o número de instruções de uma transacção em sistemas STM pode ser significativamente menor do que o número de instruções de uma transacção num SGBD;
- as transacções num sistema STM não são executadas num ambiente confinado, pelo que a leitura de valores incoerentes pode gerar desvios na sua execução tais como: a geração de excepções de hardware ou ciclos infinitos, que podem tornar difícil ou mesmo impossível cancelar as mesmas. Por esta razão, os modelos de coerência para sistemas STM são, por vezes, mais restritos que os modelos de coerência para SGBD. Enquanto em SGBD, o modelo de coerência mais forte é a serializabilidade (Bernstein, Hadzilacos, & Goodman 1986), em sistemas STM tenta-se também assegurar a *opacidade* (Guerraoui & Kapalka 2008), que impede qualquer transacção de ver resultados incoerentes (mesmo as transacções que estão condenadas a cancelar).

2.3 Armazenamento de Dados em Memória

Outro modo de melhorar o desempenho dos SGBDs é manter os dados em memória para evitar a penosas leituras e escritas em disco. As caches surgiram como ferramenta de armazenamento de dados em memória com o objectivo de aumentar o desempenho de sistemas que necessitam de aceder a grandes quantidades de informação, mas com uma latência inferior aos SGBD. As caches funcionam em conjunto com SGBD ou com qualquer outro tipo de armazenamento persistente. O aumento do desempenho é obtido através de uma baixa frequência ao acesso ao armazenamento persistente. Inicialmente, só as leituras evitavam o acesso ao disco, sendo as escritas efectuadas em disco imediatamente, no entanto, os sistemas de cache evoluíram de modo a suportar escritas em memória e efectuar mais tarde as escritas em disco, de um modo totalmente assíncrono. Outra vantagem dos sistemas de cache é possuírem uma interface mais simples de acesso aos dados que as bases de dados, normalmente suportando apenas associações chave/valor. Deste modo, elimina-se o custo de processar e optimizar o SQL dos SGBD.

2.3.1 Caches Distribuídas

À medida que as caches são usadas numa maior diversidade de contextos, os mecanismos de distribuição e replicação dos dados tornam-se um aspecto central, como forma de distribuir a carga pelos vários nós do sistema e de aumentar a fiabilidade do sistema.

O *Cassandra* (Lakshman & Malik 2010) e o *Memcached* (Fitzpatrick 2004) são dois exemplos desta tecnologia, sendo usados em aplicações bem conhecidas, tais como o *Facebook* e a *Wikipedia*, respectivamente. Os dados são guardados em associações chave/valor. A chave do mapa corresponde à chave primária de uma linha da tabela e o valor é o valor das colunas dessa linha. Uma leitura é efectuada em memória se o seu valor existir. Caso não exista, é lido do SGBD e guardado num dos nós. Por outro lado, uma escrita é primeiro efectuada no SGBD antes de ser copiada para memória. A distribuição dos dados é efectuada com base numa função de dispersão coerente.

O *Dynamo* (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007) foi desenvolvido pela *Amazon* com o objectivo de optimizar a transferência de informação entre os vários serviços disponibilizados. Como foi desenvolvido especialmente para este objectivo, o *Dynamo* suporta replicação dos dados (torna os dados tolerante a faltas) e escritas em memória (que são efectuadas em disco de uma forma assíncrona). O uso destes dois mecanismos pode originar conflitos durante as escritas. Para resolver esses conflitos e por uma questão de desempenho, o modelo de coerência suportado é *coerência eventual* (*eventually consistent*) (Vogels 2008). Neste sistema, para que uma operação de leitura ou escrita tenha sucesso, esta deve ser confirmada por uma maioria dos nós. A detecção de conflitos gerados por escritas concorrentes é efectuada e resolvida durante as leituras. A resolução é efectuada através de unificação das escritas concorrentes num novo valor, ou através da escolha da escrita mais recente. Os autores afirmam que este é o modelo que melhor satisfaz os requisitos dos seus serviços.

2.3.2 Caches Transaccionais

Os primeiros sistemas de caches não consideravam suporte para transacções (Lakshman & Malik 2010; DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, & Vogels 2007; Fitzpatrick 2004). Desse modo, não preservavam a coerência transaccional oferecida pela base de dados, violando o isolamento assegurado pelo SGBDs, nomeadamente a serializabilidade.

Aproximações mais recentes já suportam na sua arquitectura a execução de transacções, tais como o sistema *Sinfonia* (Aguilera, Merchant, Shah, Veitch, & Karamanolis 2007) e o *TxCache* (Ports, Clements, Zhang, Madden, & Liskov 2010). O *Sinfonia* suporta transacções

e replicação. As transacção são estáticas e são constituídas por três conjuntos de itens: (i) verificação (actua como pré-requisito para o sucesso da transacção), (ii) leitura e (iii) escrita. Os valores do conjunto de itens para verificação e para escrita são conhecido *à-priori*. As transacções são confirmadas através de um protocolo de confirmação em duas fases (descrito mais adiante neste capítulo) que, por uma questão de desempenho, é efectuado ao mesmo tempo que a execução. Na primeira fase do protocolo de confirmação, o coordenador (cliente) envia a transacção para todas as réplicas. Cada réplica, ao receber a transacção, tenta adquirir os trincos, faz a verificação e, se nenhuma das operações anterior falhou, faz as leituras. A aquisição de um trincos falha se este não estiver disponível. Cada réplica informa o coordenador do sucesso ou insucesso dessas operações e das respectivas leituras. Se o coordenador receber alguma resposta negativa, a transacção é cancelada. Caso contrário é confirmada e, as escritas são aplicadas e os trincos libertados.

O sistema *TxCache* assegura que os utilizadores da cache observam um corte coerente dos dados através de um mecanismo de controlo de concorrência baseado em versões. O SGBD deve suportar este mecanismo, de modo a garantir a coerência dos dados nas operações de leitura. Este requisito pode restringir os SGBD possíveis de serem utilizados ou a sua alteração.

2.3.3 Grelhas-de-Dados em Memória

As grelhas-de-dados em memória (do Inglês, *data-grids)* resultam da combinação de funcionalidades típicas das caches distribuídas, com as funcionalidades oferecidas pelos sistemas de memória transaccional por software. Estes sistemas podem ser usados isoladamente, sem qualquer tipo de armazenamento persistente auxiliar. Podem também ser usados em combinação com suportes persistentes, realizando as transferências dos dados para disco de forma assíncrona. Isto leva a que as transacções sejam totalmente executadas em memória, característica de um sistema STM. No entanto, as escritas podem ser efectuadas logo de imediato, se assim for o desejado.

Alguns exemplos de grelhas-de-dados são o *EhCache*[1], *Oracle Coherence*[2] e o *Infinispan*. Estes sistemas são flexíveis e permitem a sua configuração consoante a aplicação a utilizar. Todos eles oferecem um modo de funcionamento replicado, em que cada nó do sistema possui

[1]EhCache: `http://ehcache.org/documentation/overview.html`
[2]Oracle Coherence: `http://coherence.oracle.com`

uma cópia de todos os dados, e distribuído, em que só existe uma cópia num dos nós do sistema. Alguns destes sistemas, como o *Oracle Coherence* e o *Infinispan*, fornecem também suporte para replicação parcial. Um aspecto relevante nestes sistemas é o suporte para replicação dos dados, de modo a oferecer tolerância a faltas, pois os dados encontram-se em memória volátil e só são armazenados persistentemente de um modo diferido (se necessário). Tal como as caches distribuídas, as grelhas-de-dados disponibilizam tipicamente uma interface baseada na associação de chave/valor, e oferecem um nível de coerência mais fraco que serializabilidade.

Uma vez que o armazenamento persistente é frequentemente o principal ponto de estrangulamento no desempenho de aplicações empresariais, muitas aplicações começaram a adoptar as grelhas-de-dados como parte integrante dos seus sistemas de armazenamento de dados. As grelhas-de-dados oferecem um elevado desempenho, disponibilidade e capacidade de escala, requisitos de um elevado número de aplicações empresariais reais. Neste trabalho foi usado como protótipo o *Infinispan*, que será descrito em mais detalhe no capítulo seguinte.

2.4 Replicação de Dados

A distribuição permite que um sistema possua capacidade de escala, uma vez que a execução das transacções pode ser distribuída pelos nós. No entanto, com a falha de um dos nós pode ocorrer perda dos dados. Em alguns sistemas, a sobrevivência dos dados é um factor importante. A replicação é uma técnica que consiste na criação e manutenção de várias cópias dos dados em múltiplos nós, com o objectivo de aumentar a disponibilidade, tolerância a faltas e fiabilidade de um sistema (Osrael 2007). Além disso, permite aumentar o paralelismo do sistema, embora acarrete custos de sincronização entre nós, para garantir a coerência dos dados.

Nesta secção vamos descrever algoritmos para garantir um nível de coerência forte em ambientes com dados replicados em múltiplos nós. Vamos começar por indicar dois mecanismos de suporte transversais a todos os algoritmos e de seguida explicamos esses algoritmos. No final, terminamos com alguns exemplos de sistemas baseados nesses algoritmos.

2.4.1 Mecanismos de Suporte

2.4.1.1 Confirmação em Duas Fases

A confirmação atómica em duas fases (do Inglês, *2-Phase Atomic Commitment, 2PC*) (Atif 2009), é o algoritmo mais simples e mais usado para coordenar a confirmação ou cancelamento de uma transacção distribuída. Neste algoritmo, o nó que submete a transacção actua como coordenador e os restantes como participantes. Como o seu nome sugere, o algoritmo tem duas fases. Na primeira fase, o coordenador dá a conhecer a transacção. Ao receber a transacção, cada participante prepara-a para ser confirmada e envia de volta ao coordenador o seu voto, indicando se é ou não possível confirmar localmente a transacção. A primeira fase termina com a recepção dos votos de todos os participantes pelo coordenador. Na segunda fase, todos completam a transacção, após recepção de uma mensagem de confirmação ou de cancelamento enviada pelo coordenador. O coordenador envia uma mensagem de confirmação se receber todos os votos positivos, caso contrário, envia uma mensagem de cancelamento.

O 2PC possui a desvantagem de bloquear caso o coordenador falhe e os participantes estejam no estado não decidido, isto é, ainda não tenham recebido a resposta do coordenador. Neste estado, todos os participantes mantêm a transacção preparada, e possivelmente os dados bloqueados, até receberem a resposta final do coordenador, o que pode impedir a confirmação de novas transacções. O *Three-Phase Commit*, 3PC (Atif 2009) adiciona mais uma fase ao protocolo, de modo a evitar a ocorrência da situação acima definida.

2.4.1.2 Comunicação em Grupo

Os Sistemas de Comunicação em Grupo (SCG) oferecem comunicação fiável ponto-a-multiponto. Estes sistemas asseguram que todos os membros do grupo recebem uma cópia das mensagens enviadas para o grupo, com diferentes garantias de ordenação. Estas garantias asseguram que todos os membros do grupo recebem o mesmo conjunto de mensagens e todos concordam com a ordem de entrega das mensagens à aplicação. O SCG é um paradigma poderoso que tem sido amplamente utilizado na gestão de dados replicados, quer no contexto de sistemas de STM quer em SGBD, simplificando os algoritmos necessários para manter coerência das várias réplicas. Nos parágrafos seguintes, pretende-se descrever, de um modo geral, as garantias e as propriedades oferecidas pelos SCGs.

2.4.1.2.1 Difusão Atómica A Difusão Atómica (do Inglês *Atomic Broadcast*, ABcast), também conhecida por *Total Order Broadcast* (Hadzilacos & Toueg 1993), é uma primitiva de comunicação em grupo que permite o envio de mensagens para todos os processos, com a garantia que todos concordam no conjunto de mensagens entregues e na ordem pela qual são entregues. ABcast pode ser definido através das primitivas `TO-broadcast(m)` e `TO-deliver(m)`, onde `m` é uma mensagem, e possui as seguintes propriedades:

- *Ordem:* Se um processo entrega a mensagem A antes da mensagem B, então todos os processos entregam a mensagem A antes da mensagem B;
- *Terminação:* Se um processo correcto envia uma mensagem, então todos os processos correctos, alguma vez, entregam a mensagem;
- *Atomicidade:* Se um processo entrega uma mensagem, então todos os processos correctos, alguma vez, entregam a mensagem.

2.4.1.2.2 Difusão Atómica Optimista O ABcast é importante para construir aplicações distribuídas mas, infelizmente, a sua concretização pode ser muito dispendiosa. Isto deve-se ao número de passos de comunicação e às mensagens trocadas que são necessárias para concretizá-lo, o que pode introduzir uma latência significativa na entrega das mensagens.

A Difusão Atómica Optimista (do Inglês *Optimistic Atomic Broadcast*, OABcast) tem como objectivo minimizar os problemas de latência da primitiva anterior. Neste serviço, um membro do grupo entrega à aplicação uma mensagem assim que a recebe (entrega espontânea). Deste modo, a aplicação pode começar por processar a mensagem, em paralelo com a coordenação necessária para determinar a ordem total. No final da fase de coordenação, a mensagem (ou um identificador da mensagem) é entregue na sua ordem final (entrega final), que corresponde à ordem total. O OABcast possui portanto uma primitiva adicional, `Opt-deliver(m)`, que estima uma ordem final em cada processo (ordem espontânea), sendo caracterizado pelas seguintes propriedades (Palmieri, Quaglia, & Romano 2010; Kemme, Pedone, Alonso, Schiper, & Wiesmann 2003):

- *Terminação:* Se um processo correcto envia uma mensagem m, então, alguma vez, entrega m na ordem espontânea;

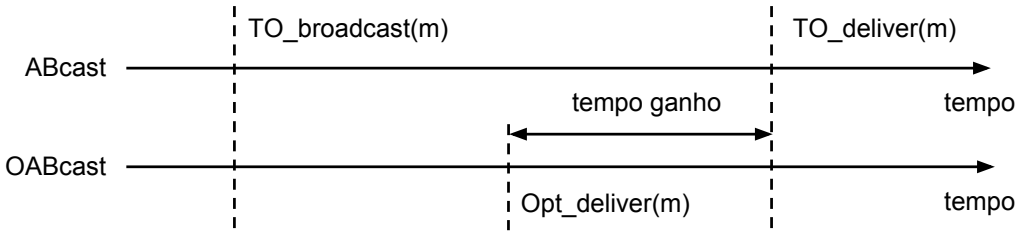


Figura 2.1: Diferença entre Difusão Atómica (ABcast) e Difusão Atómica Optimista (OABcast)

- *Acordo global:* Se um processo correcto entrega uma mensagem m na ordem espontânea, então todos os processos correctos, alguma vez, entregam m na ordem espontânea;
- *Acordo local:* Se um processo correcto entrega uma mensagem m na ordem espontânea, então entrega m na ordem final;
- *Ordem global:* Se dois processos entregam m e m' na ordem final, então fazem-no pela mesma ordem;
- *Ordem local:* Se um processo entrega na ordem final uma mensagem m, então entregou m na ordem espontânea, anteriormente.

A Figura 2.1 mostra a vantagem do OABcast em relação ao ABcast. Se a ordem espontânea for igual à ordem final, então consegue-se ganhar tempo para processamento, indicado por *tempo ganho* na figura. Verifica-se que a probabilidade da ordem final ser igual à ordem espontânea, é muito elevada quando todos os nós se encontram ligados entre si, no mesmo segmento de uma rede local (Rodrigues, Mocito, & Carvalho 2006).

2.4.1.2.3 Difusão Fiável ABcast e OABcast acima definidos garantem a ordem total na entrega de mensagens. No entanto, existem situações onde só é necessário garantir que uma cópia da mensagem seja recebida por todos os processos. A Difusão Fiável (do Inglês *Reliable Broadcast*, RBcast) é uma primitiva semelhante ao ABcast, mas que não garante a ordem total da entrega de mensagens (Schiper, Schmidt, & Pedone 2006). O RBcast é definido pelas primitivas `R-broadcast(m)` e `R-deliver(m)` e é caracterizado pelas seguintes propriedades:

- *Validade:* Se um processo correcto envia uma mensagem m, então, alguma vez, entrega m;

- *Acordo:* Se um processo correcto entrega uma mensagem m, então, alguma vez, todos os processos correctos entregam a mensagem m;

- *Integridade:* Para cada mensagem m, cada processo entrega m uma única vez, se esta foi previamente enviada.

2.4.1.2.4 Difusão Atómica Selectiva A Difusão Atómica Selectiva (do Inglês *Atomic Multicast*, AMcast) é uma variação do ABcast. Este serviço permite o envio de mensagens para um subconjunto dos processos, enquanto o ABcast envia para todos os processos. AMcast também garante que todos os processos concordam na ordem de entrega das mensagens (Schiper & Pedone 2008; Guerraoui & Schiper 2001; Schiper, Sutra, & Pedone 2010). É definido pelas primitivas `A-multicast(m)` e `A-deliver(m)` e possui as seguintes propriedades (Schiper, Sutra, & Pedone 2010):

- *Integridade:* Para qualquer processo s e qualquer mensagem m, s entrega m no máximo uma vez, se e só se $s \in m.dst$[3] e m foi previamente enviada;

- *Validade:* Se um processo correcto s envia uma mensagem m, então, alguma vez, todos os processo correctos $s' \in m.dst$ entregam m;

- *Acordo global:* Se um processo s entrega uma mensagem m, então, alguma vez, todos os processos correctos $s' \in m.dst$ entregam m;

- *Ordem prefixa uniforme:* Para qualquer duas mensagens m e m', e para quaisquer dois processos s e s' tais que $\{s, s'\} \subseteq m.dst \cap m'.dst$, se s entrega m e s' entrega m', então s entrega m' antes de m ou s' entrega m antes de m'

2.4.2 Replicação Total

Apresentam-se agora alguns dos principais algoritmos para concretizar a replicação total, isto é, para assegurar a coerência dos nós em sistemas em que todos nós (daqui em diante referido como réplica) mantêm uma cópia dos dados.

[3] $m.dst$ indica o conjunto de réplicas a quem a mensagem m se destina

2.4.2.1 Primário-Secundário

Neste algoritmo, também conhecido por replicação passiva (Osrael 2007), só uma das réplicas — o nó primário — processa os pedidos de escrita dos clientes e propaga as actualizações para as outras réplicas — os nós secundários. As actualizações podem ser propagadas de modo síncrono ou assíncrono. No primeiro caso, o primário fica bloqueado até obter uma confirmação dos secundários, indicando que as actualizações foram aplicadas. Os secundários mantêm os dados coerentes e as operações de leitura nos secundários devolvem o valor mais recente. No modo assíncrono, as actualizações são adiadas. Neste caso, as operações de leitura nos secundários podem devolver um valor antigo.

A principal vantagem desta técnica é a sua simplicidade e o facto de envolver menos processos redundantes (Zou & Jahanian 1998). Além disso, todas as operações de escritas são dirigidas ao primário, o que torna fácil a sua ordenação e seriação. Normalmente, a detecção de falha do primário é efectuada através de temporizadores (*timeouts*). Isto origina um problema no caso de o primário ficar sobrecarregado (e por isso, demorar demasiado tempo na resposta a pedidos). O atraso no primário pode ser detectado como uma falha, levando um dos secundários a assumir o seu lugar. Caso aconteça, dois primários podem estar activos originando incoerências nas réplicas.

2.4.2.2 Trincos e 2PC

Este algoritmo de gestão da replicação consiste na combinação da aquisição ávida de trincos sobre os dados acedidos com uma fase de coordenação final baseada em 2PC. Na primeira fase, os participantes adquirem os trincos sobre os dados acedidos pela transacção. Se o coordenador receber todas as respostas positivas, então é garantido que nenhuma transacção conflituosa irá ser confirmada em simultâneo. O uso de trincos simplifica bastante o algoritmo, no entanto, interbloqueios podem acontecer. Se em dois participantes, digamos P_1 e P_2, duas transacções conflituosas, digamos T_1 e T_2, são recebidas por ordem diferente, estes irão adquirir os trincos por ordem diferentes. Num dos participantes, T_2 fica bloqueada até T_1 estar concluída e vice-versa no outro. Sem nenhum mecanismo para prevenir, ou detectar e resolver esta situação for utilizado, todo o sistema pode ficar bloqueado.

2.4.2.3 Máquina de Estados Replicada

Esta abordagem envolve todas as réplicas no processamento dos pedidos do cliente. Nesta técnica, também conhecida por replicação activa (Schneider 1990), o cliente envia o seu pedido para todas as réplicas. O pedido é processado e no final as réplicas respondem ao cliente. O cliente pode esperar pela primeira, por uma maioria ou por todas as respostas. A principal vantagem desta técnica é que a falha é totalmente transparente para o cliente (Zou & Jahanian 1998). No entanto, esta técnica necessita de uma primitiva de comunicação que garanta que todas as réplicas recebem os pedidos pela mesma ordem (por exemplo, ABcast), e assim, toda a complexidade fica escondida pela primitiva de comunicação. Outra desvantagem é que esta técnica só funciona para pedidos determinísticos, isto é, dado um estado e um parâmetro de entrada, a transacção tem de chegar ao mesmo resultado. Caso contrário, as réplicas podem ficar numa estado incoerente.

Comparando replicação activa e passiva, podemos concluir o seguinte:

- A replicação activa usa mais recursos computacionais do que a passiva. Isto deve-se ao facto de todas as réplicas processarem os pedidos do cliente;
- Em casos de falha, a replicação passiva tem uma maior latência. O cliente necessita de repetir o pedido para outra réplica;
- Replicação activa necessita de operações determinísticas. Todas as réplicas necessitam de chegar ao mesmo resultado de modo a garantir a coerência dos dados. Isto já não acontece no caso da replicação passiva.

2.4.2.4 Algoritmos com Certificação

Os algoritmos com certificação são baseados no modelo de actualização diferida (*deferred update*) e usam primitivas de comunicação em grupo. De acordo com o modelo de actualização diferida, as transacções são processadas localmente numa única réplica e, em tempo de confirmação, são enviadas para as outras réplicas. Este modelo tem as seguintes vantagens (Gray, Helland, O'Neil, & Shasha 1996):

- Melhor desempenho, através da recolha e propagação de várias modificações juntas, e executar a transacção numa única réplica, possivelmente perto do cliente;

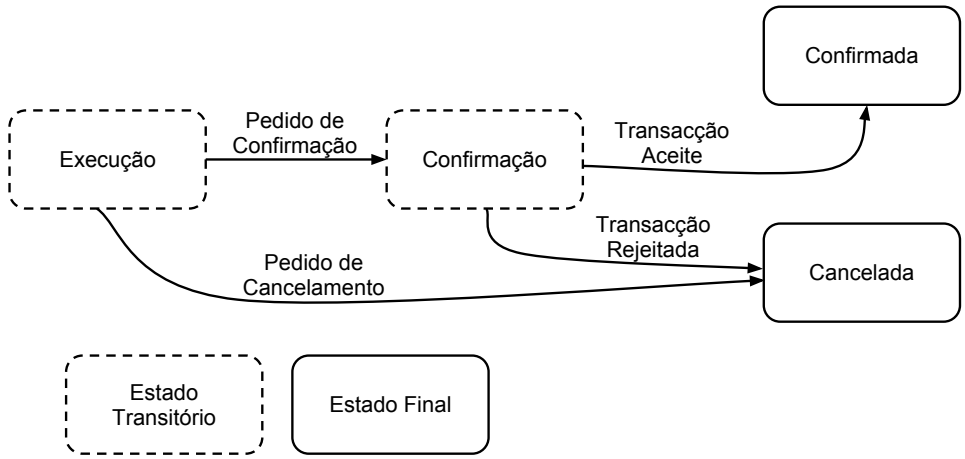


Figura 2.2: Estados de um transacção no modelo de actualização diferida

- Melhor suporte para tolerância a faltas, por simplificar a recuperação das réplicas;
- Baixa taxa de interbloqueios, pela eliminação de interbloqueios distribuídos.

No entanto, este modelo também possui algumas desvantagens, entre as quais, um aumento na taxa de cancelamentos, devido à ausência de sincronização durante a execução da transacção.

Neste modelo, uma transacção passa por estados bem definidos (Pedone, Guerraoui, & Schiper 2003) (ver Figura 2.2):

- *Execução:* A transacção começa neste estado, onde são executadas as operações de leitura e escrita. Quando a transacção pede a confirmação, passa para o estado *Confirmação* e é enviada para as outras réplicas;
- *Confirmação:* Uma transacção recebida encontra-se neste estado e mantém-se nele até o seu resultado ser conhecido. A transacção é certificada quando é recebida;
- *Confirmada/Cancelada:* Estado final da transacção.

2.4.3 Replicação Parcial

A replicação total obriga a propagar os resultados de cada transacção para todas as réplicas do sistema. Isto não é eficiente quando o sistema possui muitas réplicas. O conceito de replicação parcial refere-se a uma variação da replicação total, na qual uma réplica não possui

todos os dados (Alonso 1997). Cada réplica, só guarda um determinado subconjunto de todos os dados da base de dados e, esse conjunto pode ter cópias em várias réplicas. A ideia principal da replicação parcial é que nem todas as réplicas têm de processar uma transacção. A transacção só tem de ser enviada para o conjunto de réplicas que possuem os dados acedidos pela transacção. Assim, a rede fica menos sobrecarregada e o custo da coordenação entre réplicas é menor (Coulon, Pacitti, & Valduriez 2005). Ao contrário da replicação total, a replicação parcial permite aumentar a localidade no acesso aos dados e reduzir o número de mensagens trocadas no sistema (Coulon, Pacitti, & Valduriez 2005). A replicação parcial combina as vantagens da replicação e da distribuição mas, sem as desvantagens de cada uma delas.

No entanto, não é fácil suportar replicação parcial a partir dos protocolos de replicação total. Normalmente, a replicação total é baseada em primitivas de comunicação em grupo, que garantem que todas as réplicas recebem a transacção pela mesma ordem. Se o funcionamento for semelhante a um autómato finito, esta solução funciona bem para replicação total. No entanto, esta solução não funciona em replicação parcial (Alonso 1997). Por exemplo, considere um sistema composto por três réplicas (S_1, S_2 e S_3). A réplica S_1 contém B e C, S_2 contém A e B e S_3 contém A e C. Considere ainda que existem duas transacções t_1 e t_2 a ser executadas concorrentemente em S_1 e S_2, respectivamente. A transacção t_1 modifica B e t_2 modifica A e B. No final, S_1 e S_2 certificam ambas as transacções mas, S_3 só certifica t_2. Se a ordem total for t_1 e t_2, S_1 e S_2 confirmam t_1 e cancelam t_2 mas S_3 confirma t_2, modificando o valor de A, o que torna os dados incoerentes.

Os protocolos de replicação parcial dividem-se em três categorias (Schiper, Sutra, & Pedone 2010; Schiper, Schmidt, & Pedone 2006):

- *Genuínos:* Para cada transacção T, as réplicas que certificam T são aquelas que possuem dados acedidos por T;
- *Quase-genuínos:* Para cada transacção T, réplicas correctas que não possuem nenhuns dados acedidos por T, não armazenam permanentemente mais do que o identificador de T;
- *Não genuínos:* Para cada transacção T, todas as réplicas guardam informação sobre T, mesmo que não não possuam nenhuns dados acedidos por T.

Os protocolos não genuínos vão contra o objectivo da replicação parcial, pois todas as réplicas têm de se envolver na transacção, o que conduz aos problemas da replicação total.

2.4.4 Alguns Exemplos

Algoritmos de Certificação para SGBD Pedone, Guerraoui, & Schiper (1998) apresentam um protocolo baseado no modelo de actualização diferida. Na fase de confirmação, a transacção é difundida para as restantes réplicas e cada uma delas certifica a transacção. A transacção é confirmada se não houver conflitos com outra transacção concorrente. Uma optimização a este modelo é efectuada por Pedone, Guerraoui, & Schiper (2003). Com o objectivo de obter uma menor taxa de cancelamentos e um maior desempenho, cada transacção é reordenada numa fila de modo a gerar menos conflitos. Ao ser reordenada numa posição, tem de se garantir que a transacção não origina conflitos com as outras transacções da fila.

Algoritmos de Certificação para STM Os exemplos acima são protocolos desenvolvidos para SGBD mas, ao contrário das transacções em SGBD, as transacções em STM não possuem custos de acesso ao disco, nem custos de interpretação e optimização de SQL (Romano, Carvalho, & Rodrigues 2008). Como consequência, os custos da comunicação em grupo são mais visíveis e formam o principal ponto de estrangulamento na certificação de transacções. Um exemplo de um sistema STM replicado é o D2STM (Couceiro, Romano, Carvalho, & Rodrigues 2009). Com o objectivo de diminuir os custos da comunicação, D2STM apresenta uma nova técnica de certificação denominada *Bloom Filter Certification*. Esta utiliza uma técnica de codificação recorrendo a *bloom filters* (Bloom 1970) que permite reduzir o tamanho do RS. No entanto, introduz um custo (aumento da taxa de cancelamento) provocado pelos falsos positivos dessa técnica. A taxa de cancelamento é inversamente proporcional ao tamanho final da codificação, o que permite que esta taxa seja controlada. O modelo de execução é semelhante aos protocolos de replicação descritos anteriormente. As transacções só de leitura executam-se localmente e são sempre confirmadas, pois a *Java Versioned Software Transactional Memory* (Cachopo & Rito-Silva 2006) (JVSTM), STM que está na base deste sistema, garante que lêem de um corte válido e coerente. As restantes transacções são certificadas localmente, antes de dar início ao protocolo de certificação. Se passar na certificação local, a transacção é enviada através do ABcast com o seu RS codificado. A validação é feita verificando se os valores lidos não foram

modificados por transacções concorrentes.

Carvalho, Romano, & Rodrigues (2010) apresentam uma nova abordagem baseada em *leases*. O algoritmo é denominado de *Asynchronous Lease Certification*. Os *leases* são usados por uma réplica de modo a conceder-lhe privilégios temporários na gestão de um subconjunto dos dados. Esta abordagem consegue reduzir o tempo da fase de confirmação e protege as transacções de serem repetidamente canceladas devido a conflitos remotos. Tal como na D2STM (Couceiro, Romano, Carvalho, & Rodrigues 2009) é usada a JVSTM (Cachopo & Rito-Silva 2006) que garante que as transacções de leitura nunca são canceladas. De forma análoga aos sistemas anteriores, as transacções são executadas localmente e no final dá-se início ao algoritmo de coordenação entre réplicas. Primeiro, é feita uma certificação local e, só no caso de não haver conflitos, é que se tentam adquirir os *leases* necessários. Quando a réplica local já possui todos os *leases* é feita a certificação final, e se não houver conflitos é enviado o WS. Finalmente, depois de aplicar as modificações na réplica local, libertam-se os *leases* adquiridos.

Máquina de Estados Replicada para SGBD Kemme, Pedone, Alonso, Schiper, & Wiesmann (2003) apresentam um algoritmo que tira vantagem da primitiva OABcast. Ao usar OABcast, pode-se beneficiar da existência de redes onde se verifica a ordenação espontânea das mensagens (por exemplo, as LAN). Nestas redes existe uma grande probabilidade das mensagens serem entregues de um modo optimista por ordem total (Pedone & Schiper 1998). Segundo resultados experimentais descritos por Kemme, Pedone, Alonso, Schiper, & Wiesmann (2003), as mensagem são entregues fora da ordem nos vários nós devido, principalmente, a erros de *buffer overflow* e não a transmissões concorrentes. Este algoritmo usa transacções estáticas que podem estar num estado definido na Figura 2.3.

Cada item de dados tem uma fila de trincos e o algoritmo baseia-se em transacções estáticas. Quando a transacção é recebida os respectivos trincos são colocados nos itens acedidos pela transacção. Por uma questão de desempenho, existem dois tipos de trincos: de leitura e de escrita. À medida que são libertados, as operações da transacção são efectuadas. A libertação dos trincos depende do tipo do primeiro trinco da fila: (i) se for de leitura, então são libertados todos os trincos de leitura seguintes até encontrar um de escrita; (ii) se for de escrita, então só esse é libertado. Falta referir que, caso a transacção seja entregue fora de ordem, as operações são anuladas e os seus trincos reposicionados nas filas. No final, os trincos adquiridos são libertados.

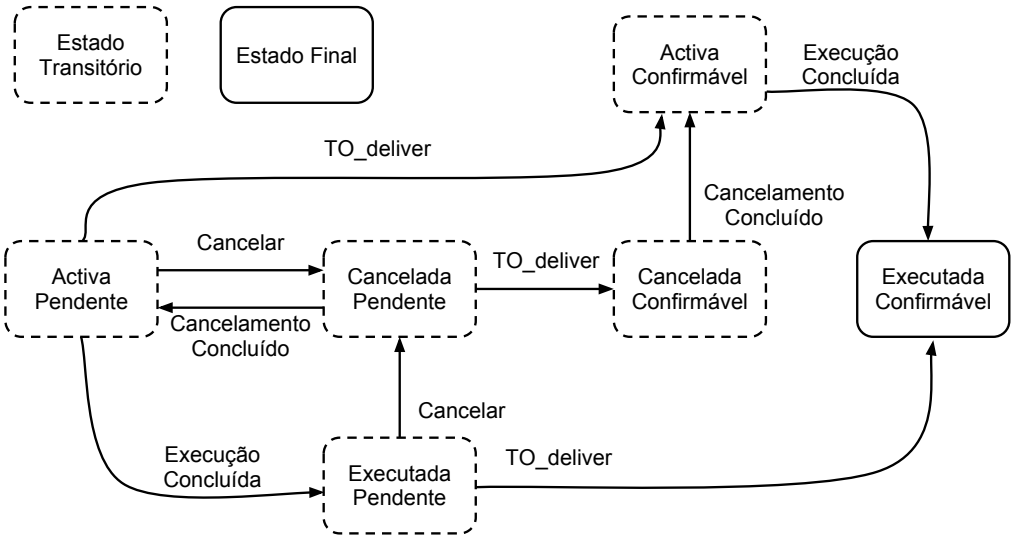


Figura 2.3: Estados das transacções

Máquina de Estado Replicada para STM O AGGRO (Palmieri, Quaglia, & Romano 2010) é outro algoritmo de replicação de STM baseado em OABcast. Quando uma transacção é iniciada, é enviada através desta primitiva para todas as réplicas. Assim que é recebida na ordem espontânea, a transacção começa logo a sua execução. A detecção de conflitos é verificada cada vez que é escrito ou lido algum valor. Além disso, permite ler dados de transacções que acabaram a sua execução mas que ainda não foram confirmadas. Isto leva a que as transacções sejam ligadas entre si. Se uma delas cancelar, devido a ler valores inválidos ou por ter sido entregue fora de ordem final, pode ocorrer um cancelamento em cadeia. No entanto, em redes com entrega espontânea, esses casos são poucos frequentes e consegue-se um aumento do desempenho do sistema.

Replicação Parcial para SGBD baseado em 2PC Sousa, Pedone, Moura, & Oliveira (2001) apresentam um algoritmo baseado em 2PC denominado *Resilient Atomic Commit* (RAC). Neste algoritmo, quando a transacção termina, o seu *dataset* é enviado para todas as réplicas através do OABcast. Tal como no 2PC, cada réplica realiza uma certificação local e envia o seu voto (sim ou não) consoante o resultado da certificação. O envio do voto pode ser efectuado de dois modos: (i) cada réplica envia o voto para as restantes réplicas envolvidas; ou (ii) cada réplica envia o voto para uma das réplicas (normalmente a réplica que executou a

transacção) e essa réplica indica às restante o resultado da transacção. Independente do modo, no final todas as réplicas chegam à mesma decisão (confirmada ou cancelada). Se a ordem final da transacção for diferente da espontânea, repetem o algoritmo.

O algoritmo de Schiper, Schmidt, & Pedone (2006) utiliza uma aproximação diferente. Neste algoritmo, as transacções são enviadas para todas as réplicas através da primitiva RBcast, que não garante ordem total. Por outro lado, este algoritmo usa a abstracção de consenso de modo a determinar uma ordem de seriação para as transacções. Para cada transacção decidida no consenso, cada réplica envolvida envia o seu voto para as restantes réplicas. Se todas as réplicas receberem votos positivos, então a transacção é confirmada.

Replicação Parcial para STM baseado em Certificação Serrano, Patino-Martinez, Jimenez-Peris, & Kemme (2007) identificam o uso de *1-Copy Serializibility* (1CS) como um dos factores que limita a capacidade de escala de um sistema. Assim, é apresentado um algoritmo com modelo de coerência *1-Copy Snapshot Isolation* (1CSI). Este tem a principal vantagem de não necessitar de enviar o RS, o que reduz o tamanho das mensagens trocadas. Usando 1CSI não ocorrem conflitos leitura-escrita, o que reduz a taxa de cancelamento. Quando a transacção é iniciada, é atribuída uma estampilha temporal de início (ST). No final da execução da transacção, esta é enviada para todas as réplicas. Cada réplica que possua itens acedidos pela transacção efectua a certificação da mesma, mas antes é atribuída uma estampilha temporal de confirmação (CT). A certificação de T é feita verificando se não existe nenhuma transacção T' tal que $T.WS \cap T'.WS \neq \emptyset \wedge T'.ST \leq T.CT \leq T'.CT$.

O P-Store (Schiper, Sutra, & Pedone 2010) é um exemplo de um sistema que usa um algoritmo genuíno. As réplicas do sistema são divididas em grupos e os dados são particionados entre os grupos, de modo a garantir que, no mesmo grupo, todas as réplicas possuem os mesmos dados. No entanto, isto não impede que um item esteja replicado em vários grupos. O algoritmo funciona do seguinte modo: quando a transacção acaba a sua execução, é enviada uma mensagem através da primitiva AMcast (ver Secção 2.4.1.2.4) para todas as réplicas que possuem itens acedidos pela transacção. Quando a transacção é entregue, é colocada numa fila. Se a transacção for local, isto é, se só acede a dados dentro de um grupo, cada réplica pode decidir por si, sem necessitar de trocar qualquer informação; caso contrário, diz-se que a transacção é global e todas as réplicas trocam votos entre si para decidir o resultado da transacção.

2.5 Discussão

Nesta secção abordou-se o problema de desenvolver um sistema de armazenamento de dados escalável, com elevado desempenho e tolerante a falta. Os SGBDs têm, para muitas aplicações, sido o sistema de armazenamento preferencial, pois suportam modelos de dados estruturados e o acesso aos mesmos garantido as propriedades ACID. No entanto, requisitos de desempenho e de capacidade de escala têm motivado a utilização de sistemas alternativos, que oferecem menos funcionalidades mas de forma muito eficiente. Neste contexto, as grelhas-de-dados, suportando distribuição, replicação em memória, e a execução de transacções por software, apresentam uma relação custo-benefício interessante no espaço de soluções. O trabalho desenvolvido nesta dissertação, foca-se neste último tipo de sistemas.

3 Replicação Parcial para Sistemas de Armazenamento em Memória

Neste capítulo são apresentados algoritmos para suportar replicação parcial em sistemas de armazenamento em memória. O capítulo começa por fazer uma análise da concretização actual de um sistema em particular (o *Infinispan*), identificando a limitação existente nessa concretização. De seguida, apontam-se estratégias para superar esta limitação e apresentam-se algoritmos seguindo essas estratégias. Finalmente descreve-se uma concretização dos algoritmos propostos, obtida através da integração destas soluções numa versão alterada do *Infinispan*.

3.1 *Infinispan*: Limitação, Desafios e Soluções

Nesta secção faz-se uma breve descrição das estratégias concretizadas na distribuição actual do *Infinispan*, identifica-se a sua principal limitação e enumera-se um conjunto de estratégias para superar esta limitação.

3.1.1 Funcionamento do Sistema

O *Infinispan*, tal como a grande maioria das grelhas-de-dados em memória, oferece um meio de armazenamento temporário de dados com elevado desempenho, disponibilidade e capacidade de escala. Esta combinação de características é sobretudo conseguida à custa de enfraquecer o modelo de coerência oferecido aos programadores, não dando garantias de serializabilidade. A concretização actual caracteriza-se também por utilizar algoritmos relativamente simples para efectuar o controlo de concorrência e a confirmação distribuída das transacções, nomeadamente recorrendo ao uso de trincos e à confirmação em duas fases (2PC).

Modelo de Coerência O Infinispan oferece alguns modelos de coerência fracos. A designação de modelo fraco advém da comparação com modelos que tentam mimetizar o comportamento de um sistema não distribuído, e portanto não replicado, onde os dados não são acedidos de forma

concorrente. Neste contexto, um modelo forte oferece aquilo que se designa por *1-Copy Serializability* (1CS), garantindo que a execução concorrente de múltiplas transacções em vários nós do sistema é equivalente a uma execução em série destas transacções numa única máquina (Bernstein, Hadzilacos, & Goodman 1986).

A escolha de um critério mais fraco melhora o desempenho do sistema, permitindo a utilização de algoritmos de controlo de concorrência e de coordenação distribuídos mais eficientes. Normalmente, sistemas como o *Infinispan*, suportam os seguintes critérios de coerência (fraca) (Berenson, Bernstein, Gray, Melton, O'Neil, & O'Neil 1995):

- *Read Committed*: daqui em diante referido como RC, que garante que uma transacção só lê os valores confirmados;
- *Repeatable Read*: daqui em diante referido como RR, que garante que duas leituras consecutivas na mesma transacção não retornam valores diferentes;
- *Repeatable Read* com verificação do *Write Skew*: daqui em diante referido como RR+WS, que, além de garantir as mesmas propriedades que RR, verifica se o valor de um chave foi alterado entre uma leitura e uma escrita numa transacção (cancelando a transacção, se for o caso). O mecanismo de verificação do *write skew* é opcional e é oferecido pelo protótipo usado, o *Infinispan*.

Confirmação de Transacções A confirmação das transacções é efectuada no *Infinispan* através do algoritmo de confirmação em duas fase, 2PC. A concretização baseia-se no padrão XA[1], uma especificação criada pelo *The Open Group* para coordenar o processamento de transacções distribuídas. Este padrão descreve uma interface e uma especificação, e tem como objectivo permitir que vários recursos (tais como, base de dados, servidores aplicacionais, caches, etc.) sejam acedidos no contexto de uma única transacção, preservando as propriedades ACID no acesso ao conjunto de recursos. Durante a execução do algoritmo podem ser invocadas várias primitivas, das quais se podem identificar as mais relevantes para este trabalho, nomeadamente: `xa_prepare`, que dá inicio à execução do algoritmo 2PC entre as várias réplicas, o `xa_commit` e o `xa_rollback` que terminam o algoritmo com a segunda fase, através da confirmação ou cance-

[1]XA: `https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?catalogno=c193`

lamento da transacção, respectivamente. O algoritmo 2PC foi descrito com mais pormenor na Secção 2.4.1.1.

3.1.2 Limitação

A principal limitação na concretização actual usada pelo *Infinispan* é a existência de interbloqueios. Como referido anteriormente, a confirmação de uma transacção é efectuada recorrendo ao algoritmo 2PC. Tal como o nome indica, este protocolo confirma a transacção através de duas fases. Na primeira fase, prepara-se a transacção nas restantes réplicas, na segunda fase confirma-se ou cancela-se a transacção. O controlo da concorrência é obtido através da utilização de trincos. Estes são obtidos na primeira fase do algoritmo de 2PC, assegurando que a transacção tem acesso exclusivo aos dados (Secção 2.4.1.1).

Esta solução tem um bom desempenho no caso de existirem poucos conflitos entre transacções. Mas em cenários em que os conflitos são comuns, nomeadamente em situações de sobrecarga, esta forma de adquirir os trincos pode levar à ocorrência de situações de interbloqueio distribuído, uma vez que réplicas diferentes podem tentar obter os mesmos trincos, mas por ordem diferente (Secção 2.4.2.2). Estes interbloqueios são difíceis de detectar e normalmente recorre-se à utilização de temporizadores para a sua resolução, um mecanismo pouco eficiente. Um mecanismo simplificado para a detecção de interbloqueios consiste em tentar adquirir o trinco sem bloquear e, caso o trinco esteja adquirido por outra transacção, cancelar a transacção. Embora mais eficiente do que recorrer a temporizadores, pois evita o tempo de espera até o temporizador terminar, esta solução não é óptima.

3.1.3 Soluções

Apresentam-se agora estratégias para superar o problema acima identificado. Este é abordado de forma faseada, onde primeiro discute-se a sua solução no caso mais simples da replicação total e só depois no caso da replicação parcial.

Abordagem ao problema para replicação total Para evitar situações de interbloqueio em sistemas de replicação total, propõe-se a utilização de primitivas de comunicação em grupo que garantem a entrega ordenada de mensagens nas réplicas do sistema (ABcast). Ao assegurar que

todas as réplicas processam os pedidos de aquisição de trincos pela mesma ordem, evitam-se as situações de interbloqueio entre transacções.

Abordagem ao problema para replicação parcial Para evitar situações de interbloqueio em sistemas de replicação parcial, usa-se uma solução estruturalmente semelhante à usada para replicação total. No entanto, evita-se a utilização de ABcast para ordenar as mensagens, uma vez que esta primitiva envolve todas as réplicas do sistema de armazenamento, enquanto apenas um sub-conjunto destas está de facto envolvido em cada transacção. Desta forma, propõe-se a utilização de primitivas de difusão em grupo com a capacidade de ordenar mensagens enviadas para grupos distintos, sempre que existam membros comuns a dois ou mais grupos. Infelizmente, existem poucos sistemas de comunicação em grupo que suportam esta funcionalidade. Em particular, este tipo de garantias não é oferecido pelo *JGroups*[2], o sistema de comunicação em grupo usado pelo *Infinispan*.

Uma maneira simples de conseguir ordem total entre mensagens para grupos diferentes consiste em simular a difusão em grupo enviando todas as mensagens em ordem total para todas as réplicas (isto é, para um único super-grupo, que é a união de todos os sub-grupos possíveis). Posteriormente, cada réplica descarta as mensagens que não lhe são destinadas. Este tipo de solução é designada por não-genuína (Serrano, Patino-Martinez, Jimenez-Peris, & Kemme 2007), trazendo poucas vantagens em relação à solução usada para replicação total do ponto de vista do desempenho, pois na prática todas as réplicas são envolvidas em todas as transacções, perdendo-se a capacidade de dispersão de carga proporcionada pelo particionamento dos dados por diferentes réplicas. Como tal, propõe-se o desenvolvimento de um protocolo de difusão selectiva com ordem total genuína para o *JGroups*.

3.2 Algoritmos de Replicação

Os critérios de coerência RC e RR são concretizados usando os seguintes algoritmos. O espaço de dados é dividido em dois conjuntos: um contexto global, onde são registadas as actualizações já confirmadas, e contextos locais, um para cada transacção em execução. O objectivo do contexto local é guardar os valores escritos pela transacção até que esta possa ser

[2]JGroups: `http://www.jgroups.org/`

confirmada. O contexto local pode também guardar valores lidos pela transacção, caso seja necessário isolar a transacção em execução de escritas posteriores nos mesmos dados.

Em pormenor, a execução de uma transacção assegurando o critério RC executa-se da seguinte forma. Sempre que a transacção lê um item que não tenha escrito, obtém o seu valor do contexto global. Sempre que a transacção realiza uma escrita, armazena o valor escrito no contexto local; posteriores leituras deste item são feitas a partir do contexto local. Se a transacção é cancelada o contexto local é descartado. Se a transacção é confirmada, todas as actualizações são aplicadas atomicamente no contexto global.

A execução de uma transacção assegurando o critério RR é semelhante à descrita anteriormente, com a seguinte diferença: quando a transacção lê um item pela primeira vez, obtém o valor do contexto global e armazena-o no contexto local; posteriores leituras a este item são realizadas no contexto local.

Uma grande vantagem dos algoritmos descritos anteriormente é que não requerem o conhecimento do conjunto de leitura para validar a transacção. De facto, vários trabalhos mostram que o conjunto de leitura de transacções em memória pode ser muito grande, e que a sua propagação na rede pode facilmente tornar-se num ponto de estrangulamento do sistema (Couceiro, Romano, Carvalho, & Rodrigues 2009; Serrano, Patino-Martinez, Jimenez-Peris, & Kemme 2007).

Os algoritmos que se apresentam de seguida, seguem os passos acima enumerados, concentrando-se o trabalho aí descrito na optimização do passo de confirmação, em que as actualizações são aplicadas de forma atómica em todas as réplicas.

Replicação total para modelo de coerência de RC e RR Como exposto anteriormente, alterou-se o algoritmo de confirmação das transacções de forma a utilizar a primitiva ABcast. Desta forma, beneficia-se do facto de todas as réplicas receberem a mesma informação pela mesma ordem, o que impede a existência de cenários de interbloqueio. Logo, deixam também de existir razões para uma transacção abortar na primeira fase da processo de confirmação em duas fases. O algoritmo, cujo pseudo-código se apresenta no Algoritmo 1 na página 36, consiste então nos seguinte passos: na primeira fase do 2PC não existe qualquer interacção com qualquer outra réplica e é sempre retornada uma resposta positiva (esta fase é apenas executada para respeitar o padrão XA). Na segunda fase, envia-se o conjunto de escrita da transacção para todas as réplicas por ordem total. Quando a mensagem com o conjunto de escrita é recebida,

as actualizações são aplicadas no contexto global (isto é, feito em todas as réplicas). Após este passo, o fio de execução que solicitou a confirmação da transacção é desbloqueado na réplica de origem.

Algorithm 1 Replicação Total para RC e RR

```
 1: function XA_PREPARE(T)
 2:     return xa_ok                          ▷ Nenhuma operação é executada nesta fase
 3: end function

 4: function XA_COMMIT(T)                     ▷ Fio de execução que executou a transacção (F1)
 5:     if T é só de leitura then
 6:         return xa_ok
 7:     end if
 8:     TO-BROADCAST(T)
 9:     bloquear fio de execução F1
10:     return xa_ok
11: end function

12: function XA_ROLLBACK(T)
13:     descartar transacção T
14:     return xa_ok
15: end function

16: upon TO-DELIVER(T)          ▷ Fio de execução que entrega transacções em ordem total (F2)
17:     aplicar T.WS
18:     desbloquear fio de execução F1        ▷ Só tem efeito na réplica que executou a transacção
```

Replicação total para modelo de coerência de RR+WS Este algoritmo segue o mesmo princípio do algoritmo anterior. No entanto, para assegurar o critério de coerência RR+WS, é necessário fazer a verificação do *write skew* na confirmação da transacção. Para fazer esta verificação, é necessário ter acesso ao conjunto de leitura da transacção. Desta forma, ou se envia o conjunto de leitura para todas as réplicas (que deste modo podem localmente fazer a verificação do *write skew*) ou se delega esta tarefa na réplica onde a transacção foi executada, a qual deve posteriormente notificar as restantes réplicas do destino da transacção. Devido aos problemas de desempenho associados à propagação do conjunto de leitura, discutidos anteriormente, optou-se pela segunda solução. Note-se que a verificação do *write skew* só pode ser efectuada quando a ordem total da transacção já é conhecida. Desta forma, utiliza-se a primeira fase da confirmação para propagar o conjuntos de escrita com ordem total, e a segunda fase para notificar todas as réplicas do resultado da verificação do *write skew*. O pseudo-código pode ser visualizado no Algoritmo 2 na página 38.

Em pormenor, na primeira fase o WS é enviado em ordem total. Assim que é recebido, cada réplica prepara a transacção e espera pela segunda fase do 2PC. Para além disso, a réplica

que executou a transacção faz a verificação do *write skew*. Se nenhum conflito for detectado, é enviada uma mensagem de confirmação. A recepção da mensagem de confirmação faz com que as modificações sejam aplicadas. Caso seja detectado um ou mais conflitos, uma mensagem de cancelamento é enviada que descarta as modificações. O envio da mensagem de confirmação ou cancelamento é efectuado sem ordem total.

Replicação parcial para modelo de coerência RC e RR Para efectuar uma leitura em replicação total, primeiro verifica-se o contexto local da transacção e, se não existir, verifica-se o contexto global da réplica. No entanto, quando se usa replicação parcial, o item de dados a procurar pode não estar armazenado localmente. Desse modo, as operações de leitura podem necessitar de um passo adicional onde têm de consultar o contexto global de outras réplicas para obter o item de dados pretendido. Este é um custo inerente à replicação parcial que não pode ser evitado. O algoritmo para este problema é em tudo semelhante ao algoritmo para replicação total com a excepção da primitiva de comunicação em grupo. Enquanto que em replicação total a primitiva é o ABcast, em replicação parcial, para tornar o algoritmo genuíno, a primitiva usada é o AMcast. O pseudo-código é igual ao Algoritmo 1 página 36, com a excepção da Linha 8 onde o TO-BROADCAST(T) é substituído pelo, A-MULTICAST(T).

Replicação parcial para modelo de coerência RR+WS A ideia é semelhante ao algoritmo para replicação total, mas foi necessário adaptar o algoritmo de modo a suportar a distribuição dos dados. Na replicação parcial, não existe nenhuma réplica com todos os itens de dados. Assim, a verificação do *write skew* necessita de ser realizada de forma distribuída. Cada réplica pode detectar apenas os conflitos referentes aos dados que armazena e, para esse efeito, necessita de ter acesso ao conjunto de leitura da transacção. Desta forma, conjuntamente com o WS, é necessário enviar o RS na difusão da mensagem na primeira fase. Tal como na replicação total, na segunda fase o coordenador envia o comando de confirmação ou cancelamento. Recapitulando, na primeira fase do 2PC, além das réplicas prepararem os dados para a confirmação da transacção, também efectuam a verificação do *write skew* nos dados que possuem. Se todas as réplicas conseguirem preparar a transacção e efectuar a verificação com sucesso, então a transacção é confirmada. Caso contrário é cancelada. O pseudo-código apresentado só inclui as funções e eventos com comportamento diferente do apresentado no Algoritmo 2 e pode ser encontrado no Algoritmo 3 na página 39.

Algorithm 2 Replicação Total para RR+WS

```
1: tx_remotas ← ∅                                   ▷ Conjunto com o ID e write set de uma transacção remota

2: function XA_PREPARE(T)                           ▷ Fio de execução que executou a transacção (F1)
3:     if T é só de leitura then
4:         return xa_ok
5:     end if
6:     TO-BROADCAST(T)
7:     marcar T como preparada
8:     bloquear fio de execução F1
9:     if T.resultado_write_skew == true then
10:        return xa_ok
11:    else
12:        return xa_err
13:    end if
14: end function

15: function XA_COMMIT(T)
16:    if T é só de leitura then
17:        return xa_ok
18:    end if
19:    R-BROADCAST(T.id,CONFIRMAR)
20:    aplicar T.WS
21:    desbloquear fio de execução F2
22:    return xa_ok
23: end function

24: function XA_ROLLBACK(T)
25:    if T foi preparada then
26:        R-BROADCAST(T.id,CANCELAR)
27:    end if
28:    descartar T
29:    desbloquear fio de execução F2
30:    return xa_ok
31: end function

32: upon TO-DELIVER(T) NA RÉPLICA i        ▷ Fio de execução que entrega a transacção em ordem total (F2)
33:    if T.id foi executada em i then
34:        T.resultado_write_skew ← verificar write skew
35:        desbloquear fio de execução F1
36:    else
37:        tx_remotas.add(T.id, T.WS)
38:    end if
39:    bloquear fio de execução F2

40: upon R-DELIVER(tx_id,op) NA RÉPLICA i
41:    if tx_id foi executada em i then
42:        return
43:    end if
44:    if op == CONFIRMAR then
45:        ws ← tx_remotas.obtem(tx_id)
46:        aplicar ws
47:    end if
48:    desbloquear fio de execução F2
49:    tx_remotas.remover(tx_id)
```

Algorithm 3 Replicação Parcial baseada em AMcast para RR+WS

```
1: function XA_PREPARE(T)
2:     if T é só de leitura then
3:         return xa_ok
4:     end if
5:     A-MULTICAST(T)
6:     marcar T como preparada
7:     bloquear fio de execução até receber todos os resultados da verificação do write skew
8:     if passou verificação do write skew then
9:         return xa_ok
10:    else
11:        return xa_err
12:    end if
13: end function

14: upon A-DELIVER(T) NA RÉPLICA i        ▷ Fio de execução que entrega a transacção em ordem total (F2)
15:    verificar write skew para as chaves locais
16:    enviar de volta o resultado da verificação
17:    if T.id não foi executada em i then
18:        tx_remotas.add(T.id, T.WS)
19:    end if
20:    bloquear fio de execução F2
```

3.3 Algoritmo de Difusão Atómica Selectiva

Atendendo a que o sistema de comunicação em grupo usado pelo *Infinispan*, o *JGroups*, não possui suporte para difusão atómica selectiva, foi necessário desenvolver este serviço, de modo a poder concretizar os algoritmos acima descritos. Foram estudados dois algoritmos distintos para suportar difusão atómica selectiva, cada um com diferentes vantagens e desvantagens: um dos algoritmos possui menor latência, pois requer a execução de um menor número de passos de comunicação, e o outro tem um menor custo de comunicação, enviando menos mensagens. O interesse em estudar estas duas alternativas prende-se com a necessidade de aferir qual destes aspectos possui maior impacto no desempenho final do sistema.

Difusão Atómica Selectiva (latência de três passos de comunicação) Este protocolo é inspirado no protocolo utilizado na concretização original do sistema ISIS (Schiper, Birman, & Stephenson 1991) e opera da seguinte maneira. O protocolo troca três tipos de mensagens: mensagens de DADOS, usadas para enviar a informação do emissor para os destinatários; mensagens de VOTO, usadas pelos destinatários para proporem um número de ordem para a mensagem; e mensagens de ORDEM, usadas pelo emissor para atribuir um número de ordem final às mensagens de dados. Cada máquina possui um relógio lógico (Lamport 1978) que é incrementado quando recebe mensagens de DADOS ou mensagens de ORDEM. A difusão ordenada é iniciada

através do envio uma mensagem de DADOS para o grupo de réplicas participantes na transacção. Quando esta mensagem é recebida, cada réplica incrementa o seu relógio lógico, atribui essa estampilha temporal à mensagem, e coloca-a numa fila ordenada no estado *Pendente*; o número de ordem local atribuído à mensagem é então enviado para o emissor através de uma mensagem de VOTO. O emissor, recolhe os vários números de sequência atribuídos pelas diferentes réplicas, calcula o máximo destes valores, e envia este número para todas as réplicas numa mensagem de ORDEM. Ao receber uma mensagem de ORDEM, cada réplica actualiza o número da mensagem, reordenando-a na fila caso seja necessário, marca a mensagem como *Final*, e actualiza o seu relógio lógico. Finalmente, as mensagens são entregues quando estão no estado *Final* e se encontram na cabeça da fila, isto é, não existe nenhuma outra mensagem *Pendente* ou *Final* com um número inferior. O pseudo-código encontra-se no Algoritmo 4 na página 41.

Difusão Atómica Selectiva (latência de dois passos de comunicação) O algoritmo anterior requer a execução de três passos de comunicação e a troca de $3D$ mensagens, onde D é o número de destinatários. Apresenta-se agora uma variante do algoritmo anterior que só requer a execução de dois passos de comunicação mas troca $D + (D-1)^2$ mensagens. A redução do número de passos é conseguida fazendo com que as mensagens de VOTO de cada participante sejam enviadas directamente para todos os outros participantes, que calculam o número de ordem final de forma independente mas determinista. Em detalhe, quando uma mensagem de DADOS é recebida por uma réplica, esta actualiza o seu relógio lógico para o valor máximo entre o valor que vem com a mensagem e o valor incrementado do relógio lógico. Este valor é atribuído à mensagem, a qual é colocada na fila no estado *Pendente*. Esse valor é então enviado para os restantes participantes numa mensagem de VOTO. Quando uma réplica recebe uma mensagem de VOTO, o número de ordem da mensagem é actualizado e a mensagem é reordenada na fila, se necessário. Além disso, o seu relógio lógico é actualizado se for necessário. Quando receber todas as mensagens de VOTO de todas os destinatários de uma dada mensagem de DADOS, então a mensagem é marcada como *Final*. A última etapa é comum à primeira versão. O pseudo-código pode ser visualizado no Algoritmo 5 na página 42. A Figura 3.1 ilustra o funcionamento de ambas as soluções.

Algorithm 4 Difusão Atómica Selectiva em três passos de comunicação

```
1: mensagens_enviada ← ∅                ▷ mantém as mensagens enviadas até ser decidida a ordem final
2: fila_ordenada ← ∅    ▷ mantém as mensagens recebidas ordenadas, até serem entregues à aplicação na sua ordem final

3: function A-MULTICAST(m)
4:     for réplica r em m.dest do
5:         SEND-UNICAST(DADOS,m, r)
6:     end for
7:     temp_ts ← 0
8:     mensagens_enviadas.adiciona(m.id, temp_ts)
9: end function

10: upon RECEIVE-UNICAST(DADOS,m) NA RÉPLICA r
11:     atomic {
12:         relogio_logico ← relogico_logico + 1
13:         ts ← relogio_logico
14:         fila_ordenada.adiciona(m.id, m.dados, ts, PENDENTE)
15:     }
16:     SEND-UNICAST(VOTO,[m.id, ts], m.origem)

17: upon RECEIVE-UNICAST(VOTO,[m.id, ts]) NA RÉPLICA r
18:     atomic {
19:         temp_ts, ← mensagens_enviadas.obterTs(m.id)
20:         temp_ts ← max(ts, temp_ts)
21:         mensagens_enviadas.actualizar(m.id, temp_ts)
22:     }
23:     if recebeu todos os votos para m.id then
24:         for réplica r em m.dest do
25:             SEND-UNICAST(ORDEM,[m.id, temp_ts], r)
26:         end for
27:         mensagens_enviadas.remover(m.id)
28:     end if

29: upon RECEIVE-UNICAST(ORDEM,[m.id, ts])
30:     atomic {
31:         relogio_logico ← max(relogico_logico, ts)
32:         fila_ordernada.actualizar(m.id, ts, FINAL)
33:     }

34: while true do                       ▷ efectuado pelo fio de execução de entrega
35:     ⟨id, dados, estado⟩ ← fila_ordenada.primeiroElemento()
36:     if estado == FINAL then
37:         fila_ordenada.remover(id)
38:         A-DELIVER(dados)
39:     end if
40: end while
```

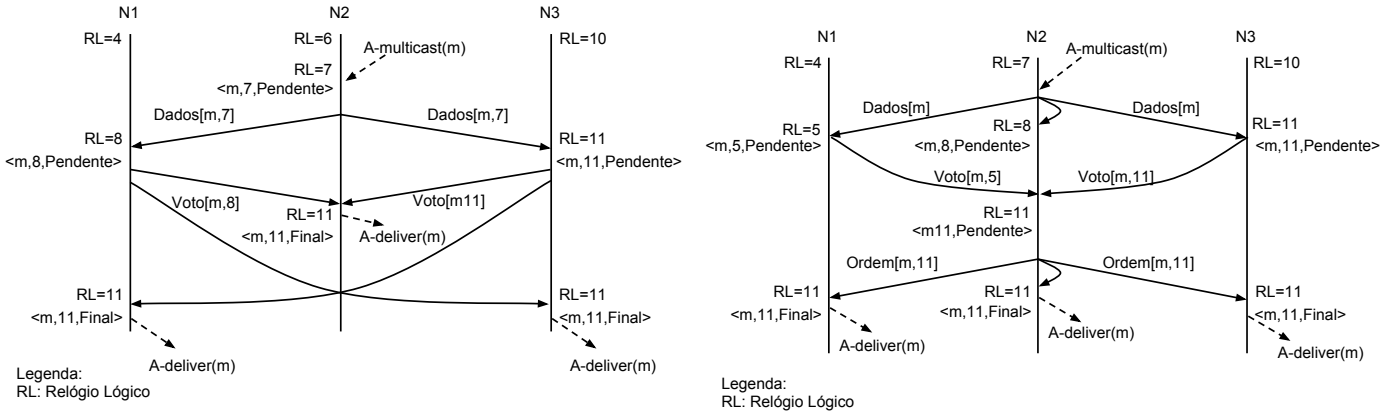
Algorithm 5 Difusão Atómica Selectiva em dois passos de comunicação

```
1: fila_ordenada ← ∅        ▷ mantém as mensagens recebidas ordenadas, até serem entregues à aplicação na sua ordem final

2: function AM-SEND(m)                                                    ▷ na réplica i
3:     atomic {
4:         relogio_logico ← relogio_logico + 1
5:         ts ← relogio_logico
6:         fila_ordernada.adiciona(m.id, m.dados, ts, PENDENTE)
7:     }
8:     for réplica r em m.dest do                                         ▷ excepto para a réplica i
9:         SEND-UNICAST(DADOS,[ts, m], r)
10:    end for
11: end function

12: upon RECEIVE-UNICAST(tipo,[ts, m]) NA RÉPLICA i
13:    atomic {
14:        relogio_logico ← max(relogico_logico + 1, ts)
15:        if fila_ordenada.contem(m.id) then
16:            temp_ts ← fila_ordenada.obterTs(m.id)
17:            temp_ts ← max(temp_ts, relogio_logico)
18:            fila_ordernada.actualizar(m.id, temp_ts, PENDENTE)
19:        else
20:            temp_ts ← relogio_logico
21:            fila_ordernada.adiciona(m.id, temp_ts, PENDENTE)
22:        end if
23:    }
24:    if é a primeira vez que recebe m.id then
25:        for réplica r em m.dest do                                     ▷ excepto para a réplica i
26:            SEND-UNICAST(VOTO,[temp_ts, m.id], r)
27:        end for
28:    end if
29:    if tipo == DADOS then
30:        fila_ordernada.actualizar(m.id, m.dados)
31:    end if
32:    if recebeu todos os votos para m.id then
33:        fila_ordernada.actualizar(m.id, FINAL)
34:    end if

35: while true do                                                         ▷ efectuado pelo fio de execução de entrega
36:    ⟨id, dados, estado⟩ ← fila_ordenada.primeiroElemento()
37:    if estado == FINAL then
38:        fila_ordenada.remover(id)
39:        A-DELIVER(dados)
40:    end if
41: end while
```



(a) Esquema com dois passos de comunicação

(b) Esquema com três passos de comunicação

Figura 3.1: Esquema da Difusão Atómica Selectiva

3.4 Concretização no *Infinispan*

O *Infinispan* é uma grelha-de-dados em memória, transaccional, distribuída e replicada, desenvolvida em código aberto pela RedHat na linguagem Java. O modelo de programação oferecido é o de um mapa que mantém uma projecção entre chaves e valores, permitindo inserir, ler, actualizar e remover objectos de dados através de uma chave que lhes é associada pelo programador. Esta interface é uma extensão da interface `java.util.Map` do Java e respeita o padrão JSR-107[3] (interface *JCache*).

O *Infinispan* oferece suporte para transacções e dois modos operacionais: replicação total e replicação parcial (referido como *replicação* e *distribuição* no sistema). No primeiro caso os dados armazenados isto é, os pares chave/valor, são replicados por todos os nós e no segundo caso só por um sub-conjunto dos mesmo. As transacções tem compatibilidade com a interface JTA (*Java Transaction API*) e a sua execução e confirmação respeita o padrão XA.

Uma transacção é uma sequência de operações sobre o mapa e é delimitada por métodos que identificam o início e fim da transacção. Uma transacção é iniciada invocando o método `begin()` e termina com a invocação do método `commit()` ou `rollback()`. Se o método `commit()` termina com sucesso, todas as operações de escrita executadas durante a transacção tornam-se visíveis no mapa, caso contrário, as actualizações são descartadas.

De um modo genérico, as operações de leitura nunca cancelam nem adquirem trincos durante

[3]http://jcp.org/en/jsr/detail?id=107

a sua execução. No entanto, consoante o critério de coerência escolhido, o valor devolvido pela operação pode ser diferente. Serão dados mais pormenores sobre estes aspectos nas secções seguintes.

Por outro lado, as operações de escrita necessitam de adquirir trincos, o que pode levar ao cancelamento de uma transacção. A aquisição de trincos é efectuada dinamicamente sem qualquer tipo de ordenação, e o tempo de espera pelo trinco é limitado por temporizadores. Deste modo, uma operação de escrita pode falhar porque: (i) o temporizador expirou ao tentar adquirir um trinco, (ii) ocorreu uma situação de interbloqueio ou (iii) falhou a verificação do *write skew*. As duas últimas situações só ocorrem se certos mecanismos estiverem activados, nomeadamente a detecção e resolução de interbloqueios e o mecanismo de *write skew* (modelo de coerência RR+WS), respectivamente.

No final da execução de uma transacção é necessário validar a transacção em todas as réplicas. Para tal, o *Infinispan* utiliza um protocolo de certificação clássico em duas fases, o 2PC. Na primeira fase, tenta adquirir os trincos remotos sobre todas as chaves modificadas. Cada réplica devolve ao coordenador, réplica que iniciou o protocolo, se conseguiu adquirir todos os trincos necessários ou não. Finalmente, na segunda fase, o coordenador responde com o resultado final da transacção para todas as réplicas, isto é, confirma a transacção se não recebeu nenhuma resposta negativa, e cancela, caso contrário. A ocorrência de interbloqueios durante a confirmação tem um grande impacto no desempenho, como será demonstrado mais adiante.

3.4.1 Arquitectura do *Infinispan*

A Figura 3.2 mostra a arquitectura do *Infinispan*. Todas as operações efectuadas sobre a cache (leitura, escrita, pedido de confirmação, etc.) são encapsuladas em comandos internos. Esses comandos são enviados pela cadeia de interceptores que interceptam e interagem com o comando a processar. Dos interceptores existentes, os mais relevantes são o `TxInterceptor` (`DistTxInterceptor` em modo distribuído), o `LockingInterceptor` (`DistLockingInterceptor`) e o `ReplicationInterceptor` (`DistributionInterceptor`). Os comandos, quando são interceptados na cadeia, possuem um contexto no qual foram criados. Os contextos podem ser locais ou remotos, consoante o comando seja criado localmente ou de outra réplica do sistema, e podem ser transaccionais ou não, consoante o comando seja execu-

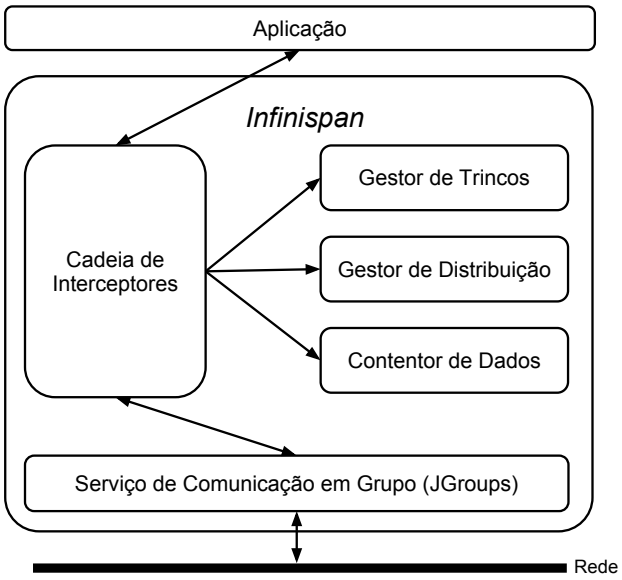


Figura 3.2: Arquitectura geral do *Infinispan*

tado dentro ou fora de uma transacção. Para ajudar no processamento dos comandos, existe ainda um conjunto de módulos com funcionalidades específicas. Entre os módulos existentes, os mais relevantes para este trabalho são o *Gestor de Trincos*, *Gestor de Distribuição* e *Contentor de Dados*, que tem como objectivo a gestão dos trincos, a distribuição das chaves pelas diversas réplicas e o armazenamento em memória, respectivamente.

Como supramencionado, uma transacção é executada de acordo com um dos critérios de coerência fraca oferecido e existem dois tipos de operações possíveis, leitura ou escrita. Uma operação designa-se de escrita sempre que um par chave/valor é actualizado, inserido ou apagado. O *Infinispan* implementa uma variante não serializável de um algoritmo de controlo de concorrência o qual nunca bloqueia ou cancela uma transacção numa operação de leitura, e utiliza uma estratégia de detecção de interbloqueio baseado em temporizadores, para detectar conflitos de escrita-escrita.

Entrando em mais pormenor sobre as operações de leitura, se o critério de coerência usado é o RC, então é retornado o valor mais recente e confirmado presente no contentor de dados. Se pelo contrário o modelo de coerência usado é RR, quando a transacção lê uma determinada chave, o seu valor é armazenado num contexto local à transacção, sendo este o valor retornado nas subsequentes operações de leitura. Em particular, o `TxInterceptor` associa a operação de

leitura à transacção e o `LockingInterceptor` tem a responsabilidade de obter o valor correcto. No modelo de coerência RC, caso a chave exista no contexto local, isto é, a chave foi previamente modificada pela transacção, esse valor é retornado, senão obtém-se o valor do contentor de dados. Caso seja o modelo de coerência RR, no `LockingInterceptor` é verificado se o valor existe no contexto local e, se existir, é esse o valor retornado. Caso contrário, obtém o valor do contentor de dados e armazena-o no contexto local.

É importante referir que as operações de leitura em modo distribuído podem necessitar de contactar outras réplicas, pois uma réplica não contém todos os dados. Isto é um custo inevitável da distribuição. O *Infinispan* tenta reduzir a frequência de acessos remotos através do uso de um mecanismo adicional, denominado cache L1. Este consiste em manter uma cópia local de chaves de outras réplicas durante um determinado período de tempo. O `DistributionInterceptor` é o interceptor responsável pelas leituras remotas e armazenamento do valor na cache L1, se este estiver activado. Neste interceptor, se se verificar que a chave não tem nenhum valor associado, é invocado ao `DistributionManager` a leitura remota da chave a outra réplica. O pedido é efectuado para todos as réplicas que possuem a chave em simultâneo e fica bloqueado até obter a primeira resposta válida ou até um temporizador disparar. O valor da chave é armazenado em cache L1, se este mecanismo estiver activo, sendo colocado no contexto local à transacção. A associação das chaves às réplicas é efectuada através de uma função de dispersão coerente. Esta função é determinista em todas as réplicas, e desse modo, qualquer réplica do sistema sabe quem são as réplicas que possuem uma determinada chave.

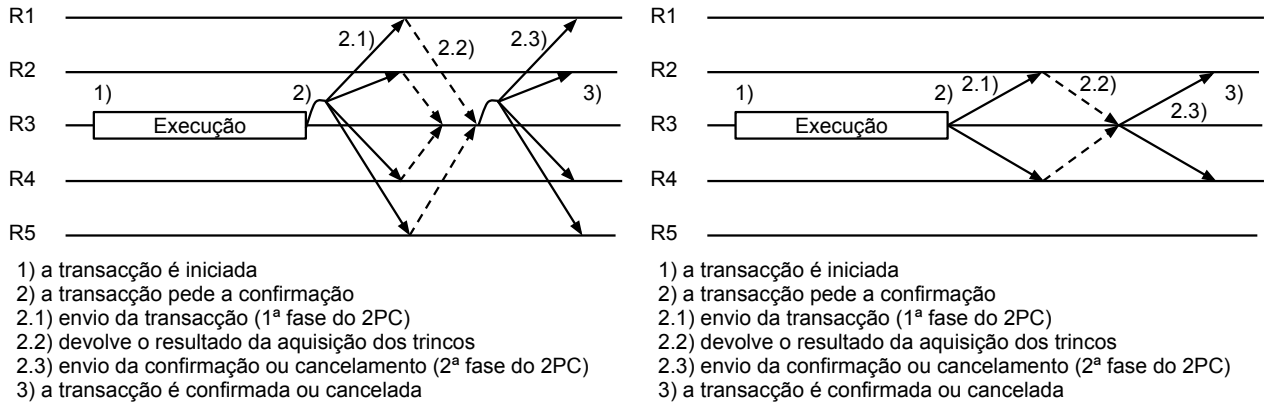
Por outro lado, numa operação de escrita o trinco correspondente à chave é adquirido localmente durante a execução da transacção. Se o mecanismo de verificação do *write skew* se encontra activo (critério de coerência RR+WS), uma verificação extra é efectuada durante a operação de escrita: se a transacção leu a chave previamente, e a chave não foi modificada pela mesma e, o seu valor actual é diferente do lido, depois de ter adquirido o trinco, a transacção é cancelada. Nas operações de escrita, os pares chave/valor são armazenados num contexto local à transacção e subsequentes escritas ou leituras dessas chaves são efectuadas sobre essa cópia local. Tal como nas leituras, o `TxInterceptor` associa a operação à transacção. O `LockingInterceptor` verifica se a chave já foi escrita previamente pela transacção, e em caso afirmativo, o valor é actualizado. Se a chave foi previamente lida, então só o trinco é adquirido e a verificação do *write skew* é efectuada, se necessário. Caso não seja nenhuma das anteriores,

o trinco é adquirido, a chave é obtida do contentor de dados e uma cópia é colocada no contexto local da transacção.

O protocolo de certificação apresenta algumas diferenças entre o modo replicado e distribuído. Na distribuição, não se faz qualquer uso do serviço de comunicação ponto-a-multiponto, sendo todas as mensagens trocadas através do serviço ponto-a-ponto fiável. Isto deve-se ao facto de uma transacção não necessitar de contactar todas as réplicas para ser certificada. Por outro lado, na replicação faz uso da comunicação ponto-a-multiponto fiável. A Figura 3.3 ilustra todo o processo de certificação em duas fases.

No final da execução de uma transacção, é efectuada a confirmação da mesma. Inicialmente, um comando de preparação (`PrepareCommand`) é criado (primeira fase do 2PC). A réplica que criou o comando é denominada de coordenador. Este comando é interceptado pelo `ReplicationInterceptor` (ou `DistributionInterceptor`) que se encarrega de o enviar para as restantes réplicas. Ao ser recebido nas restantes réplicas, os comandos de escrita associados são repetidos no `TxInterceptor`, quando esta intercepta o `PrepareCommand`, isto é, os comandos de escrita são enviados para a cadeia de interceptores. Estes comandos são executadas de igual modo ao descrito acima para as operações de escrita. No final é devolvido ao coordenador o valor `null` ou uma excepção, consoante o resultado obtido da invocação dos comandos de escrita na cadeia de interceptores. Tal como anteriormente referido, uma excepção pode ocorrer caso não consiga adquirir o trinco ou devido à verificação do *write skew*. Se o coordenador receber uma excepção, então a transacção deve ser cancelada. A primeira fase termina aqui.

Se a transacção puder ser confirmada, um comando de confirmação (`CommitCommand`) é criado e enviado. O envio do comando é efectuado pelo `ReplicationInterceptor` (ou pelo `DistributionInterceptor`). O comando é interceptado no `LockingInterceptor` que escreve os novos valores no contentor de dados e liberta todos os trincos adquiridos. Se a transacção não puder ser confirmada, um comando de cancelamento (`RollbackCommand`) é criado e enviado. Este comando é também interceptado no `LockingInterceptor` que somente liberta os trincos. Em ambos os casos, a memória ocupada pelo contexto local da transacção é libertada.



(a) Modo Replicado

(b) Modo Distribuído

Figura 3.3: Esquema do protocolo em duas fases no *Infinispan*

3.4.2 Concretização dos Algoritmos

Nesta secção descrevem-se com algum grau de pormenor as modificações efectuadas no *Infinispan* para a concretização do protocolo baseado em difusão atómica. Para tal foram criados vários componentes que são descritos de seguida e têm como objectivo oferecer as funcionalidades extras necessárias.

3.4.2.1 Solução para Replicação Total

O primeiro componente a ser concretizado foi o `TotalOrderPrepareCommand` que estende o `PrepareCommand`. Este comando é usado na primeira fase do 2PC onde é enviado o identificador global da transacção e as modificações efectuadas pela transacção. A justificação da escolha de criar um novo comando em vez de utilizar o existente foi pelo facto de este ter um funcionamento diferente e assim evitar a utilização excessiva de condições para separar as funções de ambos os comandos. Este comando será usado tanto para a replicação total como para a replicação parcial.

Foi desenvolvido um novo módulo para a gestão da confirmação das transacções recorrendo à difusão atómica. Esse módulo denomina-se `TotalOrderTransactionManager`. É neste módulo que se encontra concretizada a verificação do *write skew*. O módulo recebe como parâmetro o mapa com as chaves e os valores necessários para a verificação e, para todas as chaves do mapa, verifica se o valor lido é o mais actual. Este módulo é também responsável por assegurar a sincronização entre diferentes fios de execução.

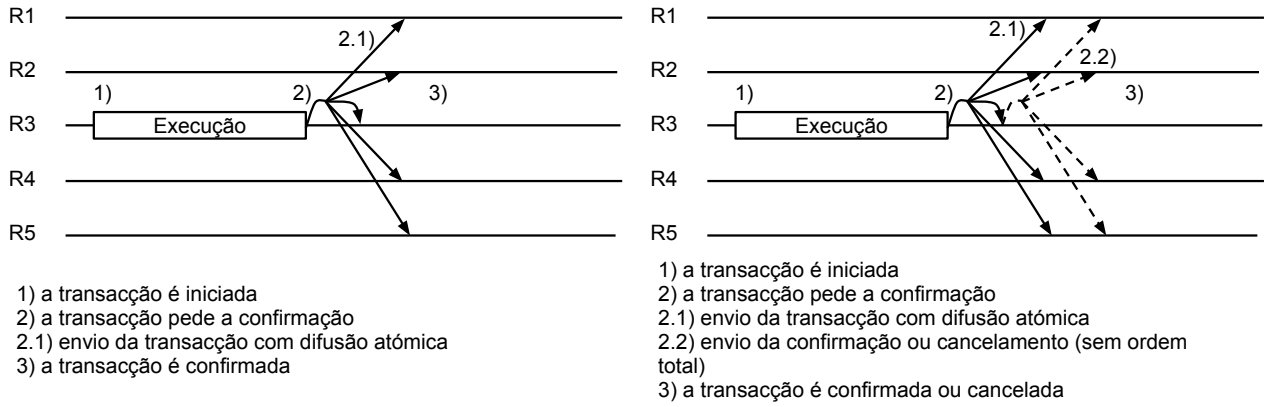
O `TotalOrderInterceptor` encontra-se no topo da cadeia de interceptores e processa os comandos relevantes para a confirmação de uma transacção, o `TotalOrderPrepareCommand` (primeira fase do 2PC), o `CommitCommand` e o `RollbackCommand` (segunda fase do 2PC). Este interceptor certifica as transacções e bloqueia o processamento dos comandos da segunda fase de confirmação, se estes forem recebidos antes da entrega do `TotalOrderPrepareCommand`. Isto acontece sempre que existem atrasos no processamento das mensagens em ordem total, devido a sobrecarga, uma vez que os comandos da segunda fase são entregues sem qualquer ordem definida, e podem portanto ultrapassar as mensagens que estão na fila de ordenação.

Ambos os interceptores referidos na arquitectura do *Infinispan* foram modificados para suportar esta solução. Nomeadamente, o `TxInterceptor` e o `ReplicationInterceptor` passaram a interceptar o `TotalOrderPrepareCommand`. O primeiro, prepara a transacção, tal como faz para o 2PC e o segundo trata de difundir em ordem total o comando. A difusão é feita de modo síncrono onde espera a confirmação da recepção e processamento do comando por parte da réplica emissora. As classes que interagem com o *JGroups* foram modificadas de modo a suportar o envio e recepção em ordem total.

A entrega das transacções é feita por um único fio de execução em ordem total. No modelo de coerência RR+WS, este fio de execução fica bloqueado até que o comando da segunda fase seja entregue. Uma vez que esta concretização seria a segunda fase de todas as transacções, mesmo as que não concorrem no mesmo acesso a dados, desenvolveu-se também uma solução alternativa, que pretende suportar a certificação de transacções em paralelo.

No entanto, é necessário garantir que as transacções que concorrem no mesmo acesso a dados sejam certificadas pela ordem de entrega. Para garantir essa ordem, foi adicionado ao `TotalOrderInterceptor` um mapa que faz uma projecção entre chaves e transacções. Para cada chave indica a última transacção que irá modificá-la.

Quando uma transacção, digamos T, é entregue numa réplica, T consulta o mapa para conhecer que transacções que a antecedem. O passo seguinte é actualizar o mapa, onde as chaves modificadas por T são associadas a T, pois esta será a última transacção a modificá-las. Finalmente T pode ser despachada para o conjunto de fios de execução de certificação. No entanto, só será certificada assim que as transacções antecedentes tenham concluído a sua certificação. Finalmente, T notifica as transacções precedentes quando termina a sua certificação, cancelando ou confirmando.



(a) Esquema **sem** verificação do *write skew*

(b) Esquema **com** verificação do *write skew*

Figura 3.4: Esquema do protocolo baseado em difusão atómica para replicação total

Na Figura 3.4 apresenta-se um esquema do protocolo descrito nesta secção. Tal como se pode verificar, se a transacção não necessitar de fazer a verificação do *write skew*, esta é confirmada num passo de comunicação. Por outro lado, se a verificação for necessária, é preciso executar mais um passo de comunicação.

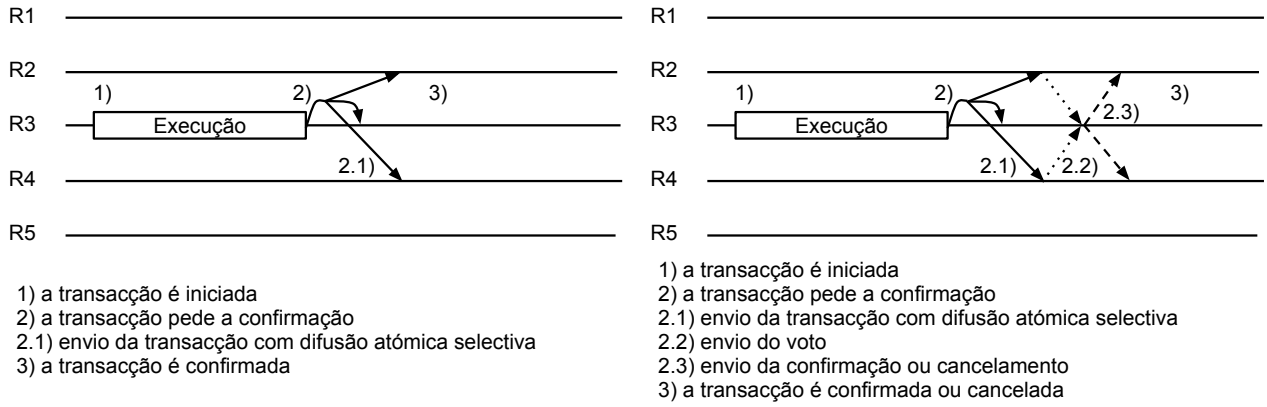
A concretização da solução apresentada nos parágrafos anteriores é o resultado de vários refinamentos a versões preliminares que foram sendo testadas e melhoradas. Esta será avaliada no capítulo seguinte.

3.4.2.2 Solução para Replicação Parcial

No `TotalOrderPrepareCommand` foi adicionada informação sobre o RS, que é necessária para a verificação do *write skew*. Esta informação é armazenada num mapa que associa uma chave ao valor lido. Para verificar se a verificação *write skew* é necessária, basta aferir se o mapa está vazio ou não. Caso se encontre vazio, então a verificação não é necessária.

Um novo comando foi criado para enviar os votos, o `VoteCommand`. Este comando contém informação sobre o resultado da verificação do *write skew*, assim como quais as chaves que foram validadas pela réplica que enviou o comando. Por questões de optimização, uma transacção pode ser confirmada se receber pelo menos um voto positivo em todas as chaves. No entanto a recepção de um voto negativo leva de imediato ao cancelamento da transacção. O `TotalOrderTransactionManager` foi modificado para suportar a troca de votos.

O `TotalOrderInterceptor` foi estendido para dar lugar ao `DistTotalOrderInterceptor`,



(a) Esquema **sem** verificação *write skew*

(b) Esquema **com** verificação *write skew*

Figura 3.5: Esquema do protocolo baseado em difusão atómica selectiva para replicação parcial

com duas diferenças principais. A primeira, é a intercepção do comando `VoteCommand`, que retira o resultado do voto e adiciona-o ao `TotalOrderTransactionManager`; a segunda diferença consiste na responsabilidade de enviar de volta o voto da verificação do *write skew*. Obviamente, só é necessário enviar o voto se a réplica em questão possui alguma das chaves acedidas pela transacção.

A Figura 3.5 ilustra uma execução do protocolo descrito nesta secção. Tal como se pode verificar, se a transacção não necessitar de fazer a verificação *write skew*, esta é confirmada num passo de comunicação. Por outro lado, se a verificação for necessária, as réplicas acedidas necessitam de enviar os seus votos. Isto vai acrescentar mais passos de comunicação.

Tal como se consegue observar, depois de concretizada a versão para replicação total, suportar a replicação parcial é relativamente simples. De facto, os algoritmos só diferem na gestão da distribuição dos dados. Deste modo, ficam os donos da chave como responsáveis pela verificação do *write skew*, em vez da réplica que executou a transacção. Esta versão será avaliada no capítulo seguinte e esperamos obter um desempenho superior ao 2PC clássico.

3.5 Concretização no *JGroups*

O *JGroups* é um sistema de comunicação em grupo que suporta comunicação fiável entre várias máquinas, daqui em diante designadas por membros, desenvolvido em código aberto pela RedHat. As principais características focam-se na detecção de entrada e saída de membros do grupo, com notificação da aplicação de quando estas mudanças acontecem, detecção de falha

de membros e sua remoção do grupo, também com notificação, envio de mensagens ponto-a-multiponto e ponto-a-ponto de um modo fiável, ordem na entrega de mensagens (FIFO ou ordem total) e com suporte a vários protocolos de comunicação (por exemplo, UDP ou TPC). Uma característica importante do *JGroups* é suportar uma pilha flexível de protocolos. Cada protocolo define uma funcionalidade, por exemplo, fiabilidade ou ordenação de mensagens. Os programadores podem adaptar o *JGroups* activando ou desactivando protocolos, de modo a satisfazer os requisitos da aplicação.

O *Infinispan* utiliza o *JGroups* como ferramenta de difusão de mensagens entre réplicas. Além disso, tira vantagens do serviço de grupos disponibilizado, para distribuir as chaves entre as réplicas na replicação parcial. Como foi referido anteriormente, a solução para replicação parcial necessita de uma primitiva de difusão atómica selectiva. Esta primitiva não é suportada pelo *JGroups*. Desse modo, foi desenvolvido um protocolo para suportar essa funcionalidade, e a sua concretização encontra-se nesta secção, onde vamos inicialmente explicar como o *JGroups* está estruturado.

3.5.1 Arquitectura do *JGroups*

O *JGroups* encontra-se concretizado com base numa arquitectura baseada em eventos. Deste modo, os protocolos são concretizados por módulos que interceptam e processam os eventos de modo a oferecer uma funcionalidade. Os protocolos podem ser combinados, criando pilhas, onde cada protocolo pode ser adicionado ou removido. Os eventos podem descer ou subir nessa pilha de protocolos, consoante venham da aplicação ou da rede, respectivamente. No entanto, também é possível os próprios protocolos criarem eventos e enviá-los para cima e/ou para baixo. Os eventos que sobem eventualmente chegam à aplicação e os que descem chegam à rede para serem transmitidos. Para implementar a nossa solução, foi desenvolvido um protocolo para cada um dos algoritmos de difusão atómica selectiva.

Os eventos (`Event`) são caracterizados por possuírem um tipo e um argumento. Dos vários tipos existentes, o único relevante para este trabalho é o tipo mensagem (`Event.MSG`). As mensagens (`Message`) são constituídas pelo endereço de origem e destino (`Address`), por uma lista de cabeçalhos (`Header`), por *flags* e pelos dados a transmitir. Tal como o nome indica, os endereços identificam o membro de origem e destino da mensagem. Os cabeçalhos são constituídos por um identificador único e por vários argumentos, consoante o tipo do cabeçalho. Cada protocolo

pode ter vários cabeçalhos ou nenhum, mas numa mensagem só pode haver um cabeçalho de cada tipo. Os cabeçalhos são maioritariamente usados para transmitir a informação de controlo necessária para garantir o bom funcionamento dos protocolos. As *flags* são usadas para evitar que certas mensagens sejam interceptadas por determinados protocolos, nomeadamente, os protocolos de controlo de fluxo, de ordenação ou de fiabilidade. Por exemplo, se queremos que uma mensagem seja entregue sem ordem total, basta colocar a *flag* `Message.NO_TOTAL_ORDER` na mensagem. Finalmente, os dados a transmitir referem-se aos dados que a aplicação quer transmitir entre membros.

3.5.2 Concretização dos Algoritmos de Difusão Atómica Selectiva

Para a concretização dos algoritmos, quatro entidades foram adicionadas para suportar a funcionalidade desejada. Essas entidades são transversais a ambos os algoritmos e são o `MessageID`, o `GroupMulticastHeader`, o `DeliverManager` e o `GroupAddress`. Tal como o nome indica, o `MessageID` representa um identificador global e único que está associado a uma mensagem. Este é constituído pelo endereço IP e porto do membro e um contador. O endereço IP e porto identificam unicamente um processo no sistema e o contador identifica inequivocamente as mensagens originadas por esse processo. O identificador tem a vantagem de serem comparáveis entre si. Esta característica é fundamental para os casos onde as mensagens têm o mesmo número de ordem. Os números de ordem estão associados às mensagens e indicam qual a ordem por que as mensagens devem ser entregues, isto é, as mensagens com número de ordem inferior devem ser entregues antes das mensagens com número de ordem superior.

O `GroupMulticastHeader` é um cabeçalho criado para identificar uma mensagem a ser processada pelos algoritmos desenvolvidos, com o objectivo de chegar a acordo da ordem de entrega da mensagem. É constituído por um tipo, um endereço do membro de origem que quer difundir a mensagem, um identificador, um grupo de destino e um número de ordem. O tipo da mensagem depende do algoritmo e o grupo de destino é um conjunto de endereços não repetidos (`java.util.Set`). Finalmente, o `GroupAddress` é um tipo de endereço para diferenciar as mensagens. Só as mensagens que tenham como endereço de destino este tipo são processadas pelos algoritmos.

O `DeliverManager` tem a funcionalidade de manter as mensagens ordenadas globalmente. Este é constituído por um mapa, que associa um identificador e os respectivo número de

ordem a um booleano. O booleano indica se a mensagem está pendente ou final, consoante o seu valor seja falso ou verdadeiro, respectivamente. O mapa utilizado é do tipo `java.util.concurrent.ConcurrentSkipListMap` pela sua característica de manter as chaves ordenadas por um comparador. O comparador desenvolvido, ordena inicialmente as mensagens pelo número de ordem e, no caso de números de ordem iguais, o desempate é efectuado pelo identificador da mensagem.

Finalmente, falta referir que os algoritmos têm um fio de execução particular para fazer a entrega ordenadas das mensagens. Quando notificado, este interage com o `DeliverManager` requisitando a próxima mensagem a ser entregue. O `DeliverManager` verifica o estado da mensagem que se encontra à cabeça do mapa. Se esta estiver no estado final, então é devolvida ao fio de execução, que faz a sua entrega à aplicação. Caso contrário, devolve `null` e o fio de execução fica bloqueado até futura notificação dos algoritmos. Este comportamento é transversal a ambos os algoritmos.

Difusão Atómica Selectiva (latência de três passos de comunicação) Neste algoritmo são usados dois conjuntos distintos: o conjunto de envio, que contém o estado das mensagens enviadas e o conjunto de recepção, que contém o estado das mensagens recebidas. Ambos os conjuntos são constituído por um mapa que associa um identificador ao estado correspondente. Para uma mensagem recebida, é mantido os dados a transmitir, o número de ordem e a indicação se a mensagem é final ou pendente. Para uma mensagem enviada, é mantido o conjunto de destino, o maior número de ordem recebido e um conjunto de membros com a proposta do número de ordem em falta.

Como descrito, o cabeçalho possui um campo para o tipo. Neste algoritmo, esse campo pode ter um dos seguintes tipos: (i) `MESSAGE`, (ii) `MESSAGE_PROPOSE` ou (iii) `MESSAGE_FINAL`. O tipo (i) identifica a mensagem como uma mensagem de dados e corresponde ao primeiro passo do algoritmo. O tipo (ii) identifica a mensagem como uma proposta do número de ordem. Finalmente, o tipo (iii) identifica a mensagem como sendo o envio do número de ordem final da mensagem. Só as mensagens do tipo (i) contêm os dados a transmitir da aplicação. Para não consumir tráfego na rede, as outras mensagens só contém o cabeçalho com a informação estritamente necessária.

Em maior detalhe, para enviar uma mensagem, o emissor deve criar uma mensagem e

colocar no destino um endereço do tipo `GroupAddress` e desactivar a *flag* `NO_TOTAL_ORDER`. Um evento com a mensagem é criado e enviado para a pilha de protocolos. Em cada protocolo é invocado o método `down(Event)`. No nosso protocolo, só são processados todos os eventos do tipo `Event.MSG`, a mensagem tem de ter um endereço do tipo `GroupAddress` no campo do destino e a *flag* `NO_TOTAL_ORDER` desactivada. Qualquer outro caso será ignorado. Desse modo, quando uma mensagem chega, é criado um identificador único para a mensagem e é adicionada ao conjunto de envio. Caso o emissor pertença ao conjunto de destino, então a mensagem é adicionada ao conjunto de recepção, o relógio lógico é incrementado e o seu valor é adicionado ao estado da mensagem no conjunto de envio. Além disso, o `DeliverManager` é notificado da criação da mensagem e do seu número de ordem temporário. A mensagem é difundida enviando N mensagens ponto-a-ponto, em que N é o número de membros de destino.

Quando é recebido algum evento da rede, este é enviado para a pilha de protocolos invocando o método `up(Event)`. No nosso protocolo, só são processados eventos do tipo `Event.MSG` e para os quais a mensagem possua um cabeçalho do tipo `GroupMulticastHeader`. Qualquer outro caso é ignorado. Desse modo, o processamento da mensagem é efectuado consoante o tipo do cabeçalho:

- `MESSAGE` – o relógio lógico é incrementado e o seu valor associado à mensagem (número de ordem). A mensagem é adicionada ao conjunto de recepção e o `DeliverManager` é notificado da presença da mensagem. Finalmente, uma mensagem do tipo `MESSAGE_PROPOSE` é criada para enviar o número de ordem associado à mensagem de volta ao emissor.

- `MESSAGE_PROPOSE` – o relógio lógico do receptor é actualizado se o seu valor for inferior ao valor proposto recebido. O número de ordem da mensagem é actualizado para o máximo entre o valor actual e o valor proposto. Finalmente verifica-se se a mensagem pode passar ao estado final. Isto só é possível depois de receber todas as propostas de todos os membros de destino. Quando for este o caso, cria-se uma mensagem do tipo `MESSAGE_FINAL` com o valor do número de ordem final e difunde-se a informação. Se o emissor pertence ao conjunto de destino, então notifica-se o `DeliverManager` do novo número de ordem e da mudança para o estado final e notifica-se o fio de execução de entrega.

- `MESSAGE_FINAL` – o relógio lógico é actualizado se o seu valor for inferior ao valor recebido. O `DeliverManager` é notificado do novo número de ordem e da mudança para o estado

final. Finalmente o fio de execução de entrega é notificado.

Finalmente, falta referir que, quando se envia a mensagem com o número de ordem final, o estado da mensagem é removido do conjunto de envio, libertando a memória ocupada. O estado da mensagem é removido do conjunto de recepção quando a mensagem é entregue à aplicação. A entrega é efectuada conforme o descrito anteriormente.

Difusão Atómica Selectiva (latência de dois passos de comunicação) Este algoritmo recorre a uma única estrutura de dados, designada conjunto de recepção, que é um mapa que associa um identificador a um estado. A informação mantida no estado é um grupo de destino, um conjunto de membros com a proposta do número de ordem em falta, os dados a transmitir e o número de ordem. Uma mensagem é considerada final quando possuir os dados a transmitir e recebeu todas as propostas previstas.

Este algoritmo utiliza três tipos de cabeçalhos: (i) `MESSAGE`, (ii) `MESSAGE_WITH_PROPOSE` e (iii) `PROPOSE`. O emissor escolhe um dos dois primeiros tipos para enviar a mensagem e o terceiro é usado na fase seguinte pelos restantes membros. Uma mensagem do tipo (i) contém os dados a transmitir, do tipo (ii) contém os dados a transmitir e o valor do número de ordem proposto e do tipo (iii) contém só o valor do número de ordem proposto.

Quando se difunde uma mensagem, é criado um identificador, o qual fica associado à mensagem, e o emissor verifica se pertence ao grupo de destino. Em caso afirmativo, o relógio lógico é incrementado e o seu valor é associado à mensagem (número de ordem). O `DeliverManager` é notificado da presença da mensagem e a mensagem é adicionada ao conjunto de recepção. De seguida envia-se os dados a transmitir e o número de ordem para os membros do grupo de destino numa mensagem do tipo (ii). Se o emissor não pertence ao grupo de destino, então envia somente os dados a transmitir para os membros do grupo de destino numa mensagem do tipo (i).

Na recepção de uma mensagem da rede, verifica-se através do identificador, se o estado da mensagem existe no conjunto de recepção. Se não existir, então adiciona-se o estado da mensagem e assinala-se que é necessário enviar a proposta do número de ordem para os restantes membros. Se a mensagem contiver os dados a transmitir (tipo (i) ou (ii)), então estes são adicionados ao estado da mensagem. Se a mensagem contiver uma proposta do número de

ordem (tipo (ii) ou (iii)), então o relógio lógico é actualizado, se o seu valor for inferior, o número de ordem da mensagem é actualizado para o máximo entre o valor actual e o valor recebido da proposta. O `DeliverManager` é notificado nesta altura. O membro que enviou a proposta é removido do conjunto de membros com propostas em falta.

Caso seja necessário enviar a proposta para os restante membros, isso é efectuado nesta altura. O valor da proposta corresponde ao máximo entre o valor no estado da mensagem e o valor do relógio lógico. O `DeliverManager` é notificado da alteração do valor e o valor é enviado numa mensagem do tipo (iii). O valor do relógio lógico fica sempre com o incremento do valor da proposta. Para concluir, verifica-se se a mensagem se encontra no estado final. Em caso afirmativo, o `DeliverManager` é notificado, assim como o fio de execução de entrega. Quando uma mensagem é entregue, o estado é removido do conjunto de recepção, libertando a memória ocupada.

4 Avaliação

Este capítulo mostra os resultados obtidos através da avaliação experimental dos algoritmos apresentados, com o objectivo de aferir e comparar o desempenho das soluções apresentadas no *Infinispan* e com diferentes configurações. A Secção 4.1 e a Secção 4.2 descrevem a configuração do sistema e os critérios de avaliação utilizados para a avaliação dos protocolos. A Secção 4.3 apresenta os resultados obtidos com o *Radargun Benchmark* e a Secção 4.4 apresenta os resultados obtidos com o TPC-C *Benchmark*. Finalmente, a Secção 4.5 apresenta uma breve análise dos resultados obtidos e as conclusão desta avaliação.

4.1 Configuração do Sistema

Todas as experiências aqui descritas foram efectuadas num *cluster* de 10 máquinas, onde cada máquina está equipada com dois processadores *Intel(R) Xeon(R) E5506* @ 2.13GHz e com 16GB de memória RAM. O sistema operativo instalado é o *Linux 2.6.32-33-server* e as máquinas encontram-se interligadas através de uma rede *Ethernet* privada com um débito de 1 *Gigabit* por segundo.

Como supramencionado, os protocolos avaliados foram concretizados no *Infinispan*, que utiliza o *JGroups* como mecanismo de comunicação em grupo. As principais configurações usadas no *JGroups* para reduzir a carga da rede são o uso de *IP Multicast* para o envio de mensagens para todas as máquinas do *cluster* e agrupamento de mensagens pequenas em mensagens de maior dimensão. Além disso, o detector de falhas foi desactivado porque as experiências foram efectuadas em ambiente estável. A primitiva ABcast implementada no *JGroups* é baseada em sequenciador. A mensagem é enviada para o sequenciador que ordena as mensagens e as envia de volta com a sua ordem correcta. Em relação à primitiva AMcast, esta teve de ser concretizadas de raiz e são apresentados os resultados com ambas as concretizações descritas anteriormente.

4.2 Critérios de Avaliação

De modo a avaliar e a comparar as soluções desenvolvidas com o 2PC foram escolhidos três critérios de avaliação, que são os seguintes:

- *Débito*: consiste no número de transacções confirmadas no sistema por segundo. Para dois protocolos, p_1 e p_2, considera-se que p_1 tem um desempenho superior a p_2, se o débito de p_1 é superior ao débito de p_2.

- *Taxa de cancelamentos*: consiste na percentagem do número de transacções canceladas em comparação com o número total de transacções executadas. As transacções podem ser canceladas por: violar o critério de coerência, não conseguir adquirir os trincos ou porque ocorreu uma situação de interbloqueio com outra transacção. Para dois protocolos, p_1 e p_2, considera-se que p_1 é melhor do que p_2, se a taxa de cancelamento de p_1 é menor do que a taxa de cancelamento de p_2, para o mesmo modelo de coerência.

- *Duração média da fase de confirmação*: consiste na duração média desde a invocação do método `commit()` até a transacção ser confirmada ou cancelada. Para dois protocolos, p_1 e p_2, considera-se que p_1 é melhor do que p_2, se a duração média da fase de confirmação de p_1 é menor do que a duração média da fase de confirmação de p_2.

4.3 *Radargun Benchmark*

4.3.1 Descrição

O *Radargun* é uma aplicação para avaliação do desempenho criada pela RedHat especificamente para este tipo de caches. Esta aplicação permite comparar o desempenho de várias caches distribuídas tais como: *Infinispan*, *EhCache*[1], *Oracle Coherence*[2], em diferentes cenários. Ao impor uma carga bastante elevada sobre os diferentes nós do sistema, permite-nos aferir qual o débito máximo de cada configuração/sistema.

[1] http://ehcache.org/documentation/overview.html
[2] http://coherence.oracle.com/display/COH/Oracle+Coherence+Knowledge+Base+Home

Parâmetro	Valor
(1) duração da simulação (minutos)	5
(2) n.º de fios de execução	8
(3) op. de escrita (%)	10
(4) n.º de operações	10
(6) detecção de interbloqueios	activada
(9) tempo máximo para aquisição de trincos (segundos)	10

Tabela 4.1: Configurações comuns a usar com o *Radargun*

O modo de funcionamento desta aplicação foi alterado de modo a que cada fio de execução execute o maior número de transacções possíveis durante um período de tempo estipulado na configuração da mesma.

De modo a avaliar cada protocolo, foram efectuadas várias avaliações com diferentes configurações. Os possíveis parâmetros de configuração para o *Radargun* são os seguintes: (1) duração da simulação, (2) número de fios de execução para executar transacções em cada réplica, (3) percentagem de operações de escrita numa transacção, (4) número de operações numa transacção e finalmente, (5) número total de itens sob os quais as transacções operam. Os primeiros quatro parâmetros são comuns para qualquer execução com esta aplicação e os seus valores podem ser consultados na Tabela 4.1.

Para configurar o modo de funcionamento do *Infinispan* temos à disposição os seguintes parâmetros: (6) mecanismo de detecção de interbloqueios, (7) critério de coerência, (8) verificação do *write skew*, (9) tempo máximo para aquisição de trincos e (10) o número de cópias de cada chave, no caso da replicação parcial.

A Tabela 4.2 e a Tabela 4.3 apresentam um sumário das configurações usadas em cada um dos testes, para replicação total e parcial, respectivamente. A coluna *Identificador* é um nome único que será atribuído ao gráfico correspondente, com o objectivo de facilitar a associação dos resultados à configuração usada na sua obtenção.

4.3.2 Resultados

Nesta secção apresentam-se os resultados obtidos com esta ferramenta. No final, é efectuada uma breve análise aos resultados obtidos com a nossa solução e os resultados obtidos com a solução nativa, o 2PC.

Identificador	**(5) n.º de itens**	**(7) critério de coerência**	**(8) verificação do** *write skew*
c-radar-1	1.000	Read Commited	N/A
c-radar-2		Repeatable Read	Activada
c-radar-3	100.000	Read Commited	N/A
c-radar-4		Repeatable Read	Activada

Tabela 4.2: Diferentes configurações a usar com o *Radargun* para replicação total

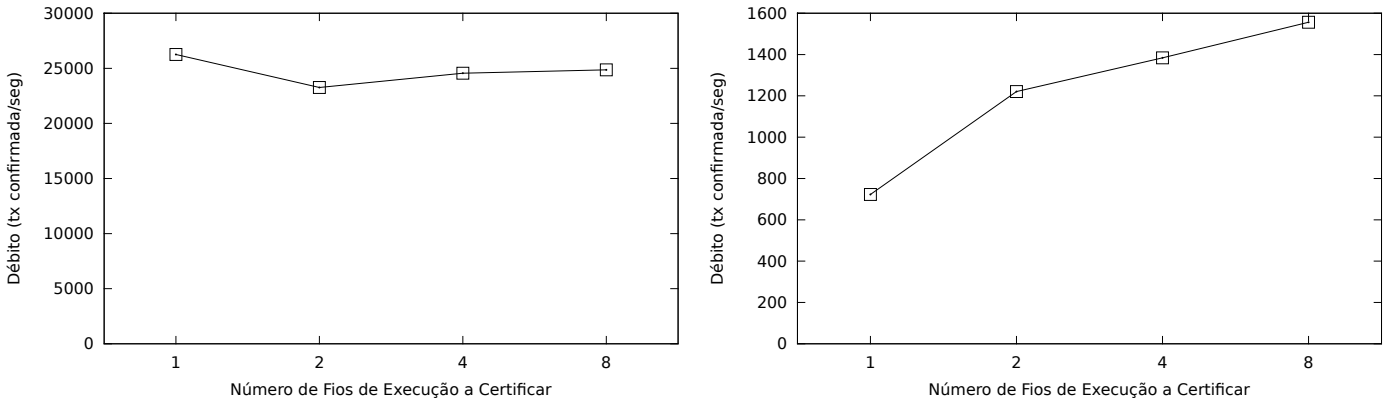
Identificador	**(5) n.º de itens**	**(7) critério de coerência**	**(8) verificação do** *write skew*	**(10) n.º de cópias por chave**
c-radar-5	1.000	Read Commited	N/A	2
c-radar-6		Repeatable Read	Activada	
c-radar-7	100.000	Read Commited	N/A	
c-radar-8		Repeatable Read	Activada	
c-radar-9	1.000	Read Commited	N/A	4
c-radar-10		Repeatable Read	Activada	
c-radar-11	100.000	Read Commited	N/A	
c-radar-12		Repeatable Read	Activada	

Tabela 4.3: Diferentes configurações a usar com o *Radargun* para replicação parcial (distribuição) com o protocolo de AMcast em 2 e 3 passos de comunicação

4.3.2.1 Certificação com Múltiplos Fios de Execução

No capítulo anterior estabelecemos a hipótese de que o uso de um único fio de execução para certificar transacções poderia ser um ponto de estrangulamento do sistema. Desse modo, desenvolvemos uma alternativa para certificar transacções sem conflitos de forma concorrente. A alternativa desenvolvida permite configurar o número de fios de execução dedicados à certificação das transacções. Para evitar avaliar todas as configurações acima referidas, variando o número de fios de execução, decidimos apresentar a avaliação da variação dos fios de execução exclusivamente nesta secção, usando um valor fixo na restante avaliação apresentada.

A Figura 4.1 ilustra o impacto que a variação do número de fios de execução de certificação tem no débito do sistema. Como se pode observar na Figura 4.1a, a variação do número de fios de execução não afecta o débito do sistema. Suspeitámos que este resultado se devesse ao facto do tempo de certificação de uma transacção ser muito baixo e o custo de manter a ordem de certificação entre transacções com conflitos não compensar o uso de múltiplos fios de execução para a certificação.



(a) Débito (*Radargun* normal)

(b) Débito (*Radargun* modificado)

Figura 4.1: Impacto do uso de certificação concorrente no débito do sistema

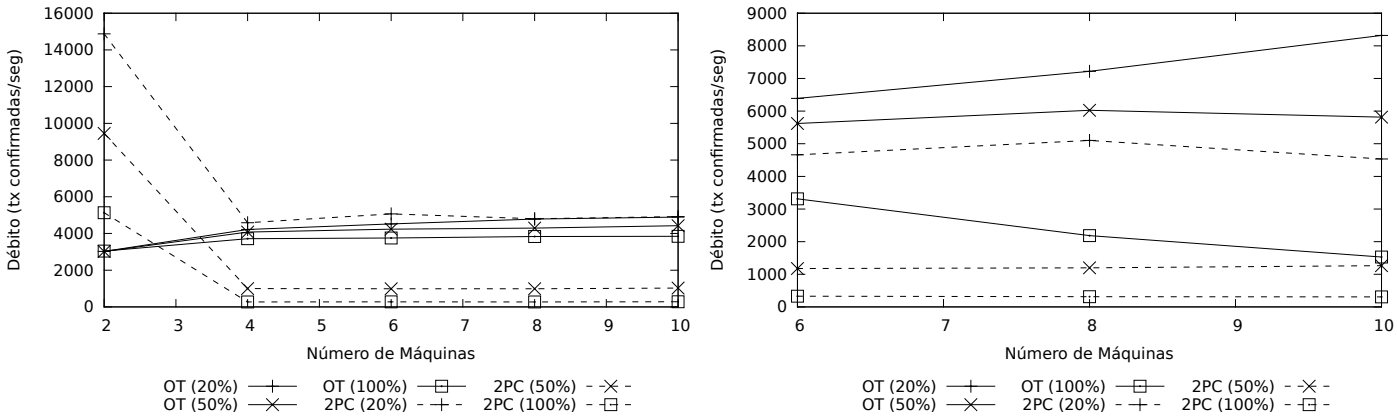
Para comprovar a hipótese anterior, foi efectuada um pequena alteração no *Radargun*, que consiste no seguinte: cada transacção lê um determinado conjunto de chaves e escreve de seguida nesse mesmo conjunto. Desse modo, conseguimos activar a verificação para o *write skew* para todas as chaves acedidas. Além disso, cada transacção acede a um conjunto de chaves completamente distinto das outras transacções, eliminando qualquer possibilidade de conflitos. A Figura 4.1b apresenta os resultados obtidos, onde cada transacção acede a um conjunto de chaves constituído por 12 chaves. Foram experimentados vários valores para o conjunto de chaves até saturar a rede.

Observando o resultado, podemos concluir que a certificação com múltiplos fios de execução pode não ser vantajoso neste contexto, pois o ponto de estrangulamento está na saturação da rede e não no processo de certificação, podendo ser vantajoso noutros sistemas, com uma certificação mais demorada.

4.3.2.2 Variação da Percentagem de Operações de Escritas

Apesar da taxa média de 10% de operações de escrita por transacção ser um valor realista, é seguramente interessante verificar o comportamento do sistema para uma taxa média de operações de escrita mais elevada. Com esse objectivo, foi efectuada uma avaliação com 20%, 50% e 100% de operações de escritas, num cenário de baixa contenção, pois queríamos eliminar a interferência dos interbloqueios nos resultados.

A Figura 4.2 ilustra o débito do sistema nas condições definidas anteriormente. Tal como



(a) Débito para replicação total

(b) Débito para replicação parcial

Figura 4.2: Impacto da variação da percentagem de operações de escritas no débito do sistema

se pode observar, o débito de ambos os protocolos decresce com o aumento da taxa média de operações de escrita. Isto deve-se ao conjunto de escrita ser de maiores dimensões, o que tem um impacto directo no custo da comunicação. No entanto, pode observar-se que o decréscimo de débito é superior para o 2PC. De facto, além do aumento do custo da comunicação, o desempenho deste protocolo é afectado pelo aumento da probabilidade das transacções entrarem em interbloqueio, pois envolvem um maior número de chaves. O fraco desempenho da nossa solução em replicação total deve-se ao uso de um sequenciador para estabelecer a ordem total, pois este torna-se o ponto de estrangulamento do sistema.

4.3.2.3 Replicação Total

A Figura 4.3 mostra o desempenho obtido, para os três critérios de avaliação definidos anteriormente, em cenários de elevada e baixa contenção e critérios de coerência RC e RR+WS. Começando por analisar o débito do sistema, observa-se que a contenção nos dados afecta bastante o débito do 2PC. Verifica-se que o débito do 2PC atinge o seu máximo com duas réplicas. Isto é explicado pelo facto onde a probabilidade de ocorrerem interbloqueios entre transacções ser bastante reduzida no cenário com duas réplicas. Esta explicação é reforçada analisando a taxa de cancelamento e a duração da fase de confirmação, verifica-se que ambas são inferiores para este caso particular.

Para 4 a 10 réplicas, o débito mantém-se estável à medida que o número de réplicas aumenta. A taxa de interbloqueios aumenta, pois a probabilidade de interbloqueios entre transacções

aumenta com o número crescente de transacções criadas, a duração da fase de confirmação também aumenta, pela mesma razão. A estabilidade do débito neste contexto é explicada pelo equilíbrio entre o número de transacções canceladas e o número de transacções criadas.

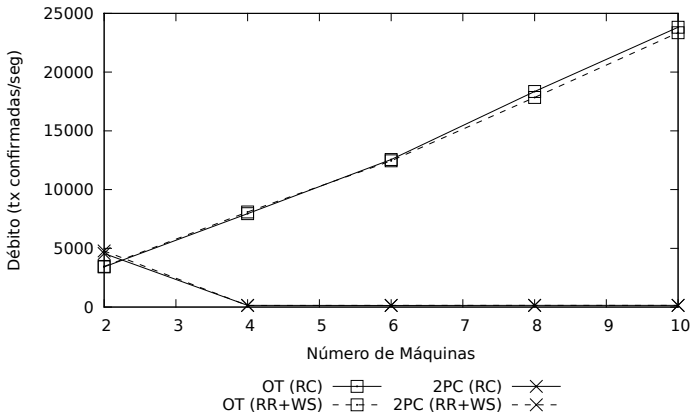
Analisando o débito da nossa solução, verifica-se que cresce com o aumento do número de réplicas. Este comportamento pode parecer estranho, no entanto, a explicação é bastante simples. Isto deve-se ao facto do *JGroups* utilizar um protocolo de ordenação baseado num sequenciador. O sequenciador recebe as mensagens das outras réplicas e atribui-lhes um número de ordem. Este comportamento beneficia a réplica com o sequenciador que consegue ordenar as suas transacções sem ser necessário ir à rede. De facto, nota-se que quase 50% do débito do sistema é originado pelo sequenciador.

Na descrição da nossa solução afirma-se que a nossa solução é livre de interbloqueios. Tal como se pode reparar, a taxa de cancelamento é virtualmente inexistente na nossa solução. Verificámos que não existem cancelamentos durante a confirmação da transacção. No entanto, o modo como as transacções são executadas não sofreu alterações e pode originar cancelamentos nesta fase. O nosso objectivo é suportar um novo algoritmo de certificação, e não modificar todo o controlo de concorrência existente e criar um novo sistema. Desse modo, os trincos ainda são adquiridos durante a execução da transacção existindo a possibilidade de interbloqueios entre transacções em execução.

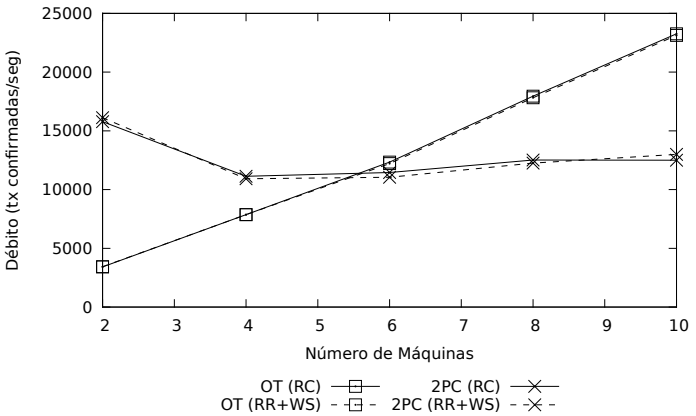
Ao analisarmos a taxa de cancelamento espera-se um aumento desta com o critério de coerência RR+WS para ambos os algoritmos analisados. No entanto, tal não se verifica, pois o *Radargun* selecciona as chaves de forma aleatória, originando uma probabilidade muito baixa de ocorrer uma leitura seguida de uma escrita consecutiva. Desse modo, o mecanismo de *write skew* é activado com pouca frequência.

4.3.2.4 Replicação Parcial

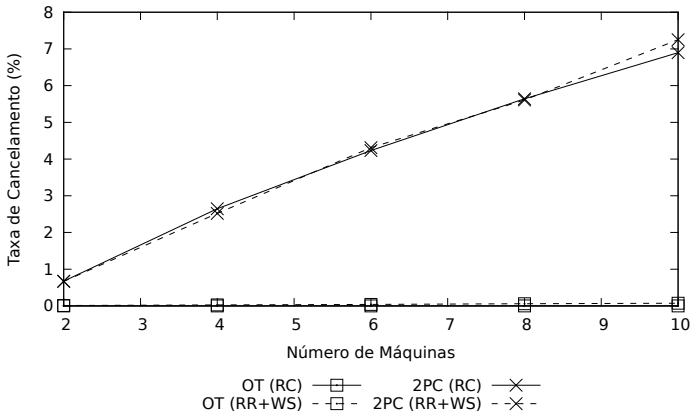
Para ajudar na interpretação dos resultados, esta secção encontra-se dividida em duas partes. Numa primeira parte, vamos analisar o resultados obtidos com grau de replicação 2, isto é, quando existem duas cópias de cada chave. Numa segunda parte, vamos analisar os resultados com grau de replicação 4.



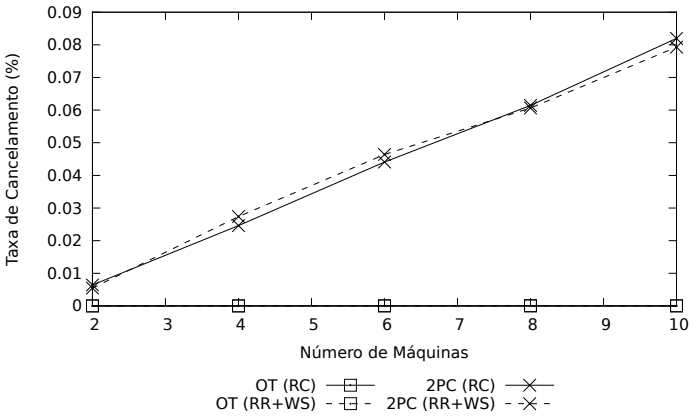
(a) Débito (*c-radar-1* e *c-radar-2*)



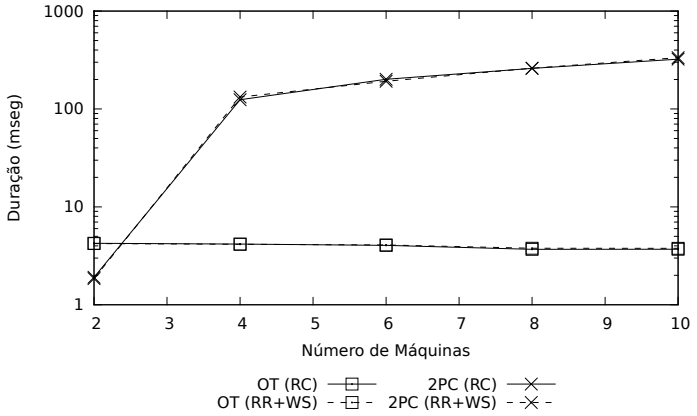
(b) Débito (*c-radar-3* e *c-radar-4*)



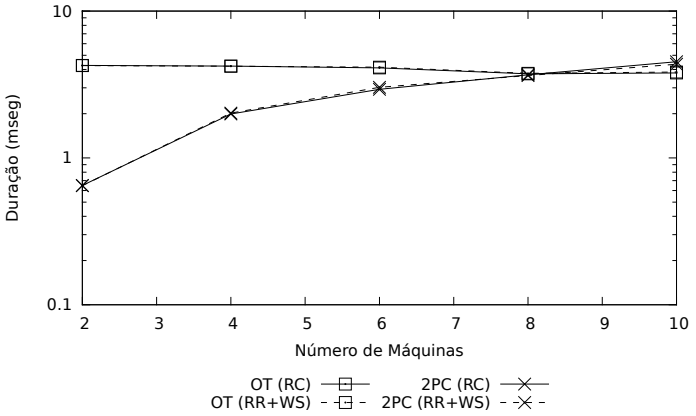
(c) Taxa de Cancelamento (*c-radar-1* e *c-radar-2*)



(d) Taxa de Cancelamento (*c-radar-3* e *c-radar-4*)



(e) Tempo de Confirmação médio (*c-radar-1* e *c-radar-2*)



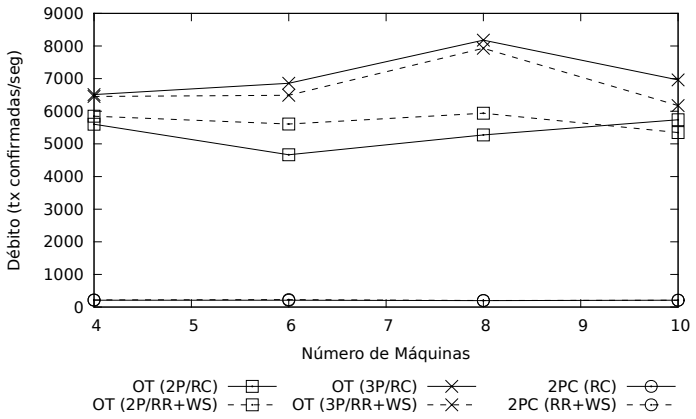
(f) Tempo de Confirmação médio (*c-radar-3* e *c-radar-4*)

Figura 4.3: Resultados para Replicação Total. À esquerda num cenário de **elevada** contenção e à direita num cenário de **baixa** contenção

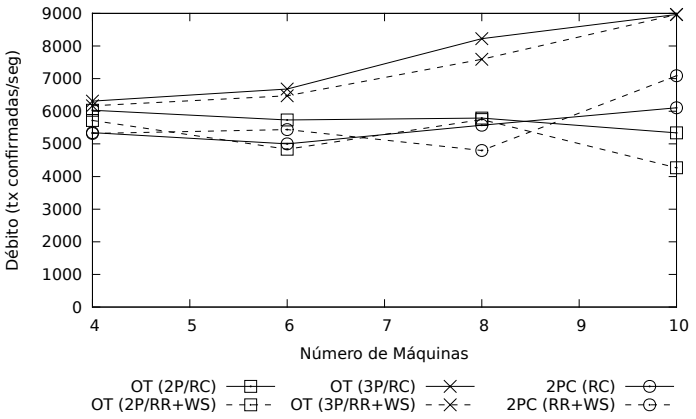
Grau de Replicação 2 A Figura 4.4 ilustra os resultados obtidos com replicação parcial num cenário de baixa contenção e num cenário de elevada contenção, com grau de replicação 2. Comparando ambos os cenários de contenção, na nossa solução nota-se uma variação quase insignificante em ambos os critérios de avaliação, tal como seria de esperar pois, ao eliminar os interbloqueios, estes não existem em ambos os cenários. O impacto da verificação do *write skew* é insignificante pela razão apresentada para a replicação total. Por outro lado, para o 2PC chega-se à mesma conclusão descrita acima e verifica-se que os interbloqueios possuem um grande impacto no sistema, tal pode ser comprovado por uma taxa de cancelamentos e por uma duração média da fase de confirmação mais elevada, no cenário de elevada contenção.

Efectuando uma análise entre as primitivas de AMcast, verifica-se que a primitiva com um menor número de passos de comunicação obtém um desempenho inferior. Isto deve-se ao número quadrático de mensagens que é gerado. Deste modo, o *JGroups* tem de garantir fiabilidade para um maior número de mensagens, o que aumenta a probabilidade de ocorrer conflitos entre os tramas UDP na rede, originando mais retransmissões das mensagens. Sendo assim, podemos concluir que neste contexto é melhor ter um passo de comunicação adicional do que um maior número de mensagens trocadas.

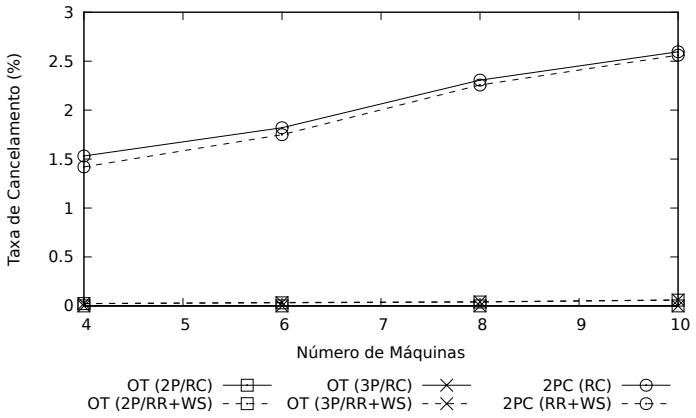
Grau de Replicação 4 A utilização de um grau de replicação 4 possui um lado positivo e um lado negativo. Por um lado, um grau de replicação mais elevado diminui a probabilidade de ser necessário contactar uma réplica remota para ler uma chave, diminuindo a frequência de leituras remotas, originando uma execução das transacções mais rápida. Por outro lado, na confirmação é necessário contactar um maior número de réplicas, originando um custo adicional nesta fase. A Figura 4.5 mostra os resultados obtidos para replicação parcial, com o grau de replicação 4. Através da análise do débito do sistema, podemos verificar que uma execução mais rápida compensa o custo da comunicação para a nossa solução com a primitiva de AMcast com 3 passos de comunicação e para o 2PC. No entanto, com a primitiva de AMcast com 2 passos de comunicação obtém um débito inferior. Isto é explicado pelo aumento quadrático do número de mensagens, o que torna a rede no ponto de estrangulamento do sistema. De facto, nota-se a diminuição do débito do sistema com o aumento do número de máquinas. Tal como concluímos anteriormente, nota-se um impacto bastante negativo pela ocorrência de interbloqueios no 2PC. Isso é provado pelo débito do sistema e pela taxa de cancelamentos.



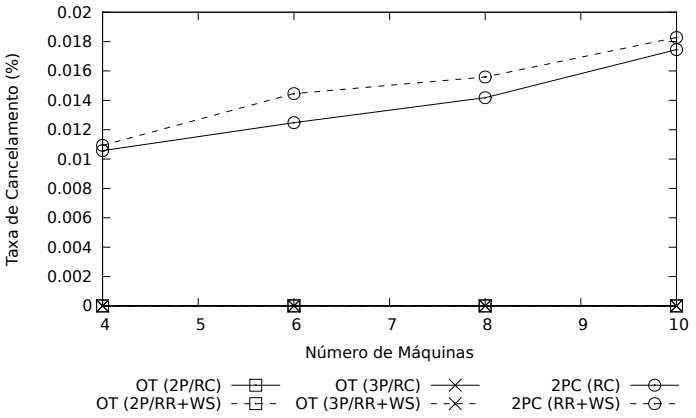
(a) Débito (*c-radar-5* e *c-radar-6*)



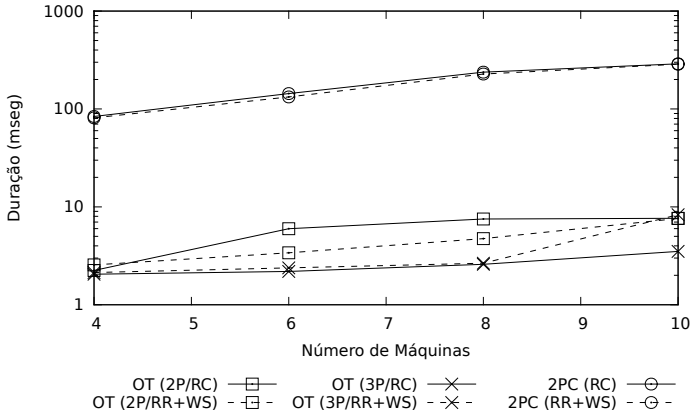
(b) Débito (*c-radar-7* e *c-radar-8*)



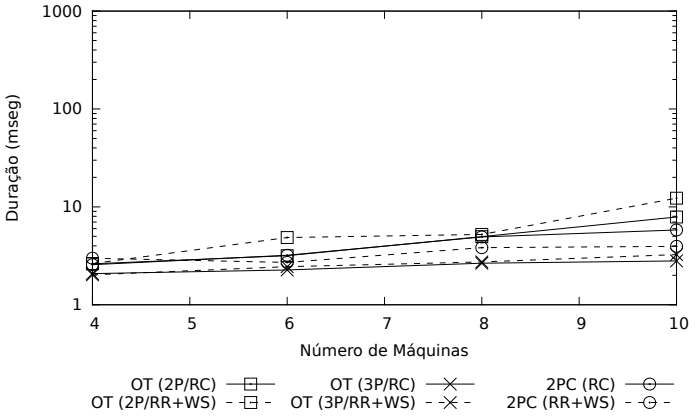
(c) Taxa de Cancelamento (*c-radar-5* e *c-radar-6*)



(d) Taxa de Cancelamento (*c-radar-7* e *c-radar-8*)

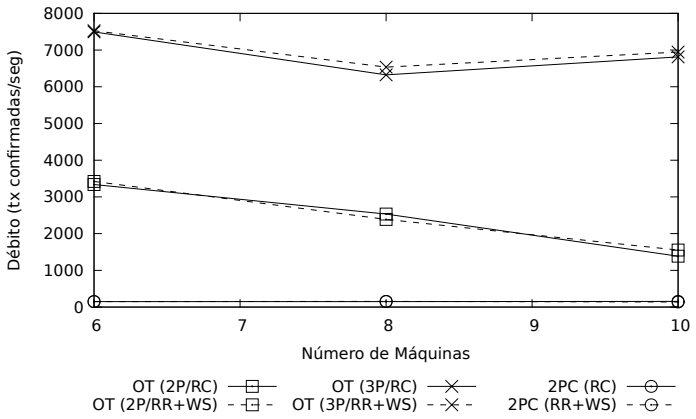


(e) Tempo de Confirmação médio (*c-radar-5* e *c-radar-6*)

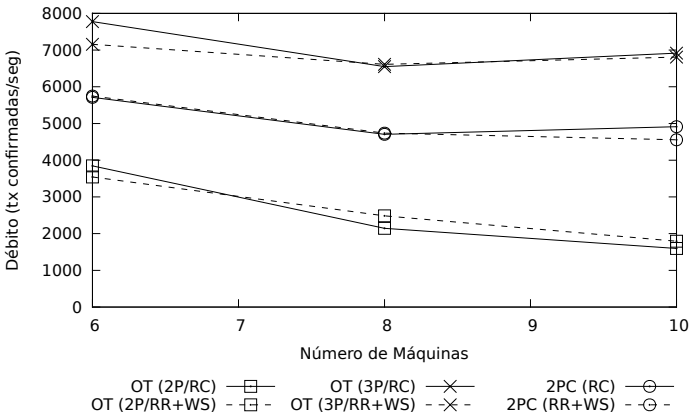


(f) Tempo de Confirmação médio (*c-radar-7* e *c-radar-8*)

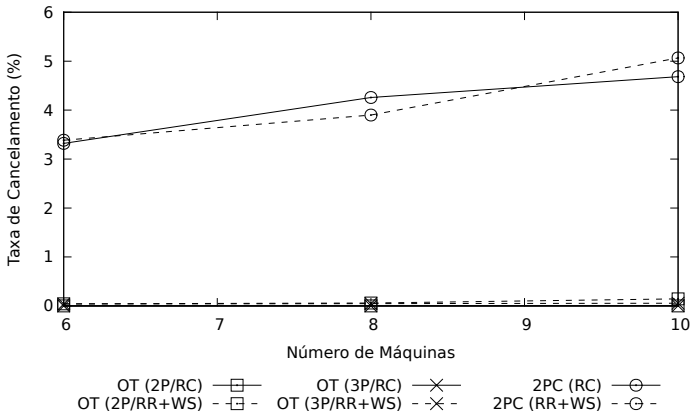
Figura 4.4: Resultados para Replicação Parcial com grau de replicação 2. À esquerda num cenário de **elevada** contenção e à direita num cenário de **baixa** contenção



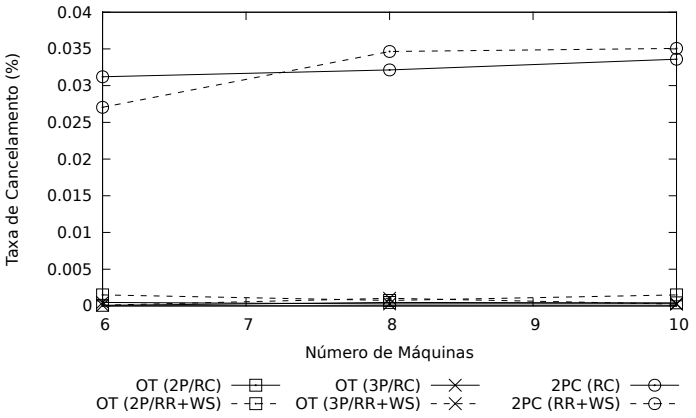
(a) Débito (*c-radar-9* e *c-radar-10*)



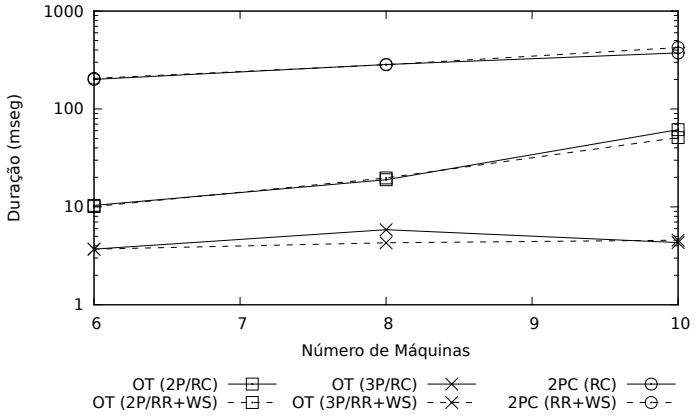
(b) Débito (*c-radar-11* e *c-radar-12*)



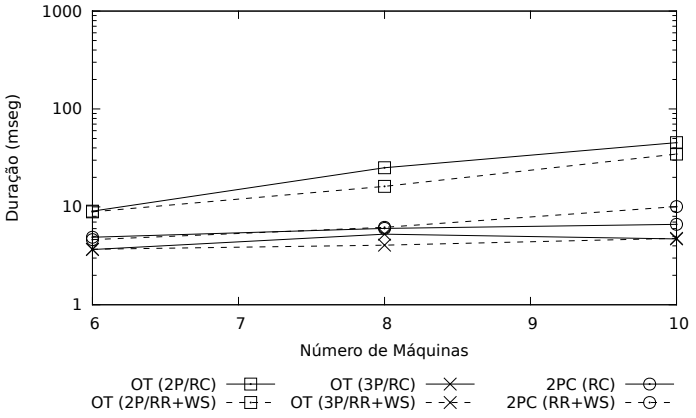
(c) Taxa de Cancelamento (*c-radar-9* e *c-radar-10*)



(d) Taxa de Cancelamento (*c-radar-11* e *c-radar-12*)



(e) Tempo de Confirmação médio (*c-radar-9* e *c-radar-10*)



(f) Tempo de Confirmação médio (*c-radar-11* e *c-radar-12*)

Figura 4.5: Resultados para Replicação Parcial com grau de replicação 4. À esquerda num cenário de **elevada** contenção e à direita num cenário de **baixa** contenção

4.4 TPC-C *Benchmark*

4.4.1 Descrição

O TPC-C[3] é um gerador de carga baseado no processamento *on-line* de transacções (OLTP, do Inglês *On-Line Transaction Processing*). Foi desenhado para simular um ambiente computacional completo onde uma população de utilizadores executam transacções sobre uma base de dados. A sua concretização foi efectuada com base na especificação do TPC versão C[4] no *Radargun*, definido anteriormente, e tem como objectivo executar o TPC-C usando o *Infinispan* como dispositivo de armazenamento possível.

O funcionamento é muito semelhante ao *Radargun*. A aplicação executa o maior número possível de transacções num período de tempo definido na configuração do teste. O TPC-C usa três tipos de transacções: *New-Order*, *Payment*, ambas transacções de leitura-escrita, e *Order-Status* (transacção de leitura), que são escolhidas com base em pesos (percentagens). O modelo de dados do TPC-C teve de ser adaptado ao modelo do *Infinispan*, que fornece um par chave/valor. Desse modo, cada tabela do TPC-C é projectada numa classe onde as colunas são representadas pelas variáveis de instância dessa classe. Cada classe tem uma chave, necessária para guardar o objecto no *Infinispan*, que é constituída por uma cadeia de caracteres. Esta é uma concatenação de cada coluna da chave primária da tabela. Qualquer classe tem um método para carregar e armazenar os dados no *Infinispan*, consoante a chave que possuir.

Entrando em mais pormenor sobre o TPC-C, este representa um negócio genérico de uma indústria que deve gerir, vender ou distribuir um serviço ou um produto. Deste modo, consegue-se obter uma diversidade de operações, tal como em aplicações reais. Sendo assim, simula uma empresa que possui um determinado número de lojas que, por sua vez, têm associado um armazém. À medida que a empresa se expande, novos armazéns e lojas são criados. Sendo assim, cada armazém fornece 10 distritos e cada distrito fornece um total de 3.000 clientes. Todos os armazéns mantém um stock de 100.000 itens, que são geridos pela empresa. Os clientes invocam a aplicação para encomendar novos itens ou consultar as suas encomendas e fazer os pagamentos das mesmas. Cada encomenda possui em média 10 itens.

Tal como o *Radargun*, o TPC-C possui parâmetros para configurar a sua execução. Os

[3] http://www.tpc.org/tpcc/
[4] http://www.tpc.org/tpcc/spec/tpcc_current.pdf

Parâmetro	**Valor**
(1) duração da simulação (minutos)	5
(2) n.º de fios de execução	8
(3) n.º de armazéns	1
(4) transacções de *Payment* (%)	45
(5) transacções de *Order-Status* (%)	5
(6) detecção de interbloqueios	activada
(9) tempo máximo para aquisição de trincos (segundos)	10

Tabela 4.4: Configurações comuns a usar com o TPC-C

Identificador	**(7) critério de coerência**	**(8) verificação do *write skew***
c-tpcc-1	Read Commited	N/A
c-tpcc-2	Repeatable Read	Activada

Tabela 4.5: Diferentes configurações a usar com o TPC-C para replicação total

parâmetros são os seguintes: (1) duração da simulação, (2) número de fios de execução para executar transacções, (3) número de armazéns, (4) percentagem de transacções de *Payment* e finalmente, (5) percentagem de transacções de *Order-Status*. A Tabela 4.4 apresenta os valores usados em todos os testes com esta aplicação. Estes valores simulam um pequeno negócio com cerca de 30.000 clientes. São efectuadas mais transacções correspondentes a encomendas e pagamentos do que transacções para obter a visualização do estado das encomendas. Não existem valores pré-definidos para fazer a avaliação com esta ferramenta, no entanto, os valores apresentados para os pesos das transacções são os mais frequentemente utilizados.

Os parâmetros para o *Infinispan* são os mesmos que foram definidos para o *Ragargun*: (6) detecção de interbloqueios, (7) critério de coerência, (8) verificação do *write skew*, (9) tempo máximo para aquisição de trincos e (10) o número de cópias de cada chave, somente para replicação parcial. A Tabela 4.5 e a Tabela 4.6 apresentam um sumário das configurações usadas. A coluna *Identificador* é atribuída aos gráficos dos resultados obtidos para identificar a configuração usada na obtenção dos resultados.

4.4.2 Resultados

Nesta secção apresentam-se os resultados obtidos. No final é efectuada uma breve análise comparativa entre os resultados obtidos com a nossa solução e os resultados obtidos com a

Identificador	**(7) critério de coerência**	**(8) verificação do** *write skew*	**(10) n.º de cópias por chave**
c-tpcc-3	Read Commited	N/A	2
c-tpcc-4	Repeatable Read	Activada	
c-tpcc-5	Read Commited	N/A	4
c-tpcc-6	Repeatable Read	Activada	

Tabela 4.6: Diferentes configurações a usar com o TPC-C para replicação parcial (distribuição) com o protocolo de AMcast em 2 e 3 passos de comunicação

solução nativa, o 2PC.

4.4.2.1 Replicação Total

A Figura 4.6 ilustra os resultados da avaliação efectuada no TPC-C com replicação total. A primeira observação a efectuar é o facto do 2PC obter um desempenho bastante baixo. Isto deve-se ao facto das transacções de escrita necessitarem de actualizar os dados de uma das 10 instâncias de *Distrito* existentes em todo o sistema, o que origina um elevado nível de contenção e daí o baixo desempenho. Isto é comprovado pela taxa de cancelamento, que atinge os 90% no pior caso, e pela duração da fase de confirmação, que atinge quase os 10 segundos, tempo limite para aquisição dos trincos.

A nossa solução consegue obter um desempenho superior devido à eliminação dos interbloqueios. De facto, com o nível de coerência RC, consegue-se um crescimento do débito do sistema devido à mesma razão indicada na análise dos resultados com o *Radargun* (o sequenciador efectua cerca de metade do débito do sistema). No entanto, no nível de coerência RR+WS, nota-se uma quebra no débito do sistema e um aumento na taxa de cancelamento na nossa solução. Estes cancelamentos são originados pelo mecanismo de *write skew*. De qualquer modo, consegue-se sempre um desempenho superior ao 2PC. O custo da segunda fase na nossa solução pode ser visualizado na duração média da fase de confirmação de uma transacção.

O mecanismo de *write skew* é activado com alguma frequência neste *benchmark*, pois o padrão de acesso a dados durante a execução de uma transacção é o seguinte: obtém o objecto, efectua a modificação e armazena-o de volta. Isto activa o mecanismo de verificação do *write skew* cada vez que o valor de uma chave necessita de ser actualizado. Enquanto na nossa solução é visível o impacto deste mecanismo, no 2PC tal não se verifica. Isto deve-se ao facto

da verificação do *write skew* no 2PC só ser efectuada depois de adquirido o trinco necessário. Ora, como existe uma elevada contenção, a aquisição dos trincos falha e a verificação do *write skew* raramente é efectuada. De facto, somente 27% dos cancelamentos são originados pelo *write skew*, sendo que a maioria é devido a interbloqueios.

4.4.2.2 Replicação Parcial

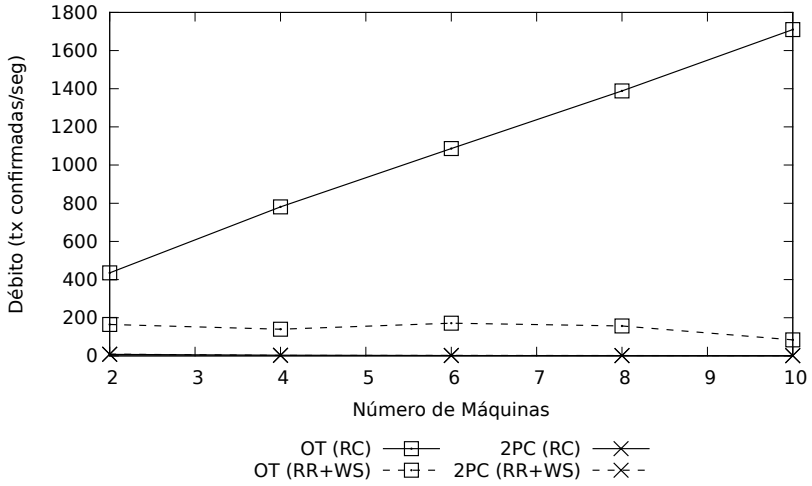
A Figura 4.7 ilustra os resultados da avaliação efectuada no TPC-C com replicação parcial. Tal como anteriormente, em ambos os graus de replicação, a nossa solução possui um débito superior ao 2PC. Também se consegue observar o impacto do *write skew* no débito e na taxa de cancelamento do sistema. De facto, a taxa de cancelamento é bastante elevada, verificando-se em média cerca de 60% a 70% de cancelamentos no sistema. Tal como no caso da replicação total, o 2PC apresenta cerca de 30% dos cancelamentos devido ao *write skew*. Finalmente, um facto digno de nota, ambas as soluções apresentam resultados semelhantes com grau de replicação 4. De facto, esta aplicação gera transacções de maio duração, cerca de 18 vezes mais lentas em comparação com o *Radargun*, o que evita a sobrecarga da rede.

4.5 Discussão

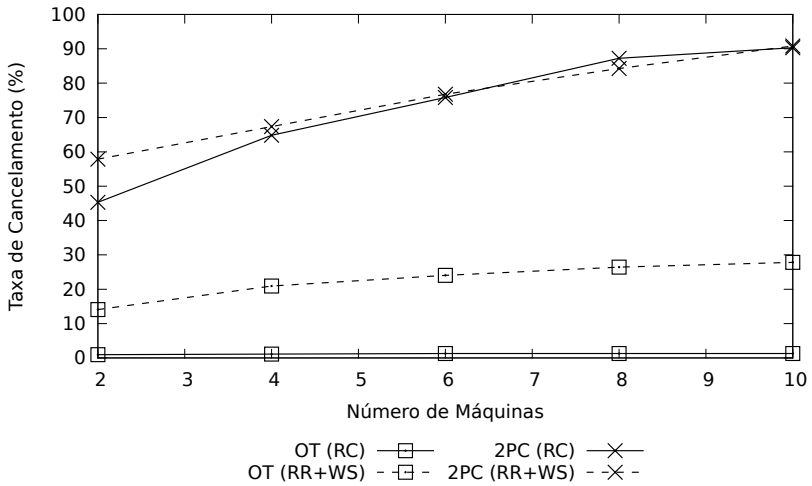
Analisando a avaliação descrita, concluímos que a nossa solução é eficiente em ambos os cenários avaliados. Além disso, verifica-se que o desempenho do 2PC está fortemente ligado ao nível do contenção que o sistema possui, ao contrário da nossa solução que tem um desempenho mais estável quer em cenário de elevada contenção, quem em cenário de baixa contenção.

Analisando as primitivas de AMcast, verifica-se que, neste contexto, é preferível usar um passo de comunicação adicional do que recorrer ao aumento do número de mensagens trocadas. Isto deve-se ao desempenho do sistema de comunicação em grupo utilizado, o *JGroups*, e da rede, que são os principais pontos de estrangulamento do sistema.

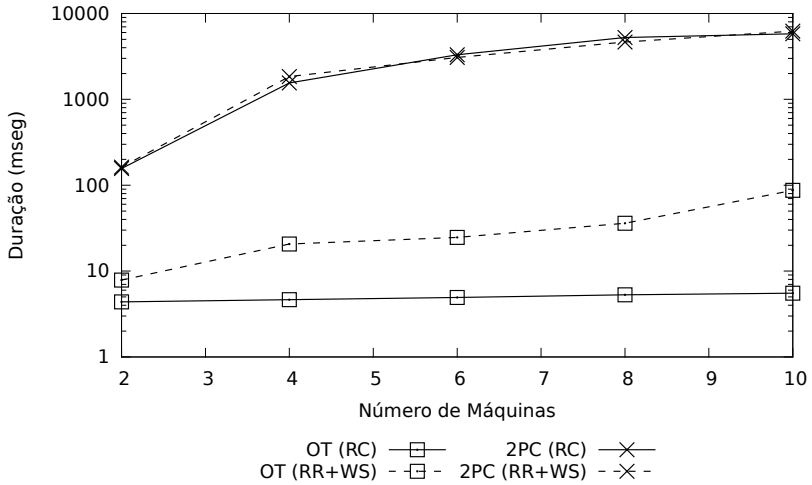
O custo da verificação do *write skew* é notório somente com o TPC-C. Isto deve-se aos acessos aleatórios do *Radargun*, onde a probabilidade desse mecanismo ser activado, é baixo. No entanto, com o TPC-C conclui-se que este mecanismo tem um forte impacto no desempenho do sistema, nomeadamente na nossa solução, pois evolui de uma taxa de cancelamento virtualmente nula, para um valor mais elevado.



(a) Débito (*c-tpcc-1* e *c-tpcc-2*)

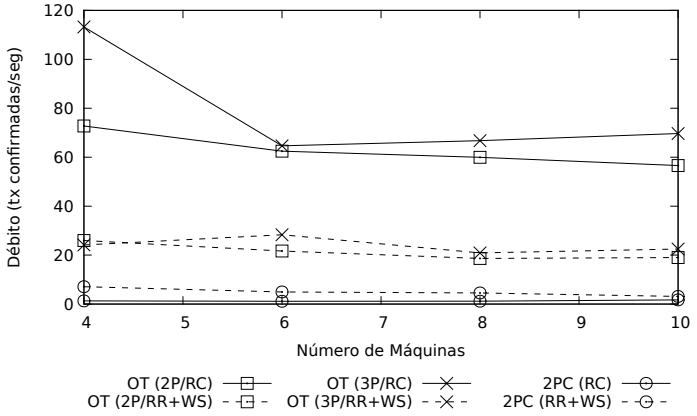


(b) Taxa de Cancelamento (*c-tpcc-1* e *c-tpcc-2*)

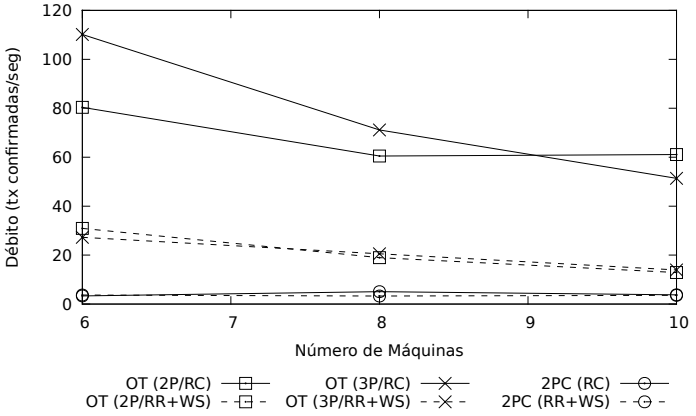


(c) Tempo de Confirmação médio (*c-tpcc-1* e *c-tpcc-2*)

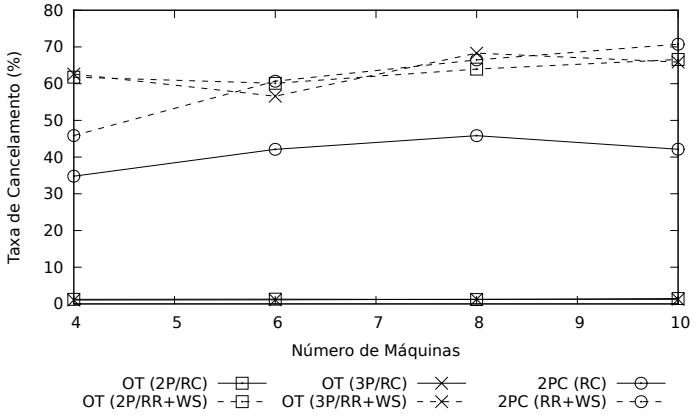
Figura 4.6: Resultados para Replicação Total no TPC-C



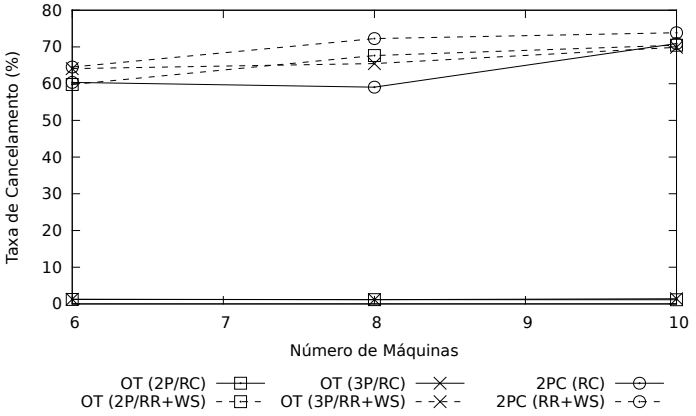
(a) Débito (*c-tpcc-3* e *c-tpcc-4*)



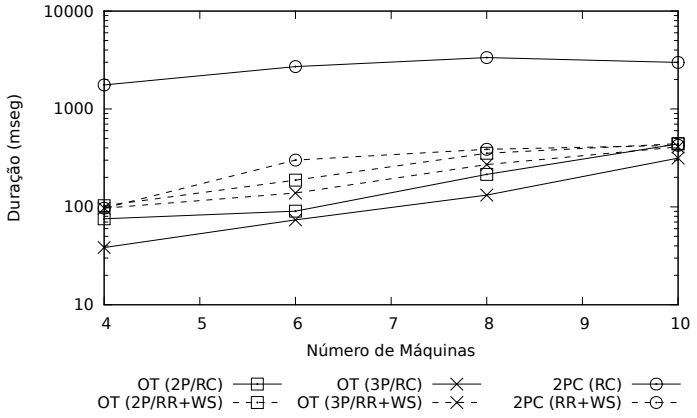
(b) Débito (*c-tpcc-5* e *c-tpcc-6*)



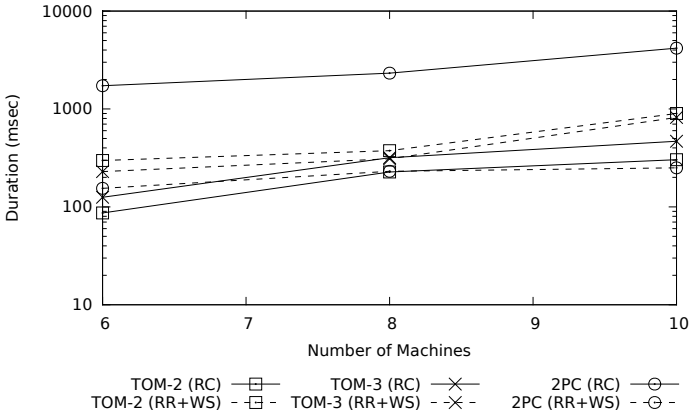
(c) Taxa de Cancelamento (*c-tpcc-3* e *c-tpcc-4*)



(d) Taxa de Cancelamento (*c-tpcc-5* e *c-tpcc-6*)



(e) Tempo de Confirmação médio (*c-tpcc-3* e *c-tpcc-4*)



(f) Tempo de Confirmação médio (*c-tpcc-5* e *c-tpcc-6*)

Figura 4.7: Resultados para Replicação Parcial no TPC-C. À esquerda com grau de replicação 2 e à direita com grau de replicação 4

A capacidade de configuração do *Radargun* permite avaliar com mais pormenor alguns dos mecanismos desenvolvidos. Em relação à certificação com múltiplos fios de execução, verifica-se que este é somente eficaz, em sistemas com certificações mais complexas. No contexto onde foi utilizado, não se obtém qualquer aumento de desempenho significativo com o seu uso.

5 Conclusão

5.1 Conclusões

Os sistemas de armazenamento de dados em memória emergiram como ferramenta para melhorar o desempenho das aplicações que necessitam de aceder grandes quantidade de dados, removendo os acessos a disco do fluxo principal de execução da aplicação. Estes sistemas também não incorrem nos custos de processar e optimizar o SQL. No entanto, oferecem suporte para transacções em memória, persistência assíncrona, distribuição e replicação de dados. Estas características tornam a execução de transacções mais eficiente, mas amplificam o custo de sincronização entre réplicas. A replicação total permite distribuir a carga entre as várias réplicas, mas incorre em grandes custos de sincronização, limitando a capacidade de escala do sistema. Desse modo, justifica-se a utilização de replicação parcial neste contexto.

Com esse objectivo, esta dissertação introduz uma alternativa aos protocolos de replicação actualmente utilizados em sistemas como o *Infinispan*. O *Infinispan* é um sistema de armazenamento de dados em memória distribuído e replicado, desenvolvido e distribuído em código aberto pela RedHat, usado no servidor aplicacional JBoss *Application Server* (JBoss AS). As soluções propostas utilizam as propriedades da difusão atómica selectiva (AMcast) para garantir uma ordem de certificação das transacções, com o objectivo de evitar interbloqueios distribuídos, originados pelo protocolo de confirmação em duas fase (2PC) em conjunto com o uso de trincos. O *Infinispan*foi utilizado como protótipo para avaliar as soluções desenvolvidas.

O primeiro algoritmo desenvolvido para replicação total utiliza a primitiva de difusão atómica (ABcast), para garantir a ordem total de entrega das mensagens. Esta solução melhorou o desempenho do sistema, obtendo um débito superior ao 2PC e uma taxa de cancelamento virtualmente nula. O passo seguinte foi a adaptação deste algoritmo para suportar replicação parcial. Para este efeito foi necessário desenvolver no *JGroups* a primitiva AMcast. Foram concretizadas duas variantes desta primitiva, de forma a comparar o custo de um passo

de comunicação com a troca de um número elevado de mensagens.

Ao fazermos a avaliação, verifica-se que a primitiva com 2 passos de comunicação possui um desempenho inferior ao da primitiva com 3 passos de comunicação. Desse modo, concluímos que, neste contexto, o número excessivo de mensagens trocadas leva a uma mais rápida saturação do sistema. No entanto, consegue-se obter sempre um débito superior ao 2PC para os casos avaliados.

5.2 Trabalho Futuro

Um rumo interessante para um trabalho futuro será optimizar e integrar de modo eficiente a primitiva de difusão atómica selectiva no *JGroups*. O objectivo principal deste trabalho não era desenvolver uma primitiva de comunicação eficiente, pelo que várias optimizações, ainda podem ser efectuadas. Além disso, a saída de membros, quer de modo ordenado, quer por falha, não é suportado na concretização actual. A sua concretização é necessária de modo a poder ser usado em ambientes reais.

É um facto conhecido que, na grande maioria das situações, oferecer um modelo de coerência forte é menos eficiente do que oferecer modelos de coerência fracos (Serrano, Patino-Martinez, Jimenez-Peris, & Kemme 2007). Desse modo, como referimos, o *Infinispan* opta por oferecer modelos de coerência fraca, tais como, *Read Committed* e *Repeatable Read*. No entanto, existem aplicação onde coerência forte é um requisito. Um exemplo dessas aplicações é o *FénixEDU* (Carvalho, Cachopo, Rodrigues, & Silva 2008). O *FénixEDU* é uma aplicação web que suporta diversas actividades de gestão no Instituto Superior Técnico, tais como, inscrições de alunos nas diversas actividades e cursos, gestão de horários, gestão de cursos, pagamentos de propinas, etc. Esta aplicação é usada por milhares de estudantes, professores e funcionários, processando cerca de 1,000,000 a 4,500,000 transacções por dias. Este tipo de sistema necessita de garantias de coerência fortes para o seu correcto funcionamento. Desta forma, é interessante estudar maneiras eficientes de oferecer também coerência forte.

Finalmente, seria também interessante efectuar uma avaliação num *cluster* de maior dimensão, de modo a observar-se o quão eficaz é a replicação parcial em comparação com a replicação total.

Referências

Aguilera, M. K., A. Merchant, M. Shah, A. Veitch, & C. Karamanolis (2007, October). Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, Stevenson, WA, USA, pp. 159–174. ACM.

Alonso, G. (1997, March). Partial Database Replication and Group Communication Primitives (Extended Abstract). In *Proceedings of the 2nd European Research Seminar on Advances in Distributed Systems*, ERSADS '97, Valais, Switzerland, pp. 171–176.

Ananian, C. S., K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, & S. Lie (2006, February). Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, San Francisco, CA, USA, pp. 316–327. IEEE Computer Society.

Atif, M. (2009, October). Analysis and verification of Two-Phase Commit and Three-Phase Commit Protocols. In *Proceedings of the International Conference on Emerging Technologies*, ICET '09, Islamabad, Pakistan, pp. 326–331. IEEE Computer Society.

Berenson, H., P. Bernstein, J. Gray, J. Melton, E. O'Neil, & P. O'Neil (1995, May). A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, San Jose, CA, USA, pp. 1–10. ACM.

Bernstein, P. A., V. Hadzilacos, & N. Goodman (1986). *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Bloom, B. H. (1970, July). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 422–426.

Cachopo, J. a. & A. Rito-Silva (2006, December). Versioned boxes as the basis for memory transactions. *Science of Computer Programming – Special issue: Synchronization and concurrency in object-oriented languages 63*, 172–185.

Carvalho, N., J. a. Cachopo, L. Rodrigues, & A. R. Silva (2008, March). Versioned transactio-

nal shared memory for the FénixEDU web application. In *Proceedings of the 2nd workshop on Dependable Distributed Data Management*, WDDDM '08, Glasgow, Scotland, pp. 15–18. ACM.

Carvalho, N., P. Romano, & L. Rodrigues (2010, November). Asynchronous Lease-Based Replication of Software Transactional Memory. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, Bangalore, India, pp. 376–396. Springer-Verlag.

Couceiro, M., P. Romano, N. Carvalho, & L. Rodrigues (2009, November). D2STM: Dependable Distributed Software Transactional Memory. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, Shanghai, China, pp. 307–313. IEEE Computer Society.

Coulon, C., E. Pacitti, & P. Valduriez (2005, July). Consistency management for partial replication in a high performance database cluster. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems*, ICPADS '05, Fukuoka, Japan, pp. 809–815. IEEE Computer Society.

Damron, P., A. Fedorova, Y. Lev, V. Luchangco, M. Moir, & D. Nussbaum (2006, October). Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XII, San Jose, CA, USA, pp. 336–346. ACM.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, & W. Vogels (2007, October). Dynamo: amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, Stevenson, WA, USA, pp. 205–220. ACM.

Dice, D., O. Shalev, & N. Shavit (2006, September). Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, DISC '06, Stockholm, Sweden, pp. 194–208. Springer.

Fitzpatrick, B. (2004, August). Distributed caching with memcached. *Linux Journal 2004*, 5.

Fraser, K. & T. Harris (2007, May). Concurrent programming without locks. *ACM Transactions on Computer Systems 25*, 5–es.

Gray, C. & D. Cheriton (1989, December). Leases: an efficient fault-tolerant mechanism for

distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, SOSP '89, Litchfield Park, AZ, USA, pp. 202–210. ACM.

Gray, J., P. Helland, P. O'Neil, & D. Shasha (1996, June). The dangers of replication and a solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, Montreal, QC, Canada, pp. 173–182. ACM.

Guerraoui, R. & M. Kapalka (2008, February). On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, Salt Lake City, UT, USA, pp. 175–184. ACM.

Guerraoui, R. & A. Schiper (2001, March). Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science 254*, 297–316.

Hadzilacos, V. & S. Toueg (1993). *Fault-tolerant broadcasts and related problems* (2nd ed.)., pp. 97–145. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Hammond, L., V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, & K. Olukotun (2004, June). Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, Munich, Germany, pp. 102–113. IEEE Computer Society.

Herlihy, M., V. Luchangco, & M. Moir (2003, July). Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, PODC '03, Boston, MA, USA, pp. 92–101. ACM.

Herlihy, M. & J. E. B. Moss (1993, May). Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, San Diego, CA, USA, pp. 289–300. ACM.

Kemme, B., F. Pedone, G. Alonso, A. Schiper, & M. Wiesmann (2003, July - August). Using Optimistic Atomic Broadcast in Transaction Processing Systems. *IEEE Transactions on Knowledge and Data Engineering 15*, 1018–1032.

Kotselidis, C., M. Ansari, K. Jarvis, M. Luján, C. Kirkham, & I. Watson (2008, September). DiSTM: A software transactional memory framework for clusters. In *Proceedings of the 37th International Conference on Parallel Processing*, ICPP '08, Portland, OR, USA, pp. 51–58. IEEE Computer Society.

Kumar, S., M. Chu, C. J. Hughes, P. Kundu, & A. Nguyen (2006, March). Hybrid transac-

tional memory. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, Manhattan, NY, USA, pp. 209–220. ACM.

Lakshman, A. & P. Malik (2010, April). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 35–40.

Lamport, L. (1978, July). Ti clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 558–565.

Marathe, V. J., W. N. S. Iii, & M. L. Scott (2005, September). Adaptive Software Transactional Memory. In *In Proceedings of the 19th International Symposium on Distributed Computing*, DISC '05, Cracow, Poland, pp. 354–368. Springer.

Minh, C. C., M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, & K. Olukotun (2007, June). An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, San Diego, CA, USA, pp. 69–80. ACM.

Moore, K., J. Bobba, M. Moravan, M. Hill, & D. Wood (2006, February). LogTM: log-based transactional memory. In *Proceesings of the 12th International Symposium on High-Performance Computer Architecture*, HPCA '06, Austin, TX, USA, pp. 254–265. IEEE Computer Society.

Osrael, J. (2007). *Replication Techniques for Balancing Data Integrity with Availability*. Ph. D. thesis, Institut fur Softwaretechnik und Interaktive Systeme, Arbeitsbereich fur Business Informaticsu Technische Universitat Wien.

Palmieri, R., F. Quaglia, & P. Romano (2010, July). AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing. In *Proceedings of the 9th IEEE International Symposium on Network Computing and Applications*, NCA '10, Cambridge, MA, USA, pp. 20–27. IEEE Computer Society.

Pedone, F., R. Guerraoui, & A. Schiper (1998, September). Exploiting Atomic Broadcast in Replicated Databases. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, Euro-Par '98, Southampton, UK, pp. 513–520. Springer-Verlag.

Pedone, F., R. Guerraoui, & A. Schiper (2003, July). The Database State Machine Approach. *Distributed and Parallel Databases 14*, 71–98.

Pedone, F. & A. Schiper (1998, September). Optimistic Atomic Broadcast. In *Proceedings of*

the 12th International Symposium on Distributed Computing, DISC '98, Andros, Greece, pp. 318–332. Springer-Verlag.

Ports, D. R. K., A. T. Clements, I. Zhang, S. Madden, & B. Liskov (2010, October). Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, Vancouver, BC, Canada, pp. 1–15. USENIX Association.

Rajwar, R., M. Herlihy, & K. Lai (2005, May). Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, Madison, WI, USA, pp. 494–505. IEEE Computer Society.

Ramakrishnan, R. & J. Gehrke (2000). *Database Management Systems* (2nd ed.). McGraw-Hill Higher Education.

Rodrigues, L., J. Mocito, & N. Carvalho (2006, April). From spontaneous total order to uniform total order: different degrees of optimistic delivery. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing*, SAC '06, Dijon, France, pp. 723–727. ACM.

Romano, P., N. Carvalho, & L. Rodrigues (2008, September). Towards distributed software transactional memory systems. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, Yorktown, NY, USA, pp. 4:1–4:4. ACM.

Ruivo, P., M. Couceiro, P. Romano, & L. Rodrigues (2011a, December). Exploiting Total Order Multicast in Weakly Consistent Transactional Caches. In *Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '11, Pasadena, CA, USA.

Ruivo, P., M. Couceiro, P. Romano, & L. Rodrigues (2011b, September). Replicação Parcial em Sistemas de Memória Transaccional. In *Proceedings of the 3rd INForum – Simpósio de Informática*, Inforum '11, Coimbra, Portugal, pp. 366–377.

Saha, B., A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, & B. Hertzberg (2006, March). McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Manhattan, NY, USA, pp. 187–197. ACM.

Schiper, A., K. Birman, & P. Stephenson (1991, August). Lightweight causal and atomic

group multicast. *ACM Transactions on Computer Systems 9*, 272–314.

Schiper, N. & F. Pedone (2008, December). Solving Atomic Multicast When Groups Crash. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, OPODIS '08, Luxor, Egypt, pp. 481–495. Springer-Verlag.

Schiper, N., R. Schmidt, & F. Pedone (2006, December). Optimistic Algorithms for Partial Database Replication. In *Proceedings of the 10th International Conference On Principles Of Distributed Systems*, OPODIS '06, Bordeaux, France, pp. 81–93. Springer-Verlag.

Schiper, N., P. Sutra, & F. Pedone (2010, October - November). P-Store: Genuine Partial Replication in Wide Area Networks. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, New Delhi, Punjab, India, pp. 214–224. IEEE Computer Society.

Schneider, F. (1990). The state machine approach: A tutorial. In B. Simons & A. Spector (Eds.), *Fault-Tolerant Distributed Computing*, Volume 448 of *Lecture Notes in Computer Science*, pp. 18–41. Springer Berlin - Heidelberg.

Serrano, D., M. Patino-Martinez, R. Jimenez-Peris, & B. Kemme (2007, December). Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, PRDC '07, Melbourne, Australia, pp. 290–297. IEEE Computer Society.

Shavit, N. & D. Touitou (1995, August). Software Transactional Memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, Ottowa, ON, Canada, pp. 204–213. ACM.

Shriraman, A., S. Dwarkadas, & M. L. Scott (2008, June). Flexible Decoupled Transactional Memory Support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, Beijing, China, pp. 139–150. IEEE Computer Society.

Shriraman, A., M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, & M. L. Scott (2007, June). An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, San Diego, CA, USA, pp. 104–115. ACM.

Sousa, A., F. Pedone, F. Moura, & R. Oliveira (2001, October). Partial Replication in the Database State Machine. In *Proceedings of the IEEE International Symposium on Network*

Computing and Applications, NCA '01, Cambridge, MA, USA, pp. 298–309. IEEE Computer Society.

Vogels, W. (2008, October). Eventually Consistent. *Queue - Scalable Web Services 6*, 14–19.

Ye, H., B. Kerherve, G. v Bochmann, & D. Bourne (2002, October). Towards database scalability through efficient data distribution in e-commerce environments. In *Proceedings of the 3rd International Symposium on Electronic Commerce*, ISEC '02, Durham, NC, USA, pp. 87–95. IEEE Computer Society.

Zhang, J., W. Chen, X. Tian, & W. Zheng (2008, December). Exploring the Emerging Applications for Transactional Memory. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '08, Dunedin, New Zealand, pp. 474–480. IEEE Computer Society.

Zou, H. & F. Jahanian (1998, October). Optimization of a Real-Time Primary-Backup Replication Service. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, SRDS '98, West Lafayette, Indiana, pp. 177–185. IEEE Computer Society.