UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

Dissertação

Uma Linguagem de Domínio Específico para Memória Transacional Distribuída em Java

Jerônimo da Cunha Ramos

Pelotas, 2015

Jerônimo da Cunha Ramos

Uma Linguagem de Domínio Específico para Memória Transacional Distribuída em Java

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Mestre em Ciência da Computação

Orientador: Prof. Dr. André Rauber Du Bois
Coorientador: Prof. Dr. Maurício Lima Pilla

Pelotas, 2015

Dados Internacionais de Catalogação na Publicação (CIP)

S175l Ramos, Jerônimo da Cunha.

Uma Linguagem de Domínio Específico para Memória Transacional Distribuída em Java. / Jerônimo da Cunha Ramos. – Pelotas, 2015.
68f. il. color. ; 30 cm.

Orientador: Dr. André Rauber Du Bois.
Coorientador: Dr. Maurício Lima Pilla.
Dissertação (Mestrado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2015.

1. Ciência da computação. 2. Memória transacional. 3. Processamento paralelo e distribuído. 4. Linguagem de programação. I. Du Bois, André Rauber. II. Pilla, Maurício Lima. III. Título.

CDD 005.1

Bibliotecária Responsável:
Francine Couto de Oliveira CRB10/2183

UNIVERSIDADE FEDERAL DE PELOTAS
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
CENTRO DE DESENVOLVIMENTO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

PPGC
Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Pelotas

ATA DE APRECIAÇÃO SOBRE A DEFESA DE DISSERTAÇÃO DE MESTRADO

NOME DA ESTUDANTE		MATRÍCULA
JERÔNIMO DA CUNHA RAMOS		12121011
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO		MESTRADO

MEMBROS DA BANCA EXAMINADORA	TÍTULO	ASSINATURA
Prof. Dr. Rodrigo Machado (UFRGS)	Doutor	
Prof. Dr. Adenaur Corrêa Yamin (UFPel)	Doutor	
Prof. Dr. Gerson Cavalheiro (UFPel)	Doutor	
Prof.ª Drª Juliana Kaizer Vizzoto (UFSM)	Doutor	

APRECIAÇÃO SOBRE A DISSERTAÇÃO

NÃO SIGILOSA

Em 28 de abril de 2015 os membros acima nomeados para a defesa da Dissertação do aluno **JERÔNIMO DA CUNHA RAMOS**, matriculado no Programa de Pós-Graduação em Computação, consideraram aprovado, estabelecendo o título definitivo da Dissertação como sendo "Uma linguagem de Domínio Especifico para Memória Transacional Distribuida em Java", e estabelecendo um prazo máximo de 30 dias para as correções e entrega da versão definitiva.

DADOS PESSOAIS DOS MEMBROS DA BANCA EXAMINADORA

NOME COMPLETO	CPF	ANO NASCIMENTO	TITULAÇÃO		
			Área	Local	Ano
Prof. Dr. Rodrigo Machado (UFRGS)	001.702.620-22	1981	Computação	UFRGS	2012
Prof. Dr. Adenaur Corrêa Yamin (UFPel)	291.222.010-68	1959	Computação	UFRGS	2004
Prof. Dr. Gerson Cavalheiro)	578797770-97	1968	Computação	INPG	1999
Prof.ª Drª Juliana Kaizer Vizzoto (UFSM)	939.174.900-30	1978	Computação	UFRGS	2006

1ª Via – Coordenador do Curso; 2ª Via – Orientador; 3ª Via – PRPPG.

DISTRIBUIÇÃO A CARGO DA COORDENAÇÃO DO PROGRAMA.

RESUMO

RAMOS, Jerônimo da Cunha. **Uma Linguagem de Domínio Específico para Memória Transacional Distribuída em Java**. 2015. 68 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2015.

Memória transacional é uma abstração na qual os acessos concorrentes à memória são gerenciados com o uso de transações, similares às de banco de dados. Existem muitas vantagens no uso deste modelo ao invés dos convencionais. Uma delas é a facilidade de programação, pois basta que o programador marque os trechos onde pode ocorrer condição de corrida e o sistema transacional se encarrega de garantir a corretude dos resultados. Outra vantagem importante é a maior possibilidade de exploração do paralelismo, já que as transações podem ser executadas de maneira otimista. Este modelo tem se mostrado promissor para máquinas SMP, no entanto existem poucos trabalhos na área focados em arquiteturas distribuídas. O estudo de arquiteturas distribuídas é extremamente importante nos dias de hoje, tanto por comporem uma grande parcela da computação de alto desempenho, quanto pela tendência de cada vez mais dispositivos computacionais comunicarem-se. Sendo assim, o presente trabalho engloba uma revisão bibliográfica sobre as principais estratégias de implementação de memórias transacionais, algumas das implementações mais importantes e as principais dificuldades para implementação de memórias transacionais distribuídas, bem como aborda as implementações, que compõe o estado da arte, da frente de pesquisa na qual este trabalho está inserido: memórias transacionais distribuídas. Deste modo, o objetivo principal deste trabalho é propor uma linguagem de domínio específico embutida em Java para implementação de transações de memória que podem envolver objetos locais e distribuídos em uma rede. Este objetivo foi abordado tanto em nível de linguagem, permitindo que o programador defina ações transacionais, quanto em nível de sistema transacional, dando suporte a execução das abstrações da linguagem. Além disso, foi implementado um protótipo, estendendo a implementação da linguagem CMTJava, para validação do modelo.

Palavras-chave: Memória Transacional, Processamento Paralelo e Distribuído, Linguagem de Programação.

ABSTRACT

RAMOS, Jerônimo da Cunha. **A Domain Specific Language for Distributed Transactional Memory in Java**. 2015. 68 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2015.

Transactional memory is an abstraction where concurrent memory accesses are handled with the use of transactions, similar to database transactions. There are many advantages in using this model instead of conventional ones. An advantage is the ease of programming, because the programmer simply marks the snippets where race condition may occur and the transactional system takes care to ensure the correctness of the results. Another important advantage is the increased possibility of exploitation of parallelism because transactions can be performed optimistically. This model has proven to be promising for SMP machines, however few studies in this area are focused on distributed architectures. The study of distributed architectures is extremely important these days, as they compose a large portion of high performance computing platforms, and because of the trend of more and more computing devices communicating with each other. Thus, this work includes a literature review on the main implementation strategies of transactional memories, some of the most important implementations and the main difficulties to implement distributed transactional memories. It also discusses the state of the art implementations of the research front where this work is inserted: distributed transactional memories. In this way, the main objective of this work is to propose a domain specific language embedded in Java for implementing memory transactions that can involve local objects and distributed objects on a network. This goal was approached both in language level, allowing the programmer to define transactional actions, and in transactional system level, supporting the implementation of the language's abstractions. In addition, a prototype was implemented, extending the implementation of CMTJava language, to validate the model.

Keywords: Transactional Memory, Parallel and Distributed Computing, Programming Language.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
cc-STM	cache-coherent Software Transactional Memory
DSM	Distributed Shared Memory
HaSTM	Hardware-assisted Software Transactional Memory
HTM	Hardware Transactional Memory
HyTM	Hybrid Transactional Memory
LSA	Lazy Snapshot Algorithm
MPI	Message-Passing Interface
NUMA	Non-Uniform Memory Access
PGAS	Partitioned Global Address Space
RAM	Random Access Memory
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SD	Sistema Distribuído
SSCA	Scalable Synthetic Compact Application
STAMP	Stanford Transactional Applications for Multi-Processing
STM	Software Transactional Memory
TCC	Transactional Coherence and Consistency
TL2	Transactional Locking II
TM	Transactional Memory

SUMÁRIO

1 INTRODUÇÃO

Para tirar proveito do paralelismo disponibilizado pelos computadores atuais, os programas devem conter atividades que possam executar concorrentemente. Normalmente, em arquiteturas de memória compartilhada, isto é feito com o uso de *threads*, variáveis compartilhadas e *locks*. Porém, este modelo de programação impõe quase toda a complexidade de exploração do paralelismo e de tratamento das condições de corrida ao programador, o que torna-o propenso a erros, como *deadlocks*. A comunidade científica vem propondo alternativas a este modelo para facilitar e popularizar a programação concorrente, como é o caso das memórias transacionais (RIGO; CENTODUCATTE; BALDASSIN, 2007).

Memória transacional é uma abstração para programação concorrente baseada na ideia de transações, similares às de banco de dados. Uma transação de memória é uma sequência atômica de operações que modificam a memória e que podem ser executadas completamente ou podem ser abortadas. Desta maneira, transações de memória executam como se estivessem modificando uma área de memória isolada do acesso de outras transações. Estas só conseguem ver o resultado após o *commit* (DU BOIS, 2008).

Este modelo possui várias vantagens sobre o modelo clássico de programação, sendo as principais: ausência de *deadlock*, possibilidade de abortar e retornar ao estado original do programa, maior possibilidade de exploração do paralelismo, facilidade de programação, escalabilidade e composabilidade (RIGO; CENTODUCATTE; BALDASSIN, 2007; DU BOIS, 2008).

Nos últimos anos as pesquisas sobre Memórias Transacionais têm sido bastante focadas em máquinas *multicore*, deixando outros tipos de arquiteturas, como *clusters* e máquinas NUMA (*Non-Uniform Memory Access*) quase inexploradas (LU; WANG; LU, 2010; KOTSELIDIS et al., 2008; BOCCHINO; ADVE; CHAMBERLAIN, 2008). Estas arquiteturas tem como principal diferencial o tempo variado de acesso à memória local e remota. No caso dos *clusters* e outros sistemas distribuídos (SD), cada nodo possui seu próprio espaço de endereçamento de memória, fazendo com que a comunicação tenha que ser feita por troca de mensagens e, geralmente, não haja

coerência de cache.

Existem diversas implementações de sistemas de memórias transacionais para máquinas *multicore*, tais como TinySTM (FELBER; FETZER; RIEGEL, 2008), SwissTM (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009), AdaptSTM (PAYER; GROSS, 2011) e CMTJava (DU BOIS; ECHEVARRIA, 2009), porém o foco destas não está em arquiteturas distribuídas, sendo assim não preveem soluções para os dificultantes citados acima. As propostas focadas em arquiteturas distribuídas são pouco numerosas, sendo estas focadas em *clusters*, como é o caso de Cluster-STM (BOCCHINO; ADVE; CHAMBERLAIN, 2008), DiSTM (KOTSELIDIS et al., 2008) e TFA (SAAD; RAVINDRAN, 2012).

Este trabalho propõe uma linguagem de domínio específico embutida em Java para implementação de transações de memória que podem envolver objetos locais e distribuídos em uma rede. As contribuições deste trabalho são:

- **Linguagem DCMTJava:** Permite que programadores possam definir ações transacionais que são objetos em Java, dessa forma métodos podem receber transações como argumentos e retornar transações como resposta de sua execução. Essas transações podem ser compostas usando uma notação especial, gerando novas ações transacionais. DCMTJava estende a linguagem CMTJava (BANDEIRA, 2013) permitindo que transações envolvam tanto objetos locais quanto remotos.

- **Sistema transacional distribuído:** Para dar suporte às abstrações da linguagem é necessário um sistema de tempo de execução que permita a composição de transações envolvendo objetos locais e remotos. O sistema aqui proposto combina dois algoritmos para transações; para objetos locais é usando um algoritmo baseado no swissTM (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009), onde conflitos de escrita-escrita são detectados de forma adiantada, evitando que essas transações executem até o final. Para objetos distribuídos os conflitos são detectados tardiamente, evitando comunicação excessiva, durante a execução das transações.

- **Prototipação:** O algoritmo proposto foi implementado, em Java, estendendo a implementação da linguagem CMTJava. Para validação, foram feitos casos de teste envolvendo diferentes números de transações com objetos locais e distribuídos

Estrutura do Texto

Este texto é composto por seis capítulos, sendo o primeiro deles esta introdução e os restantes os descritos a seguir:

- **Capítulo 2 - Memórias Transacionais:** Além de uma breve introdução ao assunto e alguns conceitos básicos, este capítulo é composto por três seções descritas a seguir:
  - **Estratégias de Implementação:** Nesta seção encontram-se diferentes formas de implementação das principais funcionalidades de sistemas transacionais, tais como detecção de conflitos, versionamento de dados, granularidade de conflitos e gerenciamento de contenção.
  - **Implementações de Memórias Transacionais:** Aqui são descritos brevemente alguns dos principais sistemas transacionais para arquiteturas *multi-core* e quais estratégias cada um deles utiliza.
  - **Memória Transacional Distribuída: Principais Dificuldades:** Esta seção descreve e aponta soluções para duas das principais dificuldades encontradas ao trabalhar-se com um sistema distribuído: sincronização e comunicação.
- **Capítulo 3 - Trabalhos Relacionados:** Este capítulo descreve três trabalhos do estado da arte de memórias transacionais distribuídas: Cluster-STM, DiSTM e TFA.
- **Capítulo 4 - DCMTJava:** Traz detalhes da linguagem e do sistema transacional distribuído desenvolvidos neste trabalho, e está organizado em 3 seções, sendo elas:
  - **Definição da Linguagem:** Esta seção apresenta características da linguagem implementada e traz alguns exemplos para demonstrar o uso de suas construções.
  - **Conceitos Gerais da Implementação:** Revisa alguns conceitos utilizados na implementação do sistema transacional, bem como traz algumas informações gerais sobre o compilador da linguagem.
  - **Sistema Transacional Distribuído:** Contém a classificação do sistema transacional nas diversas estratégias de implementação apresentadas na seção 2.1, bem como características da implementação atual e detalhes sobre o algoritmo utilizado por ela.
- **Capítulo 5 - Validação da Implementação:** Este capítulo contém informações sobre como foi realizada a validação da implementação e resultados obtidos.
- **Capítulo 6 - Considerações Finais:** Traz as conclusões e os trabalhos a serem realizados na continuidade desta pesquisa.

2 MEMÓRIAS TRANSACIONAIS

O termo “memória transacional” (do inglês: *transactional memory* ou TM) foi definido como *“uma nova arquitetura para multiprocessadores que objetiva tornar a sincronização livre de bloqueios tão eficiente (e fácil de usar) quanto técnicas convencionais baseadas em exclusão mútua”* (HERLIHY; MOSS, 1993) e é utilizado para definir qualquer mecanismo que utilize o conceito de transação para gerenciar o acesso concorrente à memória.

O conceito de transação é similar ao das transações de bancos de dados. Sendo, neste texto, definido como uma sequência de instruções executadas sequencialmente por uma *thread* respeitando as propriedades de atomicidade e de isolamento. Atomicidade é a propriedade que diz que todas as instruções são executadas (efetivação ou *commit*) ou nenhuma o é (aborto). Já a propriedade de isolamento diz que uma transação só pode enxergar os resultados da execução de outra transação após a efetivação da primeira, não podendo ver resultados intermediários. Estas propriedades garantem que o resultado da execução concorrente seja o mesmo da execução serial das transações.

Exitem diversos modelos de memórias transacionais: em hardware, em software e os modelos mistos. No modelo de Memória Transacional em Hardware (HTM), a arquitetura do processador dá suporte a toda a execução das transações, versionamento de dados e detecção de conflitos. Os dados ficam armazenados em registradores e na cache, só sendo copiados para a memória após a efetivação das transações. Já no modelo de Memória Transacional em Software (STM), o sistema de execução transacional é totalmente implementado em software. Um dos modelos mistos é o de Memória Transacional Híbrida (HyTM), onde a execução transacional ocorre em hardware, até que os limites deste sejam excedidos, quando o software assume parte do trabalho. Outro modelo é o de Memória Transacional em Software assistida por Hardware (HaSTM), onde a execução transacional é realizada em software, porém o hardware da algum suporte especial para acelerá-la (ECHEVARRIA, 2010; BANDEIRA, 2012). Este trabalho se foca em implementações de STM, porém grande parte dos conceitos nele apresentado também são válidos para os outros modelos.

2.1 Estratégias de Implementação

Diversas são as alternativas de implementação possíveis para sistemas de memórias transacionais e estas influenciam diretamente no desempenho do sistema. As principais alternativas dizem respeito à detecção de conflitos, ao versionamento de dados, a granularidade de conflitos e ao gerenciamento de contenção. Estes tópicos serão discutidos nesta seção.

2.1.1 Detecção de Conflitos

Responsável por garantir o isolamento entre as transações, a detecção de conflitos é um mecanismo essencial para o sistema de TM. Uma outra forma de garantir o isolamento seria executando somente uma transação por vez. Desta maneira, entretanto, nenhum paralelismo seria explorado.

A maioria dos sistemas de TM mantêm o controle das posições de memória em que as transações leram ou escreveram. Geralmente, cada transação armazena os endereços acessados em um conjunto de leitura (*read set*) e em um conjunto de escrita (*write set*), de acordo com o tipo de acesso. Sendo assim, um conflito ocorre quando há sobreposição entre o conjunto de escrita de uma transação e o conjunto de escrita ou de leitura de outra (RIGO; CENTODUCATTE; BALDASSIN, 2007).

A detecção de conflitos pode ser classificada em adiantada (*eager*) ou tardia (*lazy*), conforme pode ser visto a seguir.

2.1.1.1 Detecção Adiantada

Na detecção adiantada, também conhecida como pessimista, o conflito é detectado no momento em que uma transação acessa alguma posição de memória. A Figura 1 mostra três exemplos de conflitos que podem ocorrer quando utiliza-se detecção adiantada. No exemplo A, a transação T2 tenta ler a variável X, que já foi escrita por T1. Neste caso, T2 percebe o conflito e decide esperar um tempo aleatório, para tentar novamente após a efetivação de T1. Já no exemplo B, T2 escreve na variável X e esta já havia sido lida por T1. Como o valor de X lido por T1 tornou-se inconsistente, T1 é abortada e reinicia após um tempo aleatório. No exemplo C, as escritas de uma transação causam o aborto da outra e vice-e-versa, sucessivamente. Este caso é conhecido como *livelock*.

A principal vantagem do uso de detecção adiantada é evitar que transações com conflito executem desnecessariamente até o final antes de serem abortadas. Em contrapartida, pode ocorrer o aborto de transações que, dependendo do progresso de outras, poderiam ser efetivadas.

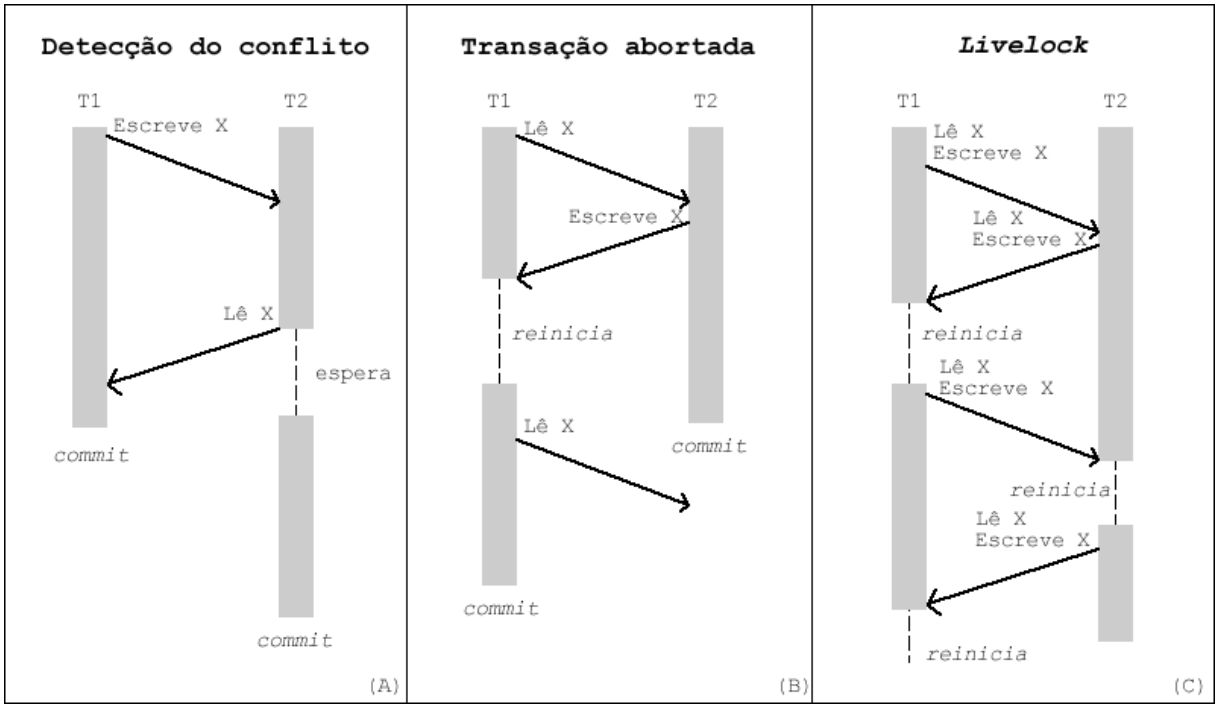

Figura 1: Detecção de Conflitos Adiantada (ECHEVARRIA, 2010)

2.1.1.2 Detecção Tardia

Quando utiliza-se detecção tardia, também conhecida como atrasada ou otimista, os conflitos só são verificados no momento da efetivação de alguma das transações. A Figura 2 mostra dois exemplos. No exemplo A, T1 escreve em X e T2 lê X. No momento em que T1 é efetivada, é verificado o conflito e T2 é reiniciada. No exemplo B, T2 lê e escreve em X, em seguida T1 também lê e escreve em X. Quando T1 é efetivada T2 é reiniciada.

A detecção tardia evita que ocorram *livelocks*, como ocorrem na adiantada. Porém, uma transação muito grande pode ser abortada várias vezes consecutivas por transações menores e nunca vir a ser efetivada. Esta situação é conhecida como *starvation*.

2.1.2 Versionamento de Dados

Como os dados alterados por uma transação só se tornam visíveis depois desta ser efetivada, é necessário manter tanto a versão modificada do dado quanto a versão antiga (MOORE et al., 2006). Se a transação for efetivada, a cópia modificada é mantida e a antiga é descartada. Já em caso de aborto, a copia antiga é mantida e a modificada descartada. Em geral, existem duas técnicas de se realizar versionamento dos dados: versionamento adiantado e versionamento tardio, como pode ser visto a seguir.

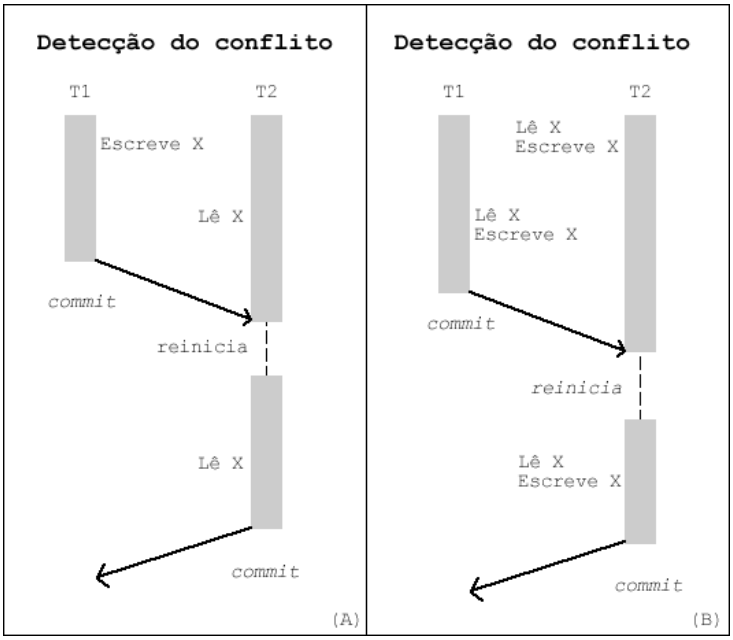

Figura 2: Detecção de Conflitos Tardia (ECHEVARRIA, 2010)

2.1.2.1 Versionamento Adiantado

Quando se utiliza versionamento adiantado, as transações escrevem os novos dados diretamente na memória, guardando o valor antigo em um *log*. Se a transação for efetivada, o valor escrito em memória é mantido e o valor que estava guardado no *log* é descartado. Em caso de cancelamento da transação, o valor mantido em *log* é copiado para a memória e o valor que estava na memória é descartado. Ambos os casos podem ser vistos na Figura 3.

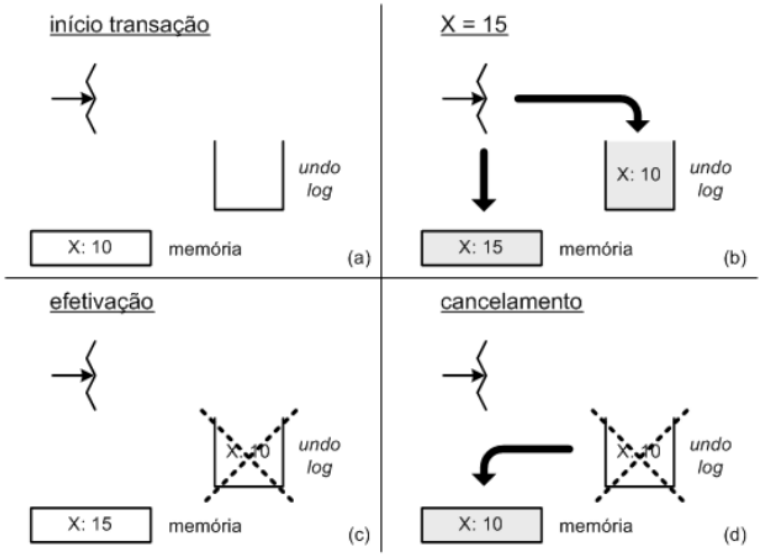

Figura 3: Versionamento de dados adiantado (RIGO; CENTODUCATTE; BALDASSIN, 2007)

A vantagem do versionamento adiantado está na efetivação mais simples e rápida das transações, já que os dados já foram gravados em memória, basta descartar o *log*. Porém, em caso de aborto, os dados do *log* devem ser restaurados, o que torna-o mais lento.

Usar o versionamento adiantado implica no uso de detecção de conflitos adiantada, pois como as escritas são realizadas diretamente na memória, é necessário que se garanta a exclusão mútua.

2.1.2.2 Versionamento Tardio

Com o versionamento de dados tardio, os dados novos não são escritos diretamente na área de memória em questão, e sim em um *buffer*. Em caso de efetivação, os valores do *buffer* deverão ser transferidos para a memória, já em caso de aborto, o *buffer* é descartado. A Figura 4 ilustra estes casos.

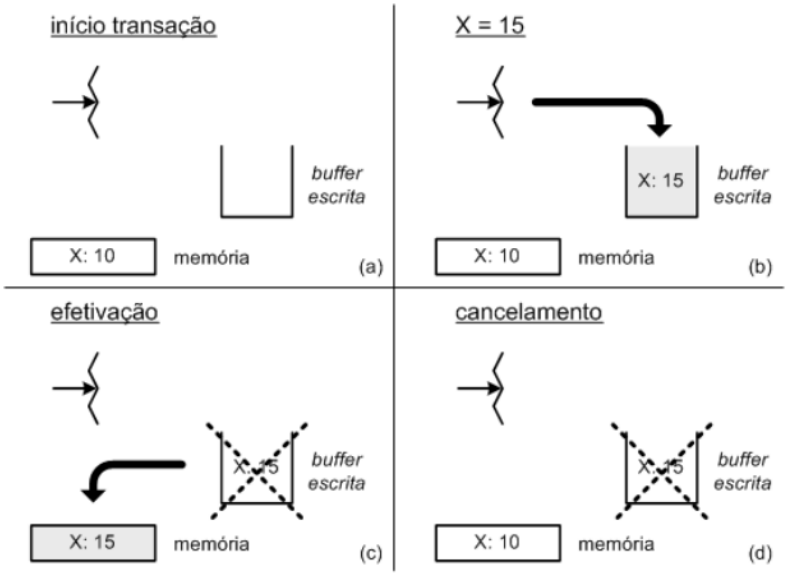

Figura 4: Versionamento de dados tardio (RIGO; CENTODUCATTE; BALDASSIN, 2007)

Ao contrario do versionamento adiantado, o tardio tem a efetivação mais lenta, pois os valores do *buffer* devem ser movidos para a memória. Já o aborto é mais simples, pois basta descartar o *buffer*.

2.1.3 Granularidade de Conflitos

A granularidade de conflitos se refere à unidade de armazenamento mínima na qual um sistema transacional detecta a ocorrência de conflitos. Em sistemas de memórias transacionais, os níveis de detecção mais comuns são: nível de objeto, de palavra ou de bloco de palavras. É necessário que haja um metadado associado a cada uma destas unidades de armazenamento, como um *lock* ou alguma estrutura

mais complexa contendo mais informações.

2.1.3.1 Nível de Objeto

É, em geral, a granularidade de mais simples implementação. Um metadado é mantido para cada objeto e todos os atributos deste são associados a uma única unidade de sincronização, como visto na Figura 5. Uma alternativa eficiente é implantar o metadado no cabeçalho do objeto. Isto reduz as faltas de cache e possibilita um único acesso ao metadado para uma série de acessos a diferentes atributos de um mesmo objeto (HARRIS; LARUS; RAJWAR, 2010).

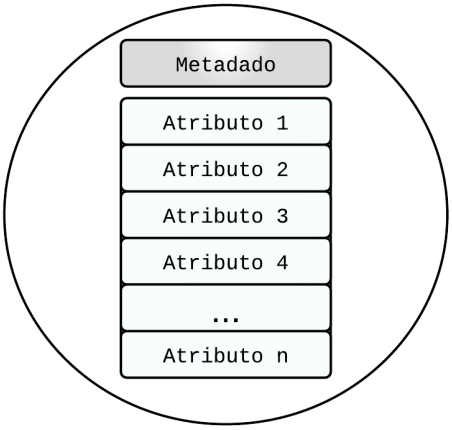

Figura 5: Nível de Objetos (BANDEIRA, 2012)

A desvantagem deste método é que ele causa a detecção de falsos conflitos. Pois se duas transações acessarem atributos distintos de um mesmo objeto, isto será considerado um conflito, quando na verdade não o é.

2.1.3.2 Nível de Palavra

Um metadado é relacionado a cada palavra de memória ao invés de a um objeto inteiro (Figura 6). Este método elimina a ocorrência de falsos conflitos, porém, com o grão tão pequeno há um grande *overhead* na detecção de conflitos, em termos de espaço e tempo.

2.1.3.3 Nível de Bloco de Palavras

Os metadados estão atrelados a blocos de palavras (Figura 7). Esta é uma técnica intermediária entre as duas anteriores. Ela diminui o *overhead* de gerenciamento, já que aumenta o tamanho do grão. Porém permite que ocorram falsos conflitos, como ocorre no nível de objetos, mas em menor número.

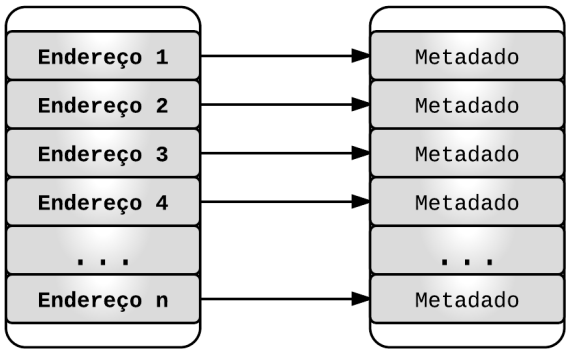

Figura 6: Nível de Palavra (BANDEIRA, 2012)

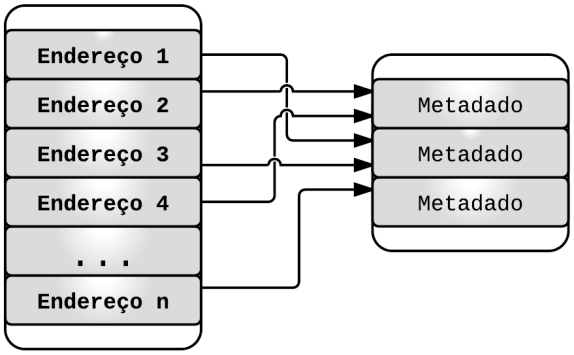

Figura 7: Nível de Bloco de Palavras (BANDEIRA, 2012)

2.1.4 Gerenciamento de Contenção

Muitos sistemas de TM possuem um gerenciador de contenção. Tal gerenciador implementa uma ou mais politicas de resolução de conflitos que entram em ação após a detecção do conflito. Estas politicas são responsáveis por decidir qual transação vai abortar e se alguma delas deve ser atrasada por um tempo.

Existem vários gerenciadores de contenção, desde os mais simples, que apenas abortam a transação que detectou o conflito, até outros mais complexos, que medem quanto trabalho uma transação já realizou para decidir se aborta ou não, por exemplo.

A escolha entre os gerenciadores de contenção pode ser dependente do sistema de memória transacional, da carga de trabalho e do mecanismo de controle de concorrência usado pelo sistema (HARRIS et al., 2007). Por exemplo, se a TM usa detecção de conflitos adiantada, o gerenciador de contenção deve fazer com que a transação abortada espere o suficiente para que a outra possa ser efetivada.

2.2 Implementações de Memórias Transacionais

A seguir serão descritas algumas implementações de STM. Foram escolhidas estas por se tratarem do estado da arte da área. Logo após, será feita uma breve

comparação entre elas.

2.2.1 TL2

TL2 (do inglês: *Transactional Locking II*) (DICE; SHALEV; SHAVIT, 2006) é um algoritmo que utiliza detecção de conflitos tardia, versionamento de dados tardio e, além disto, utiliza um relógio global como mecanismo de versão para os dados compartilhados. Este relógio é denominado GV4 e é incrementado por cada transação confirmada.

No início de cada transação, o relógio é lido e copiado para a variável local rv (*read version*). Sempre que uma transação for efetivar, é feita a comparação entre o rv da *thread* e a versão dos dados acessados por ela. Se o rv for menor que a versão do dado, a transação será abortada, pois o dado foi modificado por outra transação.

A utilização do relógio global diminui o custo de validação, porém, se o número de *threads* for alto, o relógio pode se tornar um gargalo para o sistema, visto que maior será a contenção em relação ao incremento do relógio.

Além de utilizar um gerenciador de contenção tímido, TL2 também utiliza um mecanismo de *backoff* para atrasar a reexecução de transações que já abortaram muitas vezes. O tempo de atraso é incrementado a cada vez que a transação aborta.

2.2.2 TinySTM

TinySTM (FELBER; FETZER; RIEGEL, 2008) é baseada em um algoritmo com características muito similares às do TL2, chamado LSA (*Lazy Snapshot Algorithm*). A principal diferença é que ele realiza detecção de conflitos adiantada, evitando que transações condenadas executem até o final.

Esta implementação conta com duas politicas distintas para versionamento dos dados: *write-through*, ou seja, versionamento adiantado, com uso de *undo log*; e *write-back*, que se trata de versionamento tardio, com *buffer* de escrita.

2.2.3 SwissTM

SwissTM (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009) busca obter bom desempenho tanto em transações curtas e estruturas de dados pequenas, como em transações grandes e cargas de trabalho complexas. Para tal, é utilizada uma estratégia de detecção de conflitos mista, gerenciador de contenção, versionamento atrasado e relógio global para validação.

Como dito, a detecção de conflitos é mista, visto que busca conflitos do tipo escrita/escrita de forma adiantada, para prevenir que transações condenadas sigam executando até o final, e conflitos de leitura/escrita são detectados de forma atrasada, na esperança de que a outra transação conflitante seja abortada e o conflito não seja concretizado. Assim o paralelismo explorado é maior.

O gerenciador de contenção utiliza uma estratégia que simplesmente aborta a transação que detectou o conflito, quando se tratam de transações pequenas e/ou somente leitura. Já em transações maiores o sistema muda para uma estratégia gulosa, priorizando a transação mais antiga, evitando *starvation*. Para as transações detectadas de maneira adiantada, é utilizado um *backoff* proporcional ao número de vezes que esta foi cancelada.

2.2.4 adaptSTM

adaptSTM (PAYER; GROSS, 2011) é uma biblioteca com detecção de conflitos feita em nível de palavra e utilizando um array que combina versionamento e *locks*. Ela visa explorar as oportunidades de adaptação do programa em execução e dos parâmetros do sistema. Esta adaptação não é feita de forma global, como em outras implementações, e sim a nível de *threads*, o que traz algumas vantagens como o aumento da escalabilidade devido à diminuição do gargalo de comunicação.

Esta implementação realiza várias medições, em tempo de execução, de alguns parâmetros de cada *thread*. Alguns destes parâmetros são locais de escrita e leitura, e taxas de efetivações e abortos. Com os resultados destas medições é possível tomar as decisões referentes à adaptação do sistema. Alguns dos parâmetros adaptáveis são: tamanho do *hash* do *write-set*, função *hash*, estratégia de escrita e gerenciador de contenção.

2.2.5 CMTJava

CMTJava (DU BOIS; ECHEVARRIA, 2009; BANDEIRA, 2013) é uma extensão da linguagem Java para programação de memórias transacionais. Ela oferece a abstração de objetos transacionais. Os atributos de um objeto transacional podem ser acessados somente através de métodos especiais de *get* e *set*, gerados pelo compilador. O compilador também garante os atributos só poderão ser acessados dentro de transações. As transações devem ser executadas dentro do método *atomic*, que executa a transação que for passada por argumento e, com isto, garante-se as propriedades de atomicidade e isolamento.

Ela possui três implementações de sistemas transacionais. A primeira, baseada no sistema transacional da linguagem STM Haskell (HARRIS et al., 2005), possui implementação simples, já que as transações precisam adquirir um bloqueio global para validar o *log* e para efetivar a mesma, limitando o paralelismo. A segunda é baseada no algoritmo TL2, descrito anteriormente. Por não usar um bloqueio global, esta implementação consegue maior desempenho que a anterior. Já a terceira implementação é baseada no algoritmo da swissTM, utilizando uma estratégia mista para detecção de conflitos, identificando de maneira adiantada os conflitos entre transações de escrita, que geralmente causam aborto de uma das transações, evi-

tando que uma transação condenada ao aborto execute até o final. Por outro lado, verifica de maneira tardia os conflitos entre leitura e escrita, pois isto permite maior paralelismo na execução, uma vez que esse tipo de conflito pode muitas vezes ser resolvido sem cancelamentos, quando a transação de leitura é efetivada antes da transação de escrita.

2.2.6 Comparação das Implementações

As diversas implementações de sistemas de memórias transacionais vistas até aqui possuem muitas características em comum. Elas se diferenciam principalmente quanto ao tempo de versionamento, de detecção de conflito e ao tipo de gerenciador de contenção. Estas informações foram sintetizadas na Tabela 1.

Tabela 1: Comparação entre as implementações

	TL2	**TinySTM**	**SwissTM**	**adaptSTM**	**CMTJava**
Versionamento	tardio	ambos	tardio	implementa ambos	tardio
Grão	palavra	palavra	palavra	palavra	palavra
Detecção	tardia	adiantada	mista	adiantada	mista
Gerenciador de Contenção	tímido	tímido	tímido e guloso	tímido e *backoff* exponencial	tímido

Como foi visto, as implementações TinySTM, SwissTM e adaptSTM possuem opções que podem ser setadas estaticamente, antes da execução, ou de maneira automática, durante a execução. Devido a isto, alguns dos campos da Tabela 1 possuem mais de uma opção. Alguns trabalhos fazem comparações de desempenho entre estas implementações, como é o caso de (RICO; PILLA; DU BOIS, 2012), onde é utilizado o *benchmark* STAMP (do inglês: *Stanford Transactional Applications for Multi-Processing*) (CAO MINH et al., 2008) para avaliá-las.

3 MEMÓRIA TRANSACIONAL DISTRIBUÍDA

O foco principal deste trabalho são as memórias transacionais distribuídas, sendo assim, além das características de memórias transacionais, devem ser estudadas características de Sistemas Distribuídos. Estes sistemas possuem algumas diferenças arquiteturais consideráveis, se comparados com sistemas com memória compartilhada. Em seguida, serão comentadas duas diferenças que afetam diretamente a implementação de sistemas de memórias transacionais: comunicação e sincronização. Logo após, serão vistos alguns trabalhos relacionados.

3.1 Comunicação por Troca de Mensagens

Uma peculiaridade geralmente apresentada pelos sistemas distribuídos é a inexistência de um espaço de endereçamento único, visto que cada máquina possui sua própria memória. Existem mecanismos que simulam um único espaço de endereçamento, tanto em hardware quanto em software, como é o caso de DSM (*Distributed Shared Memory*) (COULOURIS; DOLLIMORE; KINDBERG, 2007), porém sem o uso de algum destes mecanismos, ou técnica similar, não é possível utilizar variáveis compartilhadas. Uma solução clássica para este dificultante é o uso de mecanismos de troca de mensagens. As bibliotecas para este tipo de comunicação geralmente oferecem as primitivas *send()* e *receive()*. A primeira envia uma mensagem para um processo destino e a segunda recebe uma mensagem de um processo remetente.

Existem várias semânticas possíveis para estas operações, por exemplo, elas podem ser bloqueantes ou não-bloqueantes. Um *send()* bloqueante faz com que o processo fique bloqueado até que a mensagem chegue ao destino, já o não-bloqueante permite que o processo continue executando enquanto é feito o envio. Um *receive()* bloqueante faz com que o processo fique bloqueado esperando uma mensagem, já o não bloqueante permite que este prossiga mesmo sem receber nada. Ambas as primitivas também podem ter nomeação explícita e implícita, ou seja, se elas devem indicar para quem enviar ou de quem receber, ou se podem enviar para todos os interessados

e receber de qualquer um, respectivamente (DU BOIS, 2008).

Troca de mensagens é geralmente utilizada para a comunicação de processos localizados em máquinas diferentes em uma rede, porém, nada impede que estes mecanismos sejam utilizados para comunicar processos localizados em uma mesma máquina. Existem várias formas de se utilizar troca de mensagens, desde as mais baixo nível, como *sockets*, até algumas de mais alto nível, como RPC (*Remote Procedure Call*), RMI (*Remote Method Invocation*) e MPI (*Message-Passing Interface*) (GROPP; LUSK; SKJELLUM, 1999).

3.2 Problema do Relógio Global

Como observado na Seção 2.2, em geral, as implementações de memórias transacionais utilizam um relógio global para auxiliar no versionamento dos dados, ordenando os eventos. Quando se trata de uma única máquina este processo é trivial, visto que dispõe-se do relógio físico da própria máquina. Porém, como foi já mostrado por LAMPORT (1978), não se pode ordenar totalmente eventos de um sistema distribuído utilizando relógios físicos das várias máquinas, visto que não é possível sincronizá-los perfeitamente.

Ele criou uma solução simples para este problema: o relógio lógico. Este relógio não precisa ter nenhum relacionamento com o relógio físico. A ideia é que cada processo mantenha um contador que é incrementado a cada evento local e, além disso, o valor deste relógio deve ser enviado junto com cada mensagem trocada pelos processos. O processo que recebe uma mensagem compara o valor presente nesta com o seu contador local, caso o valor recebido seja maior que o seu, ele incrementa este valor e o atribui a seu contador.

Desta forma, a ordem de precedência dos eventos é dada pelo valor destes contadores. Esta técnica apresenta a falha de permitir a existência de eventos, gerados por processos distintos, com a mesma indicação de tempo. Estes eventos são chamados de concorrentes. Entretanto, muitas vezes é necessária a existência de uma ordem total dos eventos. Uma técnica bastante utilizada é fazer o desempate baseado no identificador do processo. Isto não garante que a ordem obtida seja a real, pois os identificadores de processos são gerados de maneira arbitrária, porém, muitas vezes esta ordem é adequada para fins práticos.

3.3 Trabalhos Relacionados

Memórias transacionais para máquinas *multicore* têm sido bastante estudadas. A Seção 2.2 trouxe um resumo sobre algumas das implementações provenientes destes estudos. Porém existem poucos trabalhos focados em arquiteturas distribuídas, que é

o foco deste estudo. Sendo assim, são poucos os trabalhos diretamente relacionados a este. Informações sobre alguns destes trabalhos se encontram em (MANASSIEV; MIHAILESCU; AMZA, 2006), (BOCCHINO; ADVE; CHAMBERLAIN, 2008), (KOTSELIDIS et al., 2008), (SAAD; RAVINDRAN, 2012), (ZHENG, 2002) e (HERLIHY; CALCIU, 2011). A seguir serão descritos três destes trabalhos: Cluster-STM, DiSTM e TFA. Estes foram selecionados por apresentarem características mais interessantes para a fundamentação de trabalhos futuros do grupo.

3.3.1 Cluster-STM

Em (BOCCHINO; ADVE; CHAMBERLAIN, 2008) foi descrita a proposta de uma interface que contempla as funcionalidades necessárias para a utilização de memória transacional em *clusters* de larga escala. Os *clusters* em questão utilizam o conceito de PGAS (espaço de endereço global particionado), que consiste em um tipo de DSM onde tem-se um espaço de endereçamento de memória único, acessível por todos os nodos, porém particionado de forma que seja possível saber a qual nodo pertence cada posição de memória. Isto permite que se tire proveito da afinidade de cada processador com sua memória.

3.3.1.1 Interface Proposta

Foi proposta uma interface com alguns métodos para alocação de memória para os dados transacionais e manipulação destes pelas transações. Esta interface implementa atomicidade fraca, ou seja, não são verificados os conflitos entre dados transacionais e dados não transacionais. Isto se deve ao fato de o *overhead* causado por implementações de atomicidade forte ser proibitivo para *clusters*. Para evitar que um dado transacional seja alterado fora de uma transação, sem proteção, foi imposto que um dado deve ser acessado sempre em uma transação ou nunca dentro de uma. De maneira análoga, foi imposto que um objeto deve ser inteiramente transacional ou inteiramente não-transacional.

Além dos métodos básicos para utilização de transações, como *start, commit, read, write* e alocação de memória, a interface proposta prevê alguns outros, visando desempenho. Basicamente, são métodos para realizar o movimento de múltiplas palavras em uma única mensagem, evitando a granulosidade fina, e o método *On*, para execução remota de código, evitando o envio de dados muito grandes.

A Tabela 2 contém os métodos da interface proposta e uma descrição resumida de suas funções. Como pode ser visto, são contemplados métodos para alocação de memória transacional, controle da transação, movimento de dados, e execução remota de código. Vale ressaltar que pode-se escolher onde a memória será alocada, bem como mover dados em blocos e levar a execução até onde os dados se encontram. Estas características estão presentes devido a natureza distribuída dos *clusters*, para

reduzir os gastos com comunicação.

Tabela 2: Interface proposta em (BOCCHINO; ADVE; CHAMBERLAIN, 2008)

Método	**Descrição**
Alocação de Memória	
stm_all_alloc(work_proc, size)	Aloca uma área de memória de tamanho *size* no *work_proc* e retorna para todos os nodos um ponteiro.
stm_alloc(src_proc, work_proc, size)	Aloca uma área de memória de tamanho *size* no *src_proc* e retorna para *work_proc* um ponteiro.
stm_free(work_proc, addr)	Desaloca a memória de endereço *addr* do *work_proc*.
***Start* e *Commit* das Transações**	
stm_start(scr_proc)	Começa uma transação em nome de *scr_proc*.
stm_commit(scr_proc)	Efetiva uma transação em nome de *scr_proc*.
Movimento de Dados	
stm_open_read(src_proc, addr, size)	Em nome de *scr_proc*, abre *size* bytes para leitura, começando em *addr*.
stm_read(src_proc, dest, src, size)	Em nome de *scr_proc*, copia *size* bytes da memória transacional *src*, para a memória privada *dest*.
stm_open_write(src_proc, addr, size)	Em nome de *scr_proc*, abre *size* bytes para escrita, começando em *addr*.
stm_write(src_proc, dest, src, size)	Em nome de *scr_proc*, copia *size* bytes da memória privada *src*, para a memória transacional *dest*.
stm_get(src_proc, dest, work_proc, src, size, open)	Em nome de *scr_proc*, copia *size* bytes da memória transacional *src* do *proc_work* para a memória privada local *dest*. Se *open*==true, abre os dados de origem para ler primeiro.
stm_put(src_proc, work_proc, dest, src, size, open)	Em nome de *scr_proc*, copia *size* bytes da memória privada local *src* para a memória transacional *dest* do *proc_work*. Se *open*==true, abre os dados de destino para escrita primeiro.
Trabalho Remoto	
stm_on(src_proc, work_proc, function, arg_buf, arg_buf_size, result_buf, result_buf_size)	Em nome de *src_proc*, executa a função *function* no processador *work_proc*, usando os argumentos em *arg_buf* e escrevendo os resultado em *result_buf*.

3.3.1.2 Algoritmo

São várias as decisões que devem ser tomadas ao se implementar um algoritmo de memórias transacionais. A seguir serão destacadas algumas destas, em sua maioria baseada em trabalhos anteriores focados em máquinas com caches coerentes:

- **Sincronismo de leitura**: foram implementadas tanto a opção com bloqueio para a leitura quanto a com validação de leitura;

- **Sincronismo de escrita**: optaram por escrita bloqueante;
- **Versionamento de dados**: implementados tanto o modelo com *buffer* (versionamento tardio) quanto o com *log* (versionamento adiantado) das operações;
- **Detecção de conflitos**: implementado *write-time* (detecção adiantada) e *commit-time* (detecção tardia);
- **Granularidade da detecção de conflitos**: blocos contíguos de 2^n palavras, onde n é um parâmetro;
- **Garantia de progresso**: as transações em si já impossibilitam a ocorrência de *deadlocks*, porém contra *livelocks* não foi utilizada nenhuma estratégia;
- **Onde os metadados são gravados**: no descritor de transação distribuído pelos nodos.

Devido às combinações das variantes acima, poderiam existir oito versões de algoritmo, porém versionamento adiantado não pode ser combinado com detecção tardia, logo, restam 6 combinações. Segundo (BOCCHINO; ADVE; CHAMBERLAIN, 2008), não foram implementadas duas das combinações possíveis por falta de tempo. As versões implementadas e testadas foram as seguintes:

- **RL-EA-UL**: Leitura Bloqueante, Detecção Adiantada, Versionamento Adiantado (do inglês *Read Locking, Early Acquire, Undo Logging*);
- **RV-EA-UL**: Leitura Versionada, Detecção Adiantada, Versionamento Adiantado *Read versioning, Early Acquire, Undo Logging*;
- **RL-EA-WB**: Leitura Bloqueante, Detecção Adiantada, Versionamento Tardio *Read Locking, Early Acquire, Write Buffering*;
- **RL-LA-WB**: Leitura Bloqueante, Detecção Tardia, Versionamento Tardio *Read Locking, Late Acquire, Write Buffering*.

Maiores detalhes destas implementações podem ser vistas na tabela da Figura 8.

3.3.2 DiSTM

DiSTM (*Distributed Software Transactional Memory*) (KOTSELIDIS et al., 2008) foi projetada para a fácil prototipação de protocolos de coerência de memórias transacionais distribuídas sem nenhum tipo de DSM em software ou em hardware. Trabalha com granularidade a nível de objeto e pode executar mais de uma *thread* por processador, diferentemente da Cluster-STM. Três protocolos de coerência foram implementados: um protocolo decentralizado TCC (*Transactional Coherence and Consistency*) e dois protocolos centralizados que utilizam *leases*.

RL = *read locking*, EA = *early acquire*, UL = *undo logging*, RV = *read versioning*, WB = *write buffering*, *and* LA = *late acquire*.

Operation	RL-EA-UL	RV-EA-UL	RL-EA-WB	RL-LA-WB
	Memory Allocation			
`stm_alloc`	Call `memalign` to get an allocation aligned on a CDU boundary and clear associated metadata. If `tx_depth > 0`, record the allocation.			
`stm_all_alloc`	Same as `stm_alloc`, except never inside a transaction and the result is broadcast to all processors.			
`stm_free`	Call `free`			
	Transaction Start and Commit			
`stm_start`	Increment `tx_depth`. If `tx_depth == 1`, set `tx_proc` to the current processor and invoke `setjmp`.			
`stm_commit`	Decrement `tx_depth`. If `tx_depth == 0`, then attempt to commit the transaction:			
		Validate READ CDUs, aborting on failure. Increment version of WRITE CDUs.		Acquire WRITE CDUs, aborting on failure.
			Copy data from WRITE entries	
	Release all held locks			
	Data Movement			
`stm_open_read`	For each CDU touched that is not already in the dataset, create a new READ entry and do the following atomically with respect to the associated global metadata word, aborting if the word is locked for writing:			
	Increment reader count	Record version	Increment reader count	
`stm_read`	Copy data from transactional store to private store		Copy data from buffer if there, otherwise from store	
`stm_open_write`	For each CDU touched that does not already have a WRITE entry in the dataset, do the following atomically with respect to the associated global metadata word, aborting if the word is locked for writing:			
	Abort if reader count exceeds one, or equals one and CDU is not in read set	If a READ entry exists, validate it and abort on failure (since the transaction would later abort on commit anyway).	Abort if reader count exceeds one, or equals one and CDU is not in read set.	
			Get data from store into write buffer.	
	Set write bit of metadata word			
`stm_write`	Copy from private store to transactional store		Copy from private store to buffer	
`stm_get`	If `work_proc` is remote, add `work_proc` to `remote_procs`. Optionally invoke `stm_open_read` on `work_proc`, aborting on failure. Invoke `stm_read` on `work_proc`.			
`stm_put`	If `work_proc` is remote, add `work_proc` to `remote_procs`. Optionally invoke stm_open_write on `work_proc`, aborting on failure. Invoke `stm_write` on `work_proc`.			
	Remote Work			
`stm_on`	If `work_proc` is remote, and `tx_depth > 0`, add `work_proc` to `remote_procs`. Invoke `setjmp`. Invoke specified function on `work_proc`, aborting on failure.			

Figura 8: Características dos quatro algoritmos implementados (BOCCHINO; ADVE; CHAMBERLAIN, 2008)

Ela é escrita inteiramente em Java e possui dois módulos principais: um responsável pela execução transacional e outro responsável pela comunicação entre os diferentes nodos. Para a execução transacional, foi feita uma extensão do mecanismo DSTM2 (HERLIHY; LUCHANGCO; MOIR, 2006) e a comunicação foi baseada no ProActive Framework (BADUEL et al., 2006), uma API de alto nível que utiliza RMI. Maiores detalhes sobre a implementação serão visto seguir.

3.3.2.1 Mecanismo Transacional DSTM2

DSTM2 é um sistema de STM escrito em Java que suporta execução de transações em arquiteturas de memória compartilhada. Nele, todas as transações são executadas especulativamente. Quando uma transação tenta modificar um objeto, ela o faz em uma cópia deste e, após a efetivação, substitui o original pela sua cópia, caracterizando versionamento tardio. Antes da fase de efetivação, existe uma fase na qual alguns conflitos *write-after-write* e *read-after-write* são detectados e resolvidos. A decisão de como será resolvido o conflito é do gerenciador de contenção, que pode abortar ou atrasar uma ou mais das transações conflitantes. Depois disto, as transações podem ser efetivadas. DSTM2 utiliza uma politica de sincronização *obstruction-free*, porém esta politica não previne a ocorrência de *livelocks*.

Algumas modificações foram realizadas na arquitetura da DSTM2 para suportar as funcionalidades distribuídas. As duas principais foram a forma como as transações são efetivadas e a maneira como os objetos são identificados através dos nodos do

cluster. Originalmente, DSTM2 possui os passos de *validação()* e de *commit()*. DiSTM acrescenta um novo passo para garantir a coerência entre os nodos do *cluster*. Os protocolos utilizados para satisfazer este passo serão vistos em 3.3.2.2. Além disso, cada nodo mantém uma cópia dos dados transacionais e DiSTM é responsável por manter a coerência destes dados.

3.3.2.2 Protocolos de Coerência para Memórias Transacionais Distribuídas

Como dito, três foram os protocolos de coerência implementados. Um decentralizado equivalente ao TCC e dois baseados em *leases*. A seguir serão descritos estes protocolos.

TCC

TCC faz validação tardia das transações, quando estas desejam ser efetivadas. Cada transação envia por *broadcast* seus conjuntos de leitura e escrita, antes da fase de efetivação. Todas comparam concorrentemente seus conjuntos com o conjunto recebido. Se algum conflito for detectado, uma das transações conflitantes é abortada para a outra ser efetivada com segurança. TCC foi originalmente projetado para arquiteturas com coerência de cache, porém, foi adaptado para funcionar como protocolo de coerência de memórias transacionais descentralizadas na DiSTM.

Sempre que há um conflito, a transação mais antiga tem prioridade para ser efetivada. A ordem das transações é controlada através da distribuição de "*tickets*" para as transações. Esta distribuição é feita pelo nodo mestre quando uma transação envia seus conjuntos de leitura e escrita. Vale ressaltar que isto caracteriza um controle centralizado.

A Figura 9 mostra a sequência de passos das fases de validação e de efetivação. Como pode ser visto, depois de validar os conjuntos de leitura e escrita, a transação está pronta para ser efetivada, então ela torna visíveis localmente seus dados e enviá-os para o nodo mestre que, em seguida, envia estes dados para os outros nodos que atualizam suas caches locais. Neste último passo, se alguma transação tiver lido algum dado antes da atualização da cache, esta será executada novamente após a atualização.

Lease Único

Nesta estratégia, existe um único *lease* controlado pelo nodo mestre. A ideia é que somente uma transação possa ser efetivada por vez. Conforme Figura 10, sempre que uma transação está pronta para fazer a validação global, ela solicita o *lease* para o nodo mestre. Se não estiver em posse de outra transação, ela é bem sucedida e pode ser efetivada, caso contrario, entra em uma fila para obtenção do *lease*. Quando uma transação está sendo efetivada, os dados locais dos outros nodos são atualizados

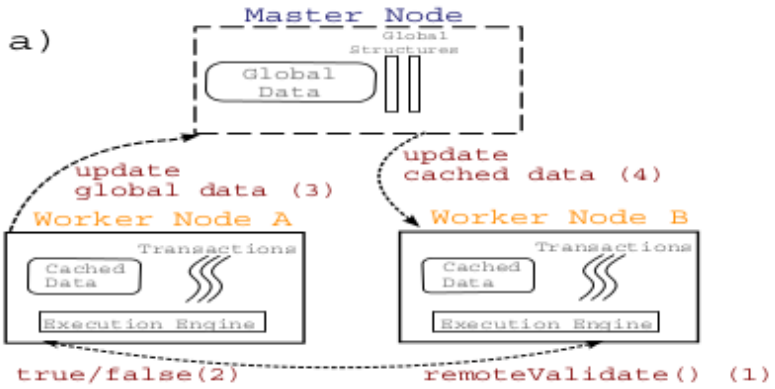

Figura 9: TCC (KOTSELIDIS et al., 2008)

e qualquer outra transação que já esteja na fila e apresente algum conflito é abortada. Logo após, o *lease* é liberado e a próxima transação pode adquiri-lo.

Este esquema de *lease* único apresenta a vantagem de não ocasionar o *overhead* causado pelo uso de *broadcast*, como no TCC. Porém, apresenta a desvantagem de introduzir um gargalo serial na fase de validação.

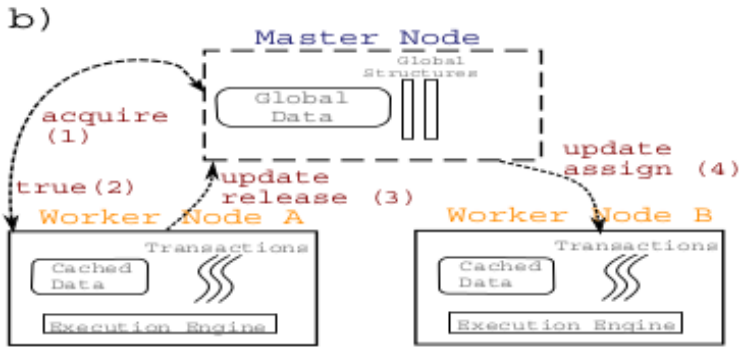

Figura 10: *Lease* Único (KOTSELIDIS et al., 2008)

Múltiplos Leases

Este esquema tenta alargar o gargalo causado pelo anterior. Nele exite um número ilimitado de *leases*. A Figura 11 ilustra o fluxograma deste esquema. Primeiramente, quando uma transação está pronta para ser efetivada globalmente, ela adquire um *lease*, porém, se outra transação tenta ser efetivada ao mesmo tempo, ela adquire outro *lease* e assim por diante. Isto causa indispensável a adição de um novo passo de validação, pois várias transações podem estar sendo efetivadas ao mesmo tempo. Então, sempre que uma transação adquire um *lease*, é feita a validação do seu conjunto de leitura e de escrita com os de todas as outras que já possuem *leases*. Se houver algum conflito, a mais nova é abortada. Após esta fase é realizada a atualização dos dados globais e, logo após, os dados locais de cada nodo são atualizados.

A vantagem deste modelo sobre o anterior é a possibilidade de diversas transações serem efetivadas ao mesmo tempo. Em contrapartida, é adicionado um novo gargalo no nodo mestre, que deve fazer a validação das transações que estão sendo efetivadas simultaneamente.

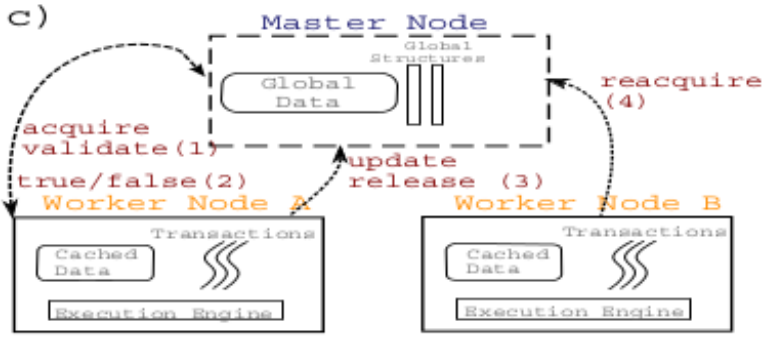

Figura 11: Múltiplos *Leases* (KOTSELIDIS et al., 2008)

3.3.3 TFA

TFA (*Transactional Forwarding Algorithm*) (SAAD; RAVINDRAN, 2012) assegura as propriedades transacionais de atomicidade, consistência e isolamento para transações distribuídas e segue o modelo de fluxo de dados (SAAD; RAVINDRAN, 2012). Cada nodo mantem um relógio local que avança assincronamente a cada efetivação de transação local. É empregada validação adiantada e neste momento os relógios são ajustados. TFA permite que transações não-conflitantes sejam efetivadas ao mesmo tempo e que várias transações executem no mesmo nodo, concorrentemente. Os bloqueios dos objetos são somente adquiridos no momento da efetivação, caracterizando detecção de conflitos tardia.

TFA foi implementado sobre o *framework* HyFlow DTM (SAAD; RAVINDRAN, 2011), é completamente descentralizado e não faz *broadcast* de mensagens. Nas próximas seções encontram-se maiores detalhes sobre o modelo proposto e o algoritmo.

3.3.3.1 Modelo

Foi considerado um sistema distribuído assíncrono, como em (HERLIHY; SUM, 2005), formado por N nodos, $N_1, N_2, ..., N_N$, conectados usando *links message-passing FIFO* ou sobre uma rede *overlay*. Cada objeto tem um identificador único e é criado ligado a um nodo *home*, porém pode ser replicado ou migrado para um nodo qualquer. TFA é responsável por fazer as réplicas e por alterar a posse dos objetos.

Transações são imóveis e cada transação é associada a um nodo. O nodo

N_x executa uma transação T. Sendo T uma sequência de operações em objetos $O_1, O_2, ..., O_S$, onde $S >= 1$. É assumido que a maior parte das transações são concorrentes. Quando uma transação tenta acessar um objeto, uma copia deste é movida para seu nodo. A troca de dono só ocorre quando uma transação é efetivada. Cada transação possui um conjunto de leitura e um de escrita, compostos pelos objetos que são acessados por ela.

Cada nodo possui um relógio local, que é incrementado cada vez que uma transação é efetivada. Já que uma transação roda em um único nodo, ele é utilizado para gerar o *timestamp*, *write version (wv)*, durante a efetivação. Cada mensagem carrega consigo este relógio e o nodo receptor compara-o com o seu relógio, assumindo o maior deles como relógio atual.

3.3.3.2 Algoritmo

O problema de localização dos objetos está fora do escopo do trabalho. Pode ser usado qualquer diretório ou protocolo de coerência de cache para isto. Foi assumido a existência de um gerenciador de diretório com dois métodos: $publish(x, N_C)$, que define que o nodo N_c é o dono do objeto x, e $locate(x)$, que identifica e retorna o nodo dono do objeto x.

Cada grão da memória transacional (atributo, objeto ou outros, de acordo com a granularidade desejada) é associado a um *lock* de escrita versionado, onde o primeiro bit indica se o *lock* está acionado e o restante indica a versão. A versão é alterada a cada efetivação bem sucedida. Cada nodo possui um relógio local que é lido pelas transações, quando são inicializadas. Posteriormente, a transação só poderá ser efetivada se todos os objetos lidos tiverem versões menores do que a sua, o que indica que nenhum foi alterado por alguma transação concorrente. Quando uma transação é efetivada, ela avança o relógio do nodo e utiliza-o para atualizar a versão do objeto.

Como os relógios são avançados assincronamente, não seria possível comparar o relógio de uma transação com a versão de um objeto que foi modificado por outro nodo. Para resolver isto, foi desenvolvida uma técnica que em certas ocasiões, adianta o relógio da transação de acordo com a versão de algum objeto remoto, checando possíveis conflitos futuros. Além disto, os relógios são atualizados a cada comunicação, resolvendo o problema dos relógios distribuídos.

Quando uma transação é inicializada, ela lê o valor do relógio do nodo em que está sendo executada e o chama de *"write version"* (wv), como visto na Figura 12. Durante a execução, a transação mantém um conjunto de leitura e um de escrita.

Um objeto acessado por uma transação pode ser local ou remoto. Para objetos locais, é verificado se $obj.version < wv$, caso contrário, houve um conflito. Para objetos remotos, é criada uma cópia local e as modificações são realizadas nela. Além disto, devido à comparação entre relógios de diferentes nodos, é necessária uma etapa

Require: Transaction $trans$, Node $node$
Ensure: Initialize transaction.

1: trans.node = node
2: trans.wv = node.clock

Figura 12: Transaction::Init (SAAD; RAVINDRAN, 2012)

adicional, chamada *"transactional forwarding"*.

Transactional Forwarding: É utilizado para lidar com os relógios assíncronos de diferentes nodos. Consiste em alterar o valor de wv para o valor do relógio do dono de um objeto lido, caso $trans.wv < obj.owner.clock$. Para fazer isto, é necessário checar se algum objeto do conjunto de leitura foi alterado desde o começo da transação. Isto é feito testando, para todos eles, se $obj.version > trans.wv$. Caso positivo, a transação é abortada, caso contrario, é $trans.wv = obj.owner.version$.

Nas Figuras 13 e 14 pode-se ver o funcionamento do *transactional forwarding*. Quando uma transação deseja abrir um objeto remoto, ela envia uma mensagem solicitando o objeto para o nodo dono do objeto, esta mensagem também contem o relógio do nodo da transação. O dono do objeto recebe a mensagem e compara seu relógio com o da mensagem, caso este seja maior, atualiza o seu. Ele envia o objeto para o nodo requisitor, também acrescentando seu relógio à mensagem. O requisitor abre o objeto, testa se precisa atualizar seu relógio, testa se precisa avançar $trans.wv$ e faz a validação.

Antes de uma transação ser efetivada, é necessário verificar se ela tem uma visão consistente dos dados (Figura 15). Para isto devem ser realizados os seguintes passos:

1. Adquirir os bloqueios dos objetos presentes no conjunto de escrita em uma ordem que não gere *deadlock*. Se algum objeto for remoto, é enviada uma mensagem solicitando que o seu dono adquira o bloqueio. Se algum dos bloqueios não puder ser adquirido, a transação é abortada e reinicializada.

2. Revalidar o conjunto de leitura para ter certeza de que a visão dos dados é consistente. Se este passo for bem sucedido, a transação está pronta para ser efetivada.

3. Incrementa o relógio local e o escreve nas variáveis de versão dos objetos que tem bloqueios foram adquiridos.

4. Libera os bloqueios e publica no diretório que é o novo dono dos objetos remotos.

Se uma transação for abortada, os bloqueios também devem ser liberados.

```
Require: Transaction trans, ObjectID id
Ensure: Open shared object for current transaction.
 1: Node owner = findObjectOwner(id)
 2: Object obj = node.RetrieveObject(trans.node, id)
 3: if obj.remote then
 4:    if trans.node.clock < obj.owner.clock then
 5:       trans.node.clock = obj.owner.clock
 6:    end if
 7:    if trans.wv < obj.owner.clock then
 8:       for all obj in transaction.readSet do
 9:          if obj.version > trans.wv then
10:             return rollback()
11:          end if
12:       end for
13:       trans.wv = obj.owner.clock
14:    end if
15: else
16:    if obj.version > trans.wv then
17:       return rollback()
18:    end if
19: end if
20: return obj
```

Figura 13: DirectoryMgr::OpenTransactional (SAAD; RAVINDRAN, 2012)

```
Require: Node requester, ObjectId id
Ensure: Send a copy of object identified by given id and owned
    by current node to the requester node.
1: if this.clock < requester.clock then
2:    this.clock = requester.clock
3: end if
4: return LocalObjects.get(id)
```

Figura 14: Node::RetrieveObject (SAAD; RAVINDRAN, 2012)

3.3.4 Comparação entre os trabalhos relacionados

Nas Subseções anteriores (3.3.1, 3.3.2 e 3.3.3) foram apresentados três trabalhos relacionados, todos provendo suporte à memória transacional distribuída. A seguir, se encontra uma comparação das principais estratégias de implementação utilizada por cada um deles. Na Tabela 3 encontra-se um resumo desta comparação.

Versionamento de Dados

DiSTM e TFA utilizam a estratégia de versionamento tardio, ou seja, escrevem em uma cópia dos dados. Na efetivação, TFA possui a peculiaridade de simplesmente publicar que o dado está em um novo local, ao invés de copiar os dados para o endereço verdadeiro dos dados. Cluster-STM implementa ambas as estratégias, ficando a cargo

```
Require: Transaction trans
Ensure: Commit transaction if valid and rollback otherwise.
 1: for all obj in transaction.writeSet do
 2:     obj.acquireLock()
 3: end for
 4: for all obj in transaction.readSet do
 5:     if obj.version > trans.wv then
 6:         return rollback()
 7:     end if
 8: end for
 9: trans.node.clock ++
10: for all obj in transaction.writeSet do
11:     obj.commitValue()
12:     obj.setVersion(trans.node.clock)
13:     obj.releaseLock()
14:     if obj.remote then
15:         DirectoryManager.setOwner(obj, trans)
16:     end if
17: end for
```

Figura 15: Transaction::Commit (SAAD; RAVINDRAN, 2012)

Tabela 3: Comparação entre os trabalhos relacionados

	Cluster-STM	**DiSTM**	**TFA**
Versionamento	Ambos	Tardio	Tardio
Grão	Bloco de Palavras	Objeto	Objeto
Detecção	Ambos	Tardia	Tardia
Endereçamento	PGAS (um tipo de DSM)	Sem DSM	Sem DSM
Modelo de Exec.	Misto	Fluxo de Controle	Fluxo de Dados
Controle	Distribuído	Centralizado ou Parc. distribuído	Distribuído

do usuário escolher qual utilizar.

Tamanho do Grão

Cluster-STM utiliza grão de bloco de palavras e este possui tamanho definido pelo usuário. Já DiSTM e TFA mantém um metadado para cada objeto. Ambas as estratégias podem gerar falsos conflitos, conforme visto na Seção 2.1.3.

Detecção de Conflitos

A detecção de conflitos também é tardia em DiSTM e em TFA. Porém, esta realiza uma pré-validação a cada vez que precisa utilizar a técnica de *transactional forwarding* (como visto em 3.3.3.2). Já Cluster-STM novamente implementa tanto a estratégia atrasada quanto a adiantada.

Tipo de Endereçamento

Esta característica diz respeito a como cada modelo enxerga a memória distribuída. Cluster-STM utiliza o modelo de PGAS, um tipo de DSM que possui algumas características especiais. O principal diferencial de PGAS é dividir a memória em 4 tipos: local do nodo, compartilhada do nodo, local de outro nodo e compartilhada de outro nodo. Com isto, isto facilita para que se possa alocar memória levando em conta os tempos de latência da rede. DiSTM e TFA não utilizam qualquer camada de DSM. DiSTM faz a comunicação utilizando ActiveObjects e TFA localiza os objetos através de algum diretório e o move para o nodo local.

Modelo de Execução Distribuída

O modelo de execução distribuída diz respeito à como se da a interação entre instruções e dados distribuídos. DiSTM utiliza o modelo de fluxo de controle, ou seja, os dados ficam imóveis e as transações trocam principalmente mensagens de controle. Cluster-STM também utiliza este modelo, porém, também possibilita que se mova código executável entre os nodos, levando a computação para perto dos dados. Já TFA, utiliza o modelo de fluxo de dados, movendo os objetos para o nodo onde está a transação que utilizará seus dados. Este modelo pode levar vantagens quando são utilizados dados pequenos, porém, com dados muito grandes, pode gerar muito trafego de rede.

Estratégia de Controle

Cluster-STM e TFA não apresentam controle central, sendo totalmente distribuídas, o que as possibilita, a priori, maior escalabilidade. Já DiSTM implementa três protocolos de coerência, dois deles totalmente centralizados e o terceiro parcialmente distribuído, apenas com um passo centralizado (distribuição de *tickets*). Esta centralização pode comprometer a escalabilidade.

4 DCMTJAVA

Este trabalho propõe a linguagem de domínio específico DCMTJava, uma linguagem com suporte à memória transacional distribuída embutida em Java. Ela estende a linguagem CMTJava (Subseção 2.2.5), que contempla apenas transações locais, de forma a acrescentar suporte à transações envolvendo objetos distribuídos. O modo como ela trata as transações locais, bem como sua sintaxe e estruturas do sistema transacional são fortemente baseados na Linguagem CMTJava. Já o algoritmo utilizado para tratar as transações distribuídas é baseado no TFA (Subseção 3.3.3). Este algoritmo move os objetos entre os nodos, ao invés de mover instruções de controle, na tentativa de diminuir o *overhead* e amenizar os efeitos da latência de comunicação.

O algoritmo TFA foi escolhido, em detrimento aos algoritmos utilizados nos outros trabalhos relacionados pois, como foi visto no capítulo anterior (Tabela 3), ele é totalmente distribuído e não faz uso de nenhum tipo de DSM, diferindo-se assim de ambos os outros dois trabalhos relacionados, já que Cluster-STM (Subseção 3.3.1) funciona sobre PGAS e DiSTM (Subseção 3.3.1) não é totalmente distribuído. Além disso, TFA utiliza estruturas de *locks* versionados que também são utilizados na CMTJava, o que facilita a implementação.

Sendo assim, DCMTJava permite a execução de transações distribuídas, movendo cópias dos objetos envolvidos para o nodo em que a transação está localizada. A classe de um objetos que pode ser movidos pela rede é marcada para que o compilador as reconheça e gere seus métodos especiais. Estes métodos têm como objetivo realizar a comunicação dos objetos e a verificação de conflitos entre os nodos. Além disso, o sistema transacional da linguagem CMTJava foi estendido com operações para comunicação dos metadados das transações e com um sistema de diretório distribuído simples para localização e envio de objetos. Nas seções seguintes serão vistos alguns conceitos gerais utilizados na implementação, bem como a definição da linguagem e sobre o sistema transacional distribuído.

4.1 Definição da Linguagem

A linguagem DCMTJava oferece ao programador abstrações de alto nível para o uso de memórias transacionais. Estas abstrações contemplam tanto transações locais quanto distribuídas, deixando transparentes ao programador o sistema transacional e a troca de mensagem entre os nodos. Esta seção descreve a linguagem, começando por características herdadas da CMTJava e, posteriormente, abordando detalhes sobre as construções criadas para o uso de objetos em transações distribuídas.

4.1.1 Características Herdadas da CMTJava

Esta subseção traz algumas características provenientes da CMTJava que são importantes para o entendimento da DCMTJava.

4.1.1.1 TObjects

TObjects são os objetos transacionais, baseados no conceito de TVar da STM Haskell. O programador deve implementar a interface TObject nas classes que terão seus objetos protegidos pelo sistema transacional. Esta interface alerta o compilador, que criará todo o código necessário para que este objeto seja acessado pelo sistema transacional. Parte desse código gerado são os métodos `get` e `set`, que são as únicas formas para se ler e escrever nos atributos desses objetos. Estes métodos retornam ações transacionais, sendo assim, devem ser utilizados dentro de blocos `STM`$\{\ldots\}$, como será visto em seguida.

A figura 16 mostra a criação de uma conta bancária que implementa TObject, portanto será protegida pelo sistema transacional. Como pode ser visto, este código não difere em nada da criação de uma classe em Java, no entanto, o compilador perceberá que ela implementa a interface TObject e gerará o código necessário para que os objetos desta classe sejam acessados pelo sistema transacional.

A principal vantagem desta abordagem é a separação entre o código da aplicação e de controle de sincronização das transações. Ou seja, o código paralelo utilizando memórias transacionais fica muito semelhante ao código sequencial, pois detalhes da sincronização da execução não estão presentes no código definido pelo desenvolvedor (BANDEIRA, 2013).

4.1.1.2 Blocos STMDO

`STMDO`$\{\ldots\}$ é uma implementação da notação `do` da linguagem Haskell. Com o uso de `STMDO`$\{a_1; ...; a_n\}$ se constrói uma ação STM que une pequenas ações transacionais $a_1; ...; a_n$ em sequência. O conjunto de operações componentes é executado de forma atômica e isolada das demais transações do sistema (BANDEIRA, 2013).

Na Figura 17 pode ser visto como é realizada a criação de métodos que utilizam

```
1  package test.Account;
2  import stm.*;
3
4  class TAccount implements TObject {
5      private volatile Double balance;
6      private volatile Integer ops;
7
8      TAccount(){
9          this(0.0d);
10     }
11
12     TAccount(Double b){
13         balance = b;
14         ops = 0;
15     }
16 }
```

Figura 16: Exemplo da criação de uma classe que implementa TObject

atributos de TObjects. O método `deposit` pega o valor atual do atributo `balance`, soma o valor a ser depositado e o escreve novamente. Além disso, incrementa o valor de `ops`.

Nota-se que o método retorna uma ação STM, que é criada utilizando a notação `STMDO`. Dentro do bloco, são descritos os comandos que farão parte da ação STM, ordenadamente. O código é muito similar ao da linguagem Java, porém as variáveis criadas dentro de blocos STMDO são variáveis de atribuição única, por isso utiliza-se um símbolo diferente `(<-)` para atribuições envolvendo estas variáveis. Estes são parte dos métodos gerados automaticamente pelo compilador, tendo como tipo de retorno `STM<Double>` e `STM<Integer>`, respectivamente. Os métodos `setBalance` e `setOps` também são gerados pelo compilador e são utilizados para escrever nos atributos do objeto.

```
1  public STM<stm.Void> deposit (Double n) {
2      return new STMDO{
3          Double balance <- this.getBalance();
4          this.setBalance(balance + n);
5          Integer ops <- this.getOps();
6          this.setOps(ops+1)
7      };
8  }
```

Figura 17: Criação de um método utilizando a notação STMDO

4.1.1.3 Método Atomic

Para executar uma transação, é necessário o uso do método `atomic`. Ele recebe como argumento uma transação e a executa atomicamente, respeitando outras chamadas de `atomic` que estão sendo executadas concorrentemente. Um exemplo de uma transação sendo executada com o uso do método `atomic` pode ser visto na Figura 18.

```
1 atomic( account.deposit(VALOR) );
```

Figura 18: Execução de uma transação utilizando o método atomic

4.1.2 RemoteTObjects

DCMTJava estende a CMTJava com RemoteTObjects. De maneira análoga aos TObjects, RemoteTObjects também são objetos transacionais, com a diferença de que estes podem fazer parte de transações distribuídas. Além dos métodos `get` e `set`, o compilador gera outros métodos para as classes que implementam esta interface, porém estes métodos adicionais não são acessados diretamente pelo programador, e sim pelo sistema transacional distribuído. Maiores detalhes sobre o sistema transacional serão vistos na Seção 4.3.

A Figura 19 traz uma classe que implementa `RemoteTAcount`. Sendo assim, os objetos desta classe, além de serem acessíveis ao sistema transacional, também podem fazer parte de transações distribuídas, sendo acessados por outros nodos.

```
1 class RemoteTAccount implements RemoteTObject {
2     ...
3 }
```

Figura 19: Exemplo da criação de uma classe que implementa RemoteTObject

4.1.3 Diretório Distribuído

O sistema transacional da DCMTJava possui um diretório distribuído que deve rodar em todas as máquinas e tem como funções principais o registro e localização e envio dos RemoteTObjects através da rede. Ele possui dois métodos que são acessíveis ao programador: `register` e `openObject`.

4.1.3.1 Método Register

O método `void register(String id, RemoteTObject obj)` é utilizado para registrar um `RemoteTObject` no diretório, tornando-o visível para os demais nodos e, assim, permitindo que estes o localizem para realizar transações distribuídas. Além do

`RemoteTObject` a ser registrado, o método `register` também tem como entrada uma String, que será utilizada como identificador deste objeto.

A Figura 20 mostra como é feita a criação e registro de um RemoteTObject. No exemplo, o objeto account será registrado no diretório com o ID ”acc1”. Através deste ID, qualquer nodo é capaz de realizar transações que envolvam este objeto, usando o método openObject para tal.

```
1 Account account = new Account();
2 register("acc1", account);
```

Figura 20: Exemplo da criação e registro de um RemoteTObject

4.1.3.2 Método OpenObject

O método `RemoteTObject openObject(String id)` tem como entrada uma String e retorna um RemoteTObject. Ele localiza em que nodo está o RemoteTObject que possui identificador igual à `id` e baixa uma referência transacional deste. Este método deve ser chamado antes do primeiro uso de um RemoteTObject, criado em outro nodo, e só pode ser usado para objetos já registrados com o método register, em qualquer nodo. Depois que um RemoteTObject é registrado, referencias para ele podem ser baixadas quantas vezes forem necessárias.

A Figura 21 mostra um exemplo do uso do método `openObject`. Ele faz download e retorna um RemoteTObject do tipo Account. Depois de feito isso, o objeto pode ser utilizado em transações como qualquer TObject criado neste nodo.

```
1 Account account = (Account)openObject("acc1");
```

Figura 21: Exemplo de um RemoteTObject sendo aberto em outro nodo

4.2 Conceitos Gerais da Implementação

Esta seção traz os conceitos gerais da implementação da DCMTJava. Vale ressaltar que ela é implementada inteiramente em software, na linguagem Java. Para tal, são utilizados os conceitos que serão vistos a seguir.

4.2.1 Closures

Um closure, também conhecido como função anônima ou função lambda, é uma função que pode usar variáveis que estão dentro de um escopo, mesmo que este não esteja ativo no momento de sua chamada. Por exemplo, se um closure é passado como argumento para um método, ele vai continuar usando as variáveis do escopo onde ele foi criado.

BGGA Closures é uma extensão Java que suporta closures (JAVA CLOSURES, 2009). Usando BGGA, uma função anônima pode ser definida usando a sintaxe:

```
{ parâmetros formais => expressão }
```

Onde tanto os parâmetros formais quanto as expressão são opcionais. Por exemplo:

```
{ int x => x + 1 }
```

A função definida acima recebe um inteiro e retorna seu valor incrementado de um. Para chamar uma função anônima usamos o método `invoke`, por exemplo:

```
String s = { => "Hello!"}.invoke();
```

Uma função anônima também pode ser referenciada por variáveis. A variável deve possuir o mesmo tipo da função, por exemplo:

```
{int => void} func = {int x => System.out.println(x)};
```

4.2.2 Mônada STM

Mônadas (WADLER, 1995) são uma forma de estruturar computações em termos de valores e sequências de computações usando esses valores. A mônada é usada para descrever computações que podem ser combinadas formando novas computações (NEWBERN, 2013). O conceito de mônadas popularizou-se pelo trabalho de Eugenio Moggi (MOGGI, 1989), que buscou na teoria das categorias uma forma de estruturar a descrição semântica de categorias como estado, exceção e continuação, para linguagens funcionais.

A mônada para ações STM é usada para passar um estado pelas computações, onde cada computação retorna uma cópia alterada desse estado. Este estado é representado pelos metadados associados à transação, ou seja, seus *logs*. A mônada para transações da CMTJava é similar a utilizada no STM Haskell.

A mônada STM é implementada como um tipo abstrato de dados que representa um contêiner para uma computação. Essas computações podem ser criadas e compostas usando três operações básicas: `bind`, `then` e `return` (BANDEIRA, 2013). Para uma mônada qualquer `m`, essas funções tem o seguinte tipo em Haskell:

```
bind ::  m a -> (a -> m b) -> m b
then ::  m a -> m b -> m b
return ::  a -> m a
```

Onde:

- `m a` é um tipo que representa uma computação da mônada `m` que quando executada produzirá um valor do tipo `a`.
- `bind` executa seu primeiro argumento e passa o resultado para seu segundo argumento (uma função) produzindo uma nova computação.
- `then` recebe duas computações como argumento e produz uma computação que irá executá-las em ordem.
- `return` cria uma nova computação para um simples valor.

4.2.3 Compilação

O compilador DCMTJava recebe códigos escritos na linguagem e gera *bytecode* para ser executado na máquina virtual Java, utilizando Java+closures como linguagem intermediária. Ele é dividido em tradução e síntese. A tradução realiza transformações no código e insere métodos e atributos que devem ser criados pelo compilador. Já a fase de síntese gera código executável, a partir do código intermediário.

A etapa de tradução se encarrega de algumas tarefas, sendo as principais:

- analises léxica e sintática;
- geração dos métodos `get`, `set` e de estruturas de dados adicionais utilizadas pelo sistema transacional;
- geração dos métodos de serialização e outros métodos necessários para o sistema transacional;
- tradução dos blocos STMDO.

Esta etapa é realizada com o auxilio da ferramenta ANTLR (*ANother Tool for Language Recognition*) (PARR, 2007), um gerador de parsers que pode ser usado para implementar interpretadores, compiladores e outros tradutores.

Os métodos `get`, `set` e demais estruturas utilizadas pelo sistema transacional têm grande parte de seus códigos intimamente ligada ao sistema transacional utilizado. O mesmo ocorre com os métodos de serialização e outros métodos gerados para os TObjects e RemoteTObjects. Sendo assim, a geração destes métodos deve levar em conta a implementação do sistema transacional. Detalhes dos mesmos serão descritos na seção seguinte.

Os métodos STMDO são traduzidos para chamadas de *bind* e *then* da mônada STM. Como exemplo, pode-se ver na Figura 22 o método deposit, traduzido para o código intermediário. Este método foi apresentado escrito em CMTJava na Figura17.

```
1 public STM<stm.Void> deposit (Double n) {
2     return  STMRTS.bind( this.getBalance(),{ Double balance =>
3             STMRTS.then( this.setBalance(balance + n),
4             STMRTS.bind( this.getOps(),{ Integer ops =>
5             this.setOps(ops+1)  }) ) });
6 }
```

Figura 22: Método criado com STMDO, traduzido para código intermediário

Atualmente a etapa de tradução não está completa, principalmente no que diz respeito aos métodos de serialização, sendo assim, a inserção destes métodos e algumas outras pequenas alterações devem ser realizadas manualmente no código intermediário. A conclusão da implementação do compilador é um dos trabalhos futuros.

4.3 Sistema Transacional Distribuído

O sistema transacional distribuído da DCMTJava é encarregado de garantir a atomicidade e isolamento das transações, bem como, realizar toda a comunicação necessária entre os nodos participantes. Como foi descrito anteriormente, a estratégia utilizada para sua implementação foi baseada no algoritmo TFA (SAAD; RAVINDRAN, 2012), que foi apresentado nos trabalhos relacionados (Seção 3.3.3).

4.3.1 Estratégias de implementação do Sistema Transacional Distribuído

Como foi visto na Seção 2.1, existem algumas estratégias de implementação com as quais pode-se classificar os sistemas transacionais, são elas: detecção de conflito, versionamento de dados, granularidade de conflitos e gerenciamento de contenção. Posteriormente, na Seção 3.3.4, os trabalhos relacionados foram classificados conforme fazem uso dessas estratégias. Além disso na comparação dos trabalhos relacionados, surgiram outros três quesitos para comparação, exclusivos para implementações distribuídas, são eles: tipo de endereçamento, modelo de execução distribuída e estratégia de controle. Esta seção traz a classificação do sistema transacional implementado neste trabalho, conforme os quesitos citados.

Como foi dito, DCMTJava suporta tanto transações locais quanto distribuídas, na implementação atual do sistema transacional, os dois tipos de transação utilizam algumas estratégias distintas. Para transações locais é utilizado o mesmo algoritmo que a CMTJava faz uso, já para as transações distribuídas, o algoritmo usado é baseado no TFA. Devido a esta peculiaridade, a classificação do sistema transacional diferente para os dois tipos de transação, pois em alguns quesitos elas diferem, como pode ser visto na Tabela 4.

Tabela 4: Características do Sistema Transacional Implementado

	Transações Locais	Transações Distribuídas
Versionamento	tardio	tardio
Grão	palavra	objeto
Detecção	mista	tardia
Gerenciador de Contenção	tímido	tímido
Endereçamento	x	sem DSM
Modelo de Execução	x	fluxo de dados
Controle	x	distribuído

Versionamento de Dados

Tanto para os objetos locais quanto para os distribuídos, é utilizada a estratégia de versionamento tardio, ou seja, os dados só são escritos em suas posições reais na efetivação. Porém, para os objetos distribuídos, é importante ressaltar que após atualizar os dados localmente, o nodo simplesmente publica que é o novo dono do objeto, ao invés de copiar os dados para o endereço de origem.

Tamanho do Grão

Em objetos locais é utilizado o grão de palavra, ou seja, um metadado para cada atributo do objecto. Já para objetos distribuídos é utilizado o grão de objeto. Esta decisão se deu pois implementações com grão de objeto apresentam menor *overhead* na detecção de conflitos, visto que apenas um metadado deve ser verificado por objeto. Isto se torna mais importante nas transações envolvendo objetos distribuídos, já que a verificação destes metadados deve utilizar a rede. Além disso, o sistema transacional não tem conhecimento de quais atributos serão acessados em outro nodo, sendo assim enviar todo o objeto foi a alternativa escolhida.

Detecção de Conflitos

Para objetos locais é utilizada uma estratégia mista para detecção de conflitos, onde conflitos de escrita-escrita são identificados de forma adiantada. Esta estratégia evita que transações condenadas a abortar executem até o final. Já os conflitos de leitura-escrita de forma tardia, pois estes podem não se concretizar, caso a transação de leitura efetive antes da de escrita ou cado uma das transações conflitantes seja abortada por outra causa. Por outro lado, transações envolvendo objetos distribuídos utilizam detecção de conflitos tardia, para evitar comunicação excessiva durante a execução da transação.

Gerenciador de Contenção

Em todos os tipos de transação, ou seja, em transações somente envolvendo objetos locais, somente envolvendo objetos distribuídos ou envolvendo ambos os tipos de objetos, é utilizado o gerenciador de contenção tímido, abortando a transação que detectou o conflito.

Tipo de Endereçamento

DCMTJava não utiliza nenhuma forma de DSM, deixando a localização dos objetos remotos para o sistema de diretório distribuído pelos nodos.

Modelo de Execução Distribuída

O modelo de execução distribuída utilizado pela implementação atual da DCMTJava é o modelo de fluxo de dados, já que os objetos são movidos para o nodo onde está a transação que utilizará seus dados.

Estratégia de Controle

O sistema transacional implementado é totalmente distribuído, o que as possibilita, a priori, maior escalabilidade.

4.3.2 Características

Na implementação atual, os objetos são copiados através dos nodos. Um objeto é criado em um nodo e, então, é registrado no diretório associado a um identificador único. Quando outro nodo deseja realizar uma transação utilizando este objeto, ele solicita-o ao diretório, que se encarrega de criar uma réplica local e retorná-la para ser utilizada pela transação. Se a transação for bem sucedida, será anunciado ao diretório que a partir deste momento o nodo da transação é o novo dono do objeto.

As transações são imóveis, ao contrário dos objetos, e cada nodo pode executar diversas delas concorrentemente. Cada transação possui um conjunto de leitura e um de escrita, composto pelos metadados de cada atributo acessado. O compilador cria um atributo `fieldInfo` para cada atributo dos TObjects e dos RemoteTObjects. Estes fieldInfos possuem um *lock* versionado de escrita e um de leitura, uma *closure* que atualiza o valor do atributo em caso de *commit*, entre outras informações. O *lock* versionado de leitura, quando liberado, possui o número de versão atual do atributo e, quando bloqueado, possui o ID da transação que o bloqueou.

Cada nodo possui um relógio local. Este relógio é incrementado a cada *commit* e é utilizado como nova versão dos atributos utilizados pela transação. Quando ocorre envio de objetos através dos nodos, os seus relógios são ajustados de maneira similar ao que ocorre no TFA.

4.3.3 Algoritmo

O algoritmo do sistema transacional atualmente implementado é baseado no algoritmo de um dos sistemas transacionais da CMTJava (BANDEIRA, 2013) e no TFA (SAAD; RAVINDRAN, 2012), sendo que o primeiro foi base para sincronização entre os objetos presentes em um mesmo nodo e o segundo para sincronização entre os nodos. Em transações envolvendo objetos distribuídos também deve ser feita a sincronização local, pois mais de uma transação de um mesmo nodo pode estar acessando o mesmo objeto remoto. A seguir serão apresentados detalhes sobre o sistema transacional implementado.

4.3.3.1 Criação de uma Transação

A Figura 23 mostra como o sistema transacional inicializa uma transação. Como pode ser visto, ele inicializa a variável `validationStamp` com o valor atual do do relógio deste nodo (`VersionClock`). Logo após ele cria os conjuntos de leitura e de escrita e, por fim, inicializa o ID desta transação.

```
1 validationStamp = VersionClock.getReadStamp();
2 writeSet = new WriteSet();
3 readSet = new ReadSet();
4 transId = geraTransId();
```

Figura 23: Inicialização de uma transação

4.3.3.2 Abrir um RemoteTObject

Para abrir um RemoteTObject, deve ser utilizado o método `openObject`. Este método, além de baixar uma réplica do objeto, também realiza a sincronização dos relógios dos nodos envolvidos na comunicação. O seu algoritmo pode ser visto na Figura 24.

O diretório do nodo dono do objeto, ao receber um mensagem solicitando um objeto, sincroniza os relógios e envia uma cópia do objeto para o nodo solicitante, conforme operações contidas na Figura 25.

Após aberto, o objeto pode ser lido e escrito pelas transações do nodo da mesma forma que um objeto local.

4.3.3.3 Leitura de um Atributo

Para realizar a leitura do valor de algum atributo de um TObject, deve-se utilizar o método `get` deste atributo. Este método é gerado automaticamente pelo compilador para cada atributo de um TObject, por exemplo, para o atributo `Double balance` da classe `TAccount`, é gerado o método `STM<Double> getBalance()`. Este método

```
1  RemoteTObject openObject(String id){
2      if (dono atual do objeto == "localhost"){
3          return objeto;
4      }
5      if (diretório já possui uma cópia atual do objeto){
6          return objeto;
7      }
8      Envia mensagem solicitando copia do objeto para seu dono atual.
          Junto a ela, envia valor atual do relógio;
9      Recebe a resposta.
10     if (resposta é um RemoteTObjeto){
11         Junto a ele virá o valor do relógio do remetente. Realiza a
              sincronização dos relógios;
12         return objeto;
13     }
14     if (resposta é informação de novo dono){
15         return openObject(id);
16     }
17 }
```

Figura 24: Algoritmo do método openObject

```
1      Recebe pedido e valor do relógio do remetente;
2      Sincroniza relógios;
3      if (dono atual do objeto == "localhost"){
4          Envia uma mensagem com uma copia do objeto para o remetente.
              Junto a esta mensagem é enviado valor atual do relógio;
5      }
6      else{
7          Envia mensagem informando novo dono;
8      }
```

Figura 25: Algoritmo para responder a uma solicitação de objeto

acessa os metadados das transações, seguindo a estratégia utilizada pelo sistema transacional.

O pseudocódigo utilizado nos métodos `get` pode ser visto na Figura 26, simplificado para mais fácil entendimento. Primeiramente, é testado se a própria transação que está acessando o método já possui o *lock* de escrita deste atributo. Se sim, ela já escreveu neste atributo, então deve retornar o valor diretamente de seu conjunto de escrita (linhas 6..8). Se não possuir o *lock*, realiza a leitura do valor até que sua versão seja igual à versão da leitura anterior e o *lock* esteja liberado. Isto garante que nenhuma outra transação alterou o valor durante a leitura (linhas 11 e 12).

Após estes passos, adiciona o atributo ao conjunto de leitura (linha 14). Caso a versão do atributo seja maior que o valor do *timestamp* de validação da transação, a transação tenta estender o seu *timestamp* para o valor do relógio do nodo, para tal

```
1  public STM<Double> getBalance() {
2      return new STM<Double> ({Trans t =>
3          TResult r = null;
4          Double result = null;
5
6          if(balanceFieldInfo.wlock bloqueado pela própria transação){
7              result = lê valor do writeset;
8              r = new TResult(result, t, Trans.Status.ACTIVE);
9          }
10         else{
11             até balanceFieldInfo.rlock liberado && version == version
                   lida na iteração anterior
12             result=balance;
13
14             t.readSet.put(balanceFieldInfo, version);
15
16             if( version > t.validationStamp && !t.extend() ){
17                 Directory.renewRemoteTObject(id);
18                     r = new TResult(null, t, Trans.Status.ABORTED);
19             }
20             else{
21                 r = new TResult(result, t, Trans.Status.ACTIVE);
22             }
23         }
24         r
25     });
26 }
```

Figura 26: Pseudocódigo de um método get

revalida todo o seu conjunto de leitura local e remotamente, se não conseguir revalidar, aborta (linhas 16..18), caso contrário retorna o valor lido (linha 21).

4.3.3.4 Escrita em um Atributo

De maneira análoga ao que acontece com o método `get` para leitura, para realizar uma escrita em um atributo de um objeto transacional deve ser utilizado o método `set` deste atributo. Ele também acessa os metadados das transações, conforme a estratégia utilizada pelo sistema transacional.

O pseudocódigo simplificado deste método pode ser visto na Figura 27. Assim como ocorre no método `get`, ele testa se a transação já é dona do *lock* do atributo. Se for, ela simplesmente sobrescreve o valor do atributo no conjunto de escrita. Caso contrário, a transação tenta adquirir o *lock* de escrita e se falhar, aborta. Se conseguir adquirir o *lock*, adiciona este atributo ao conjunto de escrita com o valor a ser escrito.

Por último, caso a versão do atributo seja maior que o valor do *timestamp* de validação da transação, ela tenta estender o seu *timestamp* da mesma forma que

ocorre no método `get`.

```
1  public STM<stm.Void> setBalance (Double n) {
2      return new STM<stm.Void>({Trans t =>
3          TResult r = null;
4          if(balanceFieldInfo.wlock bloqueado pela própria transação){
5              t.writeSet.put(balanceFieldInfo, n);
6              r = new TResult(new stm.Void(), t, Trans.Status.ACTIVE);
7          }
8          else{
9              if(balanceFieldInfo.wlock.compareAndSet(null, t.transId,
                   false, true)){
10                 t.writeSet.put(balanceFieldInfo, n);
11                 if(balanceFieldInfo.rlock.getReference() > t.
                       validationStamp && !t.extend()){
12                     Directory.renewRemoteTObject(id);
13                     r = new TResult(null, t, Trans.Status.ABORTED);
14                 }
15                 else{
16                     r = new TResult(new stm.Void(), t, Trans.Status.
                           ACTIVE);
17                 }
18             }
19             else{
20                 r = new TResult(null, t, Trans.Status.ABORTED);
21             }
22         }
23         r
24     });
25 }
```

Figura 27: Pseudocódigo de um método set

4.3.3.5 Commit

O método `commit` é responsável por realizar a transposição dos valores do conjunto de escrita para a memória, no caso dos objetos locais, e por informar o diretório de que o nodo atual é o novo dono, no caso dos objetos remotos. Porém, antes disso, ele deve verificar se houve algum conflito, pois se ocorreu, a transação deve abortar e ser reexecutada.

O algoritmo que este sistema transacional utiliza para o método `commit` pode ser visto na Figura 28, na forma de pseudo código. Primeiramente, ele tenta adquirir o *lock* de leitura para todos os objetos no conjunto de escrita (linhas 2..4). Isto garante que nenhuma outra transação irá realizar leitura em algum dos atributos enquanto ele é escrito. Se falhar ao adquirir algum dos *locks*, todos os *locks* são liberados e o *commit* retorna *false*, abortando a transação (linhas 5 e 6).

```
1  public <A> boolean commit(){
2      for(FieldInfo field : writeSet.log.keySet()){
3          Long ref = (Long)field.rlock.getReference();
4          if(!field.rlock.compareAndSet(ref,transId,false, true)){
5              releaseLocks();
6              return false;
7          }
8      }
9      Long writeVersion = VersionClock.getWriteStamp();
10     if(writeVersion > validationStamp+1){
11         if(!validate()) {
12             releaseLocks();
13             return false;
14         }
15     }
16     for(String id : objetosRemotosNoWS){
17         if(!Directory.lockRemoteTObject(id)) {
18             libera locks remotos adquiridos;
19             releaseLocks();
20             return false;
21         }
22     }
23     for(String id : objetosRemotosNoRS){
24         solicita ao dono do objeto a versão atual dos atributos;
25         testa se a versão de cada atributo > validationStamp {
26             releaseRemoteLocks(locksRemotosAdquiridos);
27             releaseLocks();
28             Directory.renewRemoteTObject(id);
29             return false;
30         }
31     }
32     if(writeSet.log.size()!=0){
33         for(Map.Entry entry : writeSet){
34             FieldInfo<A> key = (FieldInfo<A>) entry.getKey();
35             A newvalue = (A)entry.getValue();
36             key.updateField.invoke(newvalue);
37             key.rlock.compareAndSet(transId, writeVersion, true,
                   false);
38             key.wlock.compareAndSet(transId, null, true, false);
39         }
40         for(String id : objetosRemotosNoWS){
41             Directory.changeOwner(id);
42         }
43     }
44     releaseRemoteLocks(locksRemotosAdquiridos);
45     return true;
46 }
```

Figura 28: Algoritmo do método `commit`

Com o método `VersionClock.getWriteStamp`, incrementa e lê o valor do relógio local, sendo esta a versão de escrita que será utilizada (linha 9). Então, testa se alguma outra transação realizou *commit* desde o começo da transação (linha 10), caso positivo, tenta validar o conjunto de leitura da transação, se não conseguir, aborta (linhas 11..13). Neste ponto da execução, uma transação que somente possua atributos de objetos locais em seu conjunto de leitura já estaria pronta para escrever os valores na memória. Porém transações com objetos remotos ainda não estão pontas, tendo que realizar as operações para verificar conflitos distribuídos.

O primeiro passo para a verificação de conflitos distribuídos é adquirir todos os *locks* dos objetos remotos presentes no conjunto de escrita, caso não consiga, aborta (16..20). Isto garante que transações de outros nodos, envolvendo estes objetos não realizem *commit* enquanto esta não acabar. Logo após, para cada objeto remoto no conjunto de leitura, solicita ao seu dono a versão atual de cada atributo e testa se esta é maior do que a versão de validação da transação, se sim, significa que o objeto já foi modificado desde o começo da transação, então aborta e atualiza a cópia local do objeto(23..29). Se passar por esta validação, a transação está pronta para realizar o *commit*.

Se esta não for uma transação de somente leitura (linha 32), percorre todo o conjunto de escrita (linhas 33..36) invocando a *closure* armazenada no atributo `updateField` do `fieldInfo` (linha 36). Esta closure atualiza o atributo do objeto com o valor presente no conjunto de escrita. Libera os *locks* de leitura e de escrita locais (linhas 37 e 38). Após estas operações, os objetos já estão atualizados neste nodo. O próximo passo é informar o diretório que este nodo é o novo dono dos objetos remotos presentes no conjunto de escrita (40 e 41). Para finalizar, todos os *locks* de todos os objetos remotos são liberados (linha 44).

5 VALIDAÇÃO DA IMPLEMENTAÇÃO

Foram realizados alguns testes para validar o funcionamento do sistema transacional distribuído implementado, avaliando a corretude dos resultados. A seguir, serão discutidos detalhes dos testes, começando pelo ambiente nos quais eles foram executados e concluindo com os resultados destes experimentos.

5.1 Ambiente de Validação

Para a execução dos testes, foram utilizados 2 servidores Dell PowerEdge T430, com a seguinte descrição cada:

- Processador Intel(R) Xeon(R) CPU E5-2420, 1.90GHz, 6 cores reais que executam 12 threads com Hyper-Threading;
- 8GB DIMM DDR3 Synchronous 1600 MHz;
- Interface de Rede Ethernet NetXtreme BCM5720 Gigabit Ethernet PCIe;
- 1 Hard Disk com capacidade de 1TB - Seagate ST1000NM0033-9ZM.
- Sistema operacional GNU/Linux, através da distribuição Ubuntu 14.04 LTS - Trusty x86_64 - Kernel Version 3.13.0-30-generic.

Além disso, as máquinas foram interligadas através de um *switch* TP-LINK TL-SG1008D Gigabit.

5.2 Experimento Realizado

Para validar a corretude do sistema foi utilizando uma implementação simples de conta bancária, onde, cada conta possui um atributo que representa seu saldo (`balance`) e outro que representa o número de operações realizadas (`ops`). Esta conta possui os métodos de saque e de depósito, implementados na forma de transações. Estes métodos consistem em subtrair e somar, respectivamente, um valor X `balance` e incrementar `ops` em 1.

O teste consiste em fazer um determinado número de operações, utilizando um certo número de *threads* por nodo. Desta forma, existe concorrência entre as *threads* de um mesmo nodo e também entre os nodos. Saques e depósitos são realizados sempre com o mesmo valor, sendo assim, após a execução, o saldo deve ser o mesmo saldo inicial. Ao final da execução de cada teste, tanto o saldo quanto o número de operações foi testado, para verificar a corretude da execução.

Um dos nodos cria uma conta e a registra no diretório. Esta será compartilhada entre todos os nodos, sendo acessada por transações de cada um deles. Além disso, cada nodo também cria uma conta acessada exclusivamente por transações locais.

Em cada nodo são passados como argumentos da execução: (1) número de *threads*, sendo que em cada nodo são criadas N *threads* para transações locais e N para operações distribuídas, totalizando 2N; (2) número de operações na conta compartilhada a serem executadas por *thread*; e (3) número de operações na conta local a serem executadas por *thread*.

Por exemplo, considere que foram utilizados dois nodos e foram passados os argumentos: 6, 10000, 20000. O teste será executado com 6 *threads* executando transações locais e 6 *threads* executando operações distribuídas, em cada nodo; 60000 transações distribuídas serão realizadas por cada nodo, utilizando a conta compartilhada; e 120000 operações serão realizadas por cada nodo em sua conta local.

Foram executados testes com diferentes combinações de parâmetros de entrada. Em todos eles, o número de transações se manteve fixo em 6. Tanto para operações na conta compartilhada quanto para operações na conta local, foram utilizados valores entre 10000 e 80000, variando de 10000 em 10000. Foram utilizadas todas as combinações desses valores.

O teste foi executado 30 vezes para cada combinação de entradas. Para cada combinação foi calculada a média dos tempos de execução coletados, bem como o coeficiente de variação, que representa o desvio padrão dividido pela média. A análise dos resultados se encontra na seção a seguir.

5.3 Resultados Obtidos

A Tabela 5 contém a média do tempo de execução, em segundos, de cada uma das combinações de parâmetros de execução. Nas colunas tem-se o tempo o número de operações realizadas com objetos locais por *thread* e, nas linhas, o número de operações com o objeto compartilhado por *thread*. Os coeficientes de variação destes dados podem ser vistos na Tabela 6.

Para melhor visualização dos resultados, foi gerado o gráfico da Figura 29. Em seu eixo X, ele traz o número de transações por *thread*, envolvendo a conta compartilhada, realizadas por cada nodo. No eixo Y o tempo de execução em segundos e, contém

Tabela 5: Tempo de Execução em Segundos dos Testes com Transações Envolvendo Objetos Locais e Objeto Remoto (nas colunas o número de transações com objetos locais e nas linhas com objetos remotos)

Op./Thread	10000	20000	30000	40000	50000	60000	70000	80000
10000	7.1071	7.4288	7.7331	8.0415	8.2802	8.5572	8.8488	9.0994
20000	7.4066	7.5248	7.8600	8.1397	8.3923	8.7225	9.0106	9.2556
30000	7.7531	7.9158	7.9783	8.2786	8.5907	8.8475	9.1713	9.4784
40000	8.0521	8.1665	8.2648	8.4064	8.7530	8.9966	9.3291	9.5527
50000	8.2777	8.4297	8.6107	8.7415	8.9269	9.1057	9.4134	9.7218
60000	8.5349	8.7099	8.8480	9.0431	9.1418	9.2244	9.5661	9.9165
70000	8.8501	9.0005	9.1706	9.3479	9.4707	9.5957	9.7243	10.0522
80000	9.1068	9.3053	9.4866	9.6263	9.7708	9.9055	9.9791	10.1478

Tabela 6: Coeficiente de Variação das Execuções com Transações Envolvendo Objetos Locais e Objeto Remoto (nas colunas o número de transações com objetos locais e nas linhas com objetos remotos)

Op./Thread	10000	20000	30000	40000	50000	60000	70000	80000
10000	0.0132	0.0105	0.0103	0.0159	0.0136	0.0214	0.0180	0.0173
20000	0.0113	0.0116	0.0116	0.0155	0.0196	0.0166	0.0163	0.0189
30000	0.0133	0.0161	0.0153	0.0141	0.0161	0.0131	0.0163	0.0158
40000	0.0181	0.0142	0.0106	0.0175	0.0176	0.0171	0.0169	0.0169
50000	0.0145	0.0166	0.0139	0.0196	0.0176	0.0187	0.0193	0.0196
60000	0.0139	0.0141	0.0137	0.0180	0.0198	0.0202	0.0215	0.0196
70000	0.0146	0.0229	0.0153	0.0211	0.0203	0.0257	0.0207	0.0224
80000	0.0221	0.0160	0.0200	0.0180	0.0190	0.0245	0.0264	0.0299

uma linha para cada número de operações realizadas com a conta local de cada nodo. Também foi gerado o gráfico da Figura 31, para visualizar-se o coeficiente de variação.

Pode-se concluir, ao analisar o gráfico da Figura 29, que o tempo de execução cresce de forma linear conforme aumenta-se o número de operações, porém o aumento ocorre de forma muito inferior ao aumento do número de transações, já que, por exemplo, com 10000 transações na conta local e 10000 na conta compartilhada tem-se em torno de 7,1s. Já com 10000 na local e 80000 na compartilhada, tem-se 9,1, ou seja, o número de operações subiu 8 vezes e o tempo apenas em torno de 1,3 vezes.

O gráfico da Figura 30 contém os mesmos dados do gráfico anterior, porém este representa as operações locais no eixo X e as operações com objeto compartilhado nas linhas. Ele foi plotado com o objetivo de verificar a diferença de tempo de execução de operações locais e remotas, analisando a diferença entre X operações com objeto local e Y com objeto compartilhado, e Y com local e X com compartilhado, ou seja, é o gráfico gerado a partir da Tabela 5 transposta.

Comparando ambos os gráficos, pode-se ver que eles são muito semelhantes, o

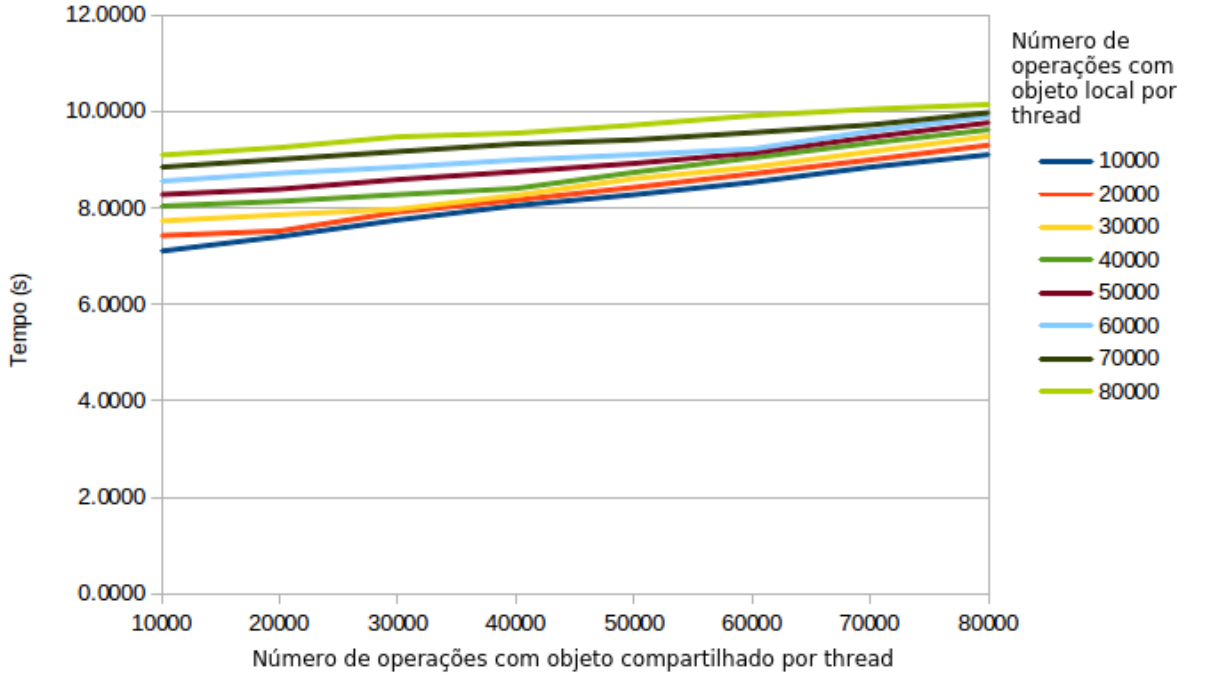

Figura 29: Tempo de Execução dos Testes com Transações Envolvendo Objetos Locais e Objeto Remotos

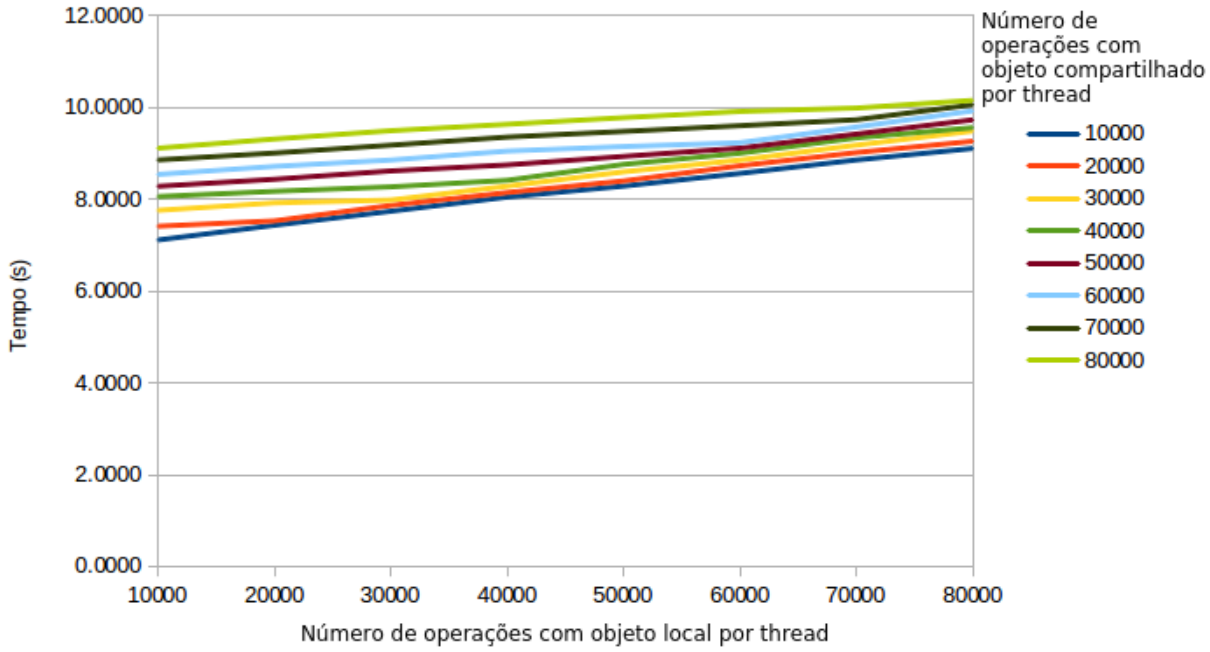

Figura 30: Tempo de Execução dos Testes com Transações Envolvendo Objetos Locais e Objeto Remoto (Transposto)

que sugere que transações envolvendo somente objetos locais e envolvendo somente objetos remotos estão com tempo muito similar. Para uma melhor análise, foram comparados os tempos caso a caso, algumas destas comparações são vistas nos gráficos das Figuras 32, 33 e 34. Eles trazem a comparação de execuções com o mesmo

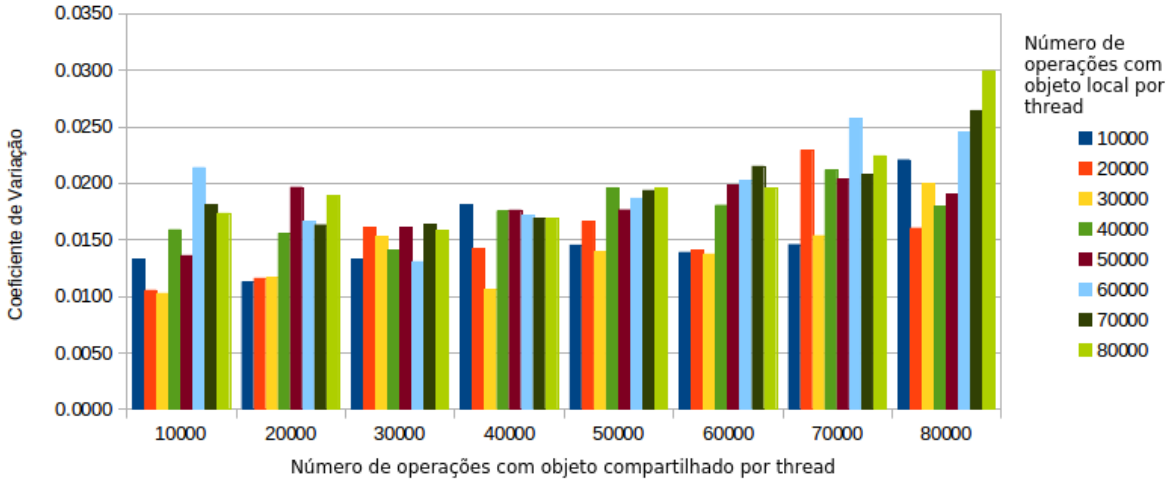

Figura 31: Coeficiente de Variação das Execuções com Transações Envolvendo Objetos Locais e Objeto Remoto (nas colunas o número de transações com objetos locais e nas linhas com objetos remotos)

número de operações, porém as barras azuis tem no eixo x o número de operações locais e as barras laranjas tem no x as operações com o objeto compartilhado. Ambas possuem o outro número de operações na legenda do gráfico.

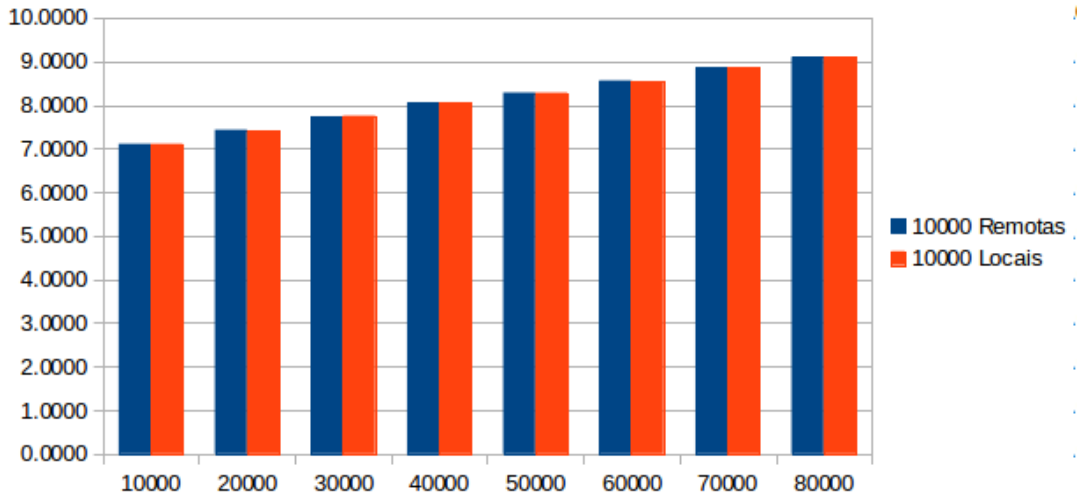

Figura 32: 10000 Transações com Objetos Locais X 10000 com Objeto Remoto

No gráfico da Figura 32, os tempos de operações com objetos locais e com objetos remotos foram extremamente similares. Já os gráficos das Figura 33 e 34 foram os que apresentaram maior diferença entre os valores, porém esta diferença continua sendo pequena, não sendo superior ao desvio padrão, o que leva a crer que transações envolvendo objetos locais e transações envolvendo objetos remotos realmente estão com tempo muito similar, porém é provável que esteja ocorrendo serialização das transações devido à alta contenção, fazendo com que o paralelismo extra adicionado

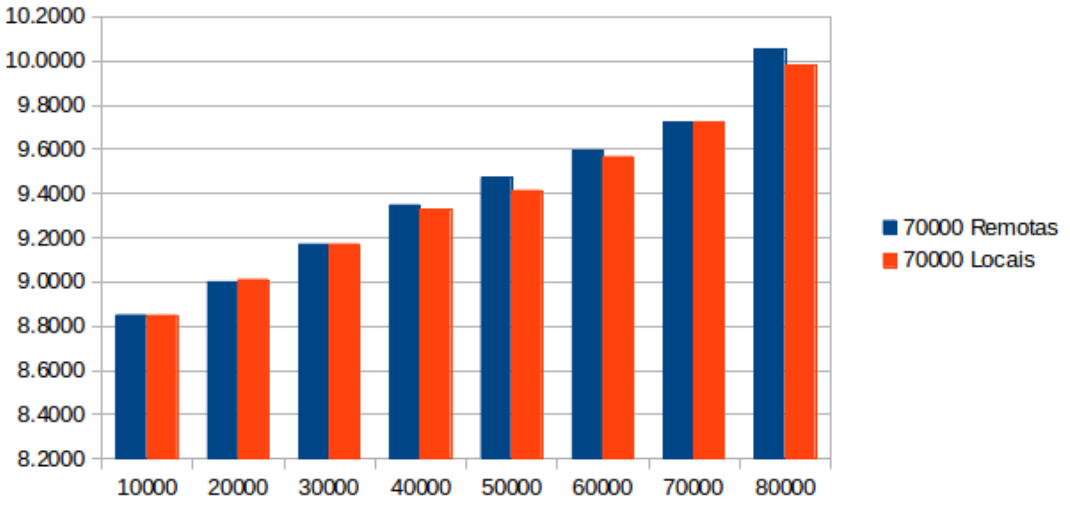

Figura 33: 70000 Transações com Objetos Locais X 70000 com Objeto Remoto

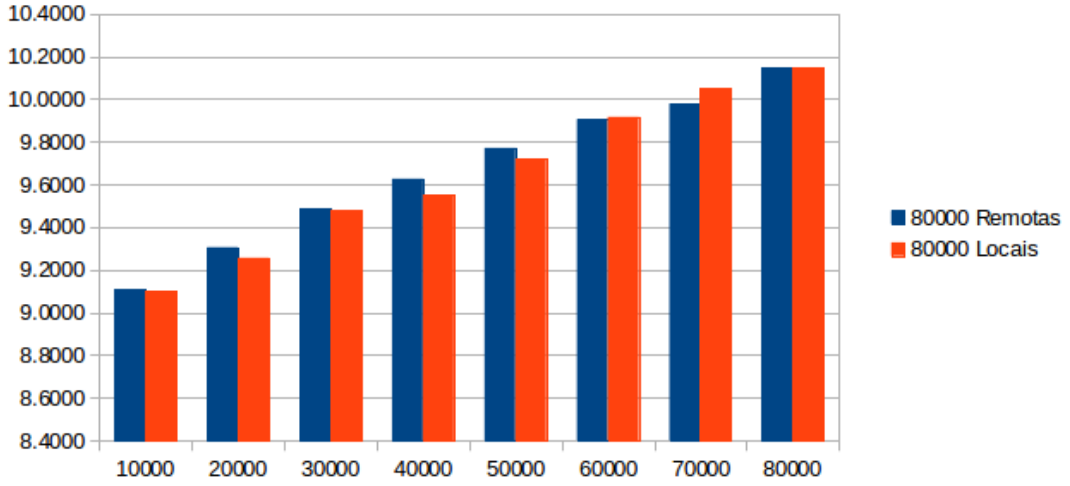

Figura 34: 80000 Transações com Objetos Locais X 80000 com Objeto Remoto

pela distribuição não seja totalmente aproveitado.

Uma possível causa para isto pode estar nas características da aplicação implementada para os testes. Ela representa um caso extremo de conflitos, tanto por possuir apenas um objeto compartilhado quanto por suas transações serem muito pequenas. Ao se combinar estas duas características, cria-se um cenário onde todas as *threads*, de todos os nodos, devem adquirir os *locks* de um mesmo objeto e, como as transações são pequenas, o tempo em que a transação fica com *locks* adquiridos se torna relativamente elevado. Sendo assim, a contenção na aplicação que foi utilizada para os testes é extremamente elevada, influenciando o tempo de execução.

Para verificar o *overhead* adicionado por esta implementação, seus tempos de execução foram comparados com os tempos de execução da CMTJava (BANDEIRA, 2013). Foram realizados testes com a CMTJava fazendo as mesmas operações dos

testes realizados com a DCMTJava, porém com dois objetos locais, ao invés de um local e um compartilhado. As comparações de tempo foram realizadas entre as execuções com o mesmo número total de operações, sendo que, na DCMTJava, foi utilizada a configuração de 50% das operações locais e 50% utilizando o objeto compartilhado. A Tabela 7 traz a média dos tempos de execução.

Tabela 7: Tempo das Execuções em Segundos, com CMTJava e DCMTJava com o Mesmo Número Total de Operações

Total de Op.	**40000**	**80000**	**120000**	**160000**	**200000**	**240000**	**280000**	**320000**
CMTJava	1.4301	2.1245	2.8540	3.4905	4.2670	5.0242	5.6408	6.3089
DCMTJava	7.1071	7.5248	7.9783	8.4064	8.9269	9.2244	9.7243	10.1478

Para melhor visualização dos resultados, foi plotado o gráfico da Figura 35. Como esperado, o tempo de execução da DCMTJava foi maior, nos casos avaliados, pois esta apresenta o tempo adicional gasto nas comunicações pela rede para evitar conflitos distribuídos. Este tempo se torna ainda mais significante devido às transações desta aplicação de teste serem pequenas. Já a CMTJava executa em um único nodo, não sendo afetada por este *overhead* .

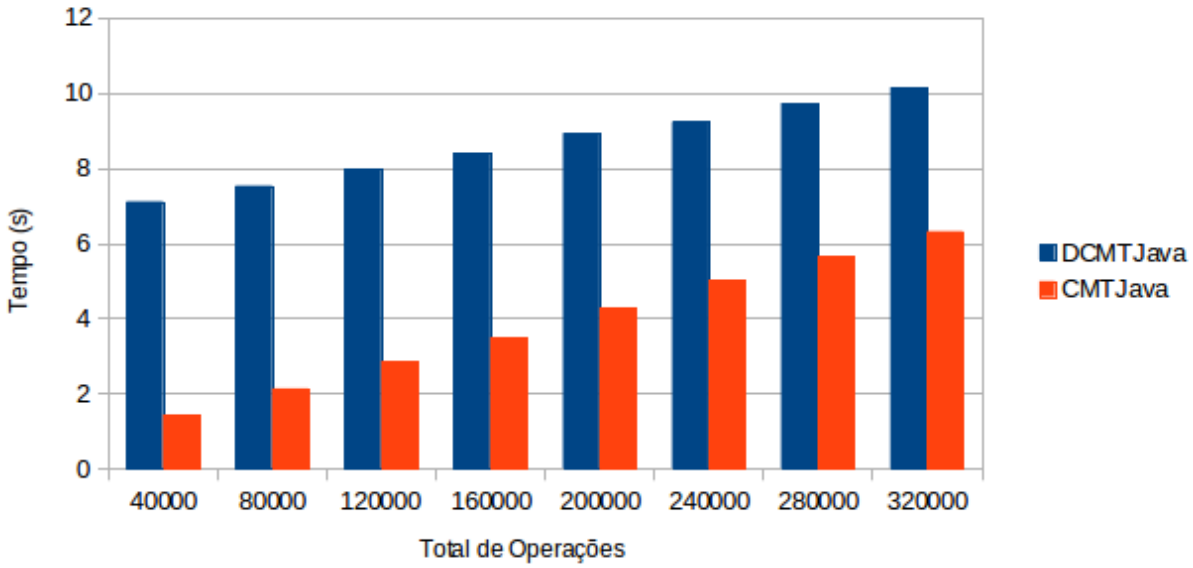

Figura 35: Tempo das Execuções com CMTJava e DCMTJava com o Mesmo Número Total de Operações

Analisando o gráfico, percebe-se que a diferença de tempo entre as execuções de ambas as linguagens diminui conforme é aumentado o número de operações, como era esperado, já que a DCMTJava distribuiu o trabalho entre dois nodos e a CMTJava não, saturando sua capacidade de processamento antes da DCMTJava. Isto leva a

crer que se forem realizados testes com ainda mais operações, em algum momento o tempo da DCMTJava será melhor que o da CMTJava, porém são necessários mais testes para que isto possa ser afirmado. O fato da curva de crescimento do tempo de execução da DCMTJava ser menos acentuada do que a curva da CMTJava, possivelmente se deve ao fato da primeira ter acesso a um maior número de núcleos de processamento, já que executa em duas máquinas.

Com estes testes foi possível validar o funcionamento do sistema transacional proposto por este trabalho, mesmo com uma aplicação que representa um caso desfavorável para execução distribuída. Acredita-se que em testes futuros, que apresentem características mais próximas às de aplicações reais, os resultados de tempo serão muito melhores, pois essas aplicações geralmente possuem menor contenção do que a que foi utilizada nos testes.

6 CONSIDERAÇÕES FINAIS

Hoje em dia há uma tendência de cada vez mais dispositivos computacionais se comunicarem entre si. Além disso, grande parcela da computação de alto desempenho também é composta por sistemas distribuídos. Estes fatos tornam o estudo de arquiteturas distribuídas cada vez mais importante. Estas arquiteturas são relativamente difíceis de programar, pois apresentam, além dos problemas adicionais ao se tratar a concorrência, outras dificuldades adicionais, como sincronização e comunicação. Tais dificuldades fazem com que o desenvolvimento de estratégias de programação mais simples seja importante.

O modelo de memórias transacionais (TM) tem se mostrado bastante promissor para facilitar a programação de arquiteturas paralelas, porém, devido às características diferenciadas das arquiteturas distribuídas, os modelos de memória transacional já propostos não podem ser utilizados diretamente. Sendo assim, o principal objetivo deste trabalho foi estender uma linguagem de programação para que esta suportasse transações envolvendo objetos distribuídos.

Para tal, foram estudados os fundamentos necessários para o uso do modelo de TM em arquiteturas distribuídas, bem como os principais conceitos de TM, focando nas estratégias de implementação. Também foram analisadas as características de três trabalhos que compõe o estado da arte de STM para arquiteturas distribuídas.

Este trabalho propôs a linguagem DCMTJava que estende a linguagem CMTJava, permitindo que transações envolvam tanto objetos locais como distribuídos. Para a implementação da linguagem, foi proposta a combinação de dois algoritmos para transações. Objetos locais são validados usando o algoritmo da swissTM (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009), algoritmo comprovadamente mais rápido que outros algoritmos clássicos de transações, como TL2 (DICE; SHALEV; SHAVIT, 2006) e RSTM (MARATHE et al., 2006). Para objetos remotos, a validação é feita de forma parecida com o algoritmo TFA (SAAD; RAVINDRAN, 2012), permitindo que objetos sejam validados somente no final da transação, evitando comunicação excessiva.

O funcionamento do sistema transacional da linguagem foi validado, através dos testes apresentados no Capítulo 5 e foi comprovado que ele funciona da maneira

esperada.

6.1 Trabalhos Futuros

São diversas as possibilidades de trabalhos a serem realizados na continuidade deste. Alguns deles são citados a seguir:

- Realização de testes de escalabilidade, para analisar como a implementação se comportará quando se escala o número de nodos do sistema e o número de objetos;
- Implementação de aplicações mais complexas para realização de testes;
- Conclusão da implementação do compilador da linguagem, visto que, atualmente, algumas estruturas são geradas manualmente;
- Otimização da implementação para ganhar desempenho, tanto nas transações locais quanto nas distribuídas;
- Utilização de outros modelos para as transações distribuídas, como algum que acesse os objetos em seu nodo de origem, ao invés de movê-los, pois alguns objetos não podem ser movidos. Além disso, isto permitirá que sejam realizadas comparações entre as implementações para descobrir-se qual estratégia funciona melhor em cada cenário.

REFERÊNCIAS

BADUEL, L.; BAUDE, F.; CAROMEL, D.; CONTES, A.; HUET, F.; MOREL, M.; QUILICI, R. **Grid Computing: Software Environments and Tools**. [S.l.]: Springer-Verlag, 2006.

BANDEIRA, R. d. L. **Programação Multicore com Memórias Transacionais**. 2012. 40p. Trabalho Individual de Mestrado — Programa de Pós-Graduação em Computação, UFPel, Pelotas, RS.

BANDEIRA, R. d. L. **Um Sistema de Detecção de Conflitos com Invalidação Mista para a Linguagem CMTJava**. 2013. 74p. Dissertação de Mestrado — Programa de Pós-Graduação em Computação, UFPel, Pelotas, RS.

BOCCHINO, R. L.; ADVE, V. S.; CHAMBERLAIN, B. L. Software transactional memory for large scale clusters. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 13., 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.247–258. (PPoPP '08).

CAO MINH, C.; CHUNG, J.; KOZYRAKIS, C.; OLUKOTUN, K. STAMP: Stanford Transactional Applications for Multi-Processing. In: IISWC '08: PROCEEDINGS OF THE IEEE INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION, 2008. **Anais...** [S.l.: s.n.], 2008.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Sistemas Distribuídos - 4ed**: Conceitos e Projeto. [S.l.]: BOOKMAN COMPANHIA ED, 2007.

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional locking II. In: DISTRIBUTED COMPUTING, 20., 2006, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2006. p.194–208. (DISC'06).

DRAGOJEVIć, A.; GUERRAOUI, R.; KAPALKA, M. Stretching transactional memory. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2009., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.155–165. (PLDI '09).

DU BOIS, A. R. Memórias Transacionais e Troca de Mensagens: Duas Alternativas para a Programação de Máquinas Multi-Core. **SBC, P. A. (Ed.). Escola Regional de Alto Desempenho**, [S.l.], p.43–76, 2008.

DU BOIS, A. R.; ECHEVARRIA, M. A. Domain Specific Language for Composable Memory Transactions in Java. In: DSL '09: PROCEEDINGS OF THE IFIP TC 2 WORKING CONFERENCE ON DOMAIN-SPECIFIC LANGUAGES, 2009, Berlin, Heidelberg. **Anais...** Springer-Verlag, 2009. p.170–186.

ECHEVARRIA, M. **Uma Linguagem de Domínio Específico para Programação de Memórias Transacionais em Java**. 2010. 80p. Dissertação de Mestrado — Programa de Pós-Graduação em Informática, UCPel, Pelotas, RS.

FELBER, P.; FETZER, C.; RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 13., 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.237–246. (PPoPP '08).

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI (2nd ed.)**: portable parallel programming with the message-passing interface. Cambridge, MA, USA: MIT Press, 1999.

HARRIS, T.; CRISTAL, A.; UNSAL, O.; AYGUADE, E.; GAGLIARDI, F.; SMITH, B.; VALERO, M. Transactional Memory: An Overview. **Micro, IEEE**, [S.l.], v.27, n.3, p.8–29, may-june 2007.

HARRIS, T.; LARUS, J.; RAJWAR, R. **Transactional Memory, 2nd Edition**. 2nd.ed. [S.l.]: Morgan and Claypool Publishers, 2010.

HARRIS, T.; MARLOW, S.; PEYTON-JONES, S.; HERLIHY, M. Composable memory transactions. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2005, New York, NY, USA. **Proceedings...** ACM, 2005. p.48–60. (PPoPP '05).

HERLIHY, M.; CALCIU, I. Work in Progress: Shared Nothing Transactional Memory. In: SYSTEMS FOR MULTI-CORE ARCHITECTURES (SFMA/EUROSYS), 2011. **Anais...** [S.l.: s.n.], 2011.

HERLIHY, M.; LUCHANGCO, V.; MOIR, M. A flexible framework for implementing software transactional memory. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, 21., 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p.253–262. (OOPSLA '06).

HERLIHY, M.; MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In: OF THE 20TH ANNUAL INTERNATIONAL SYMPOSIUM ON

COMPUTER ARCHITECTURE, 1993, New York, NY, USA. **Proceedings...** ACM, 1993. p.289–300. (ISCA '93).

HERLIHY, M.; SUM, Y. Distributed Transactional Memory for Metric-space Network. In: DISC'05, 2005. **Anais...** [S.l.: s.n.], 2005.

KOTSELIDIS, C.; ANSARI, M.; JARVIS, K.; LUJáN, M.; KIRKHAM, C.; WATSON, I. DiSTM: A Software Transactional Memory Framework for Clusters. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 2008., 2008, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2008. p.51–58. (ICPP '08).

LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. **Commun. ACM**, [S.l.], v.21, n.7, p.558–565, 1978.

LU, K.; WANG, R.; LU, X. Brief announcement: NUMA-aware transactional memory. In: ACM SIGACT-SIGOPS, 29., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.69–70. (PODC '10).

MANASSIEV, K.; MIHAILESCU, M.; AMZA, C. Exploiting distributed version concurrency in a transactional memory cluster. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p.198–208. (PPoPP '06).

MARATHE, V. J.; SPEAR, M. F.; HERIOT, C.; ACHARYA, A.; EISENSTAT, D.; III, W. N. S.; SCOTT, M. L. Lowering the overhead of nonblocking software transactional memory. In: DEPT. OF COMPUTER SCIENCE, UNIV. OF ROCHESTER, 2006. **Anais...** [S.l.: s.n.], 2006.

MOGGI, E. Notions of Computation and Monads. **Information and Computation**, [S.l.], v.93, p.55–92, 1989.

MOORE, K. E.; BOBBA, J.; MORAVAN, M. J.; HILL, M. D.; WOOD, D. A. LogTM: Log-based Transactional Memory. In: **Proceedings of the 12th International Symposium on High-Performance Computer Architecture**. [S.l.: s.n.], 2006. p.254–265.

NEWBERN, J. **All About Monads**. Disponível em: http://www.haskell.org/haskellwiki/All_About_Monads.

PARR, T. **The Definitive ANTLR Reference**: Building Domain-Specific Languages. [S.l.]: Pragmatic Bookshelf, 2007.

PAYER, M.; GROSS, T. Performance evaluation of adaptivity in software transactional memory. In: PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE (ISPASS), 2011 IEEE INTERNATIONAL SYMPOSIUM ON, 2011. **Anais...** [S.l.: s.n.], 2011. p.165 –174.

RICO, T.; PILLA, M.; DU BOIS, A. Energy Consumption on Software Transactional Memories. In: COMPUTER SYSTEMS (WSCAD-SSC), 2012 13TH SYMPOSIUM ON, 2012. **Anais...** [S.l.: s.n.], 2012. p.194 –201.

RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. Memórias Transacionais: Uma Nova Alternativa para Programação Concorrente. In: MINICURSOS DO VIII WSCAD, 2007. **Anais...** [S.l.: s.n.], 2007.

SAAD, M.; RAVINDRAN, B. **Supporting STM in Distributed Systems**: Mechanisms and a Java Framework. [S.l.: s.n.], 2011.

SAAD, M.; RAVINDRAN, B. Transactional Forwarding: Supporting Highly-Concurrent STM in Asynchronous Distributed Systems. In: COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD), 2012 IEEE 24TH INTERNATIONAL SYMPOSIUM ON, 2012. **Anais...** [S.l.: s.n.], 2012. p.219 –226.

WADLER, P. Monads for Functional Programming. In: ADVANCED FUNCTIONAL PROGRAMMING, FIRST INTERNATIONAL SPRING SCHOOL ON ADVANCED FUNCTIONAL PROGRAMMING TECHNIQUES-TUTORIAL TEXT, 1995, London, UK, UK. **Anais...** Springer-Verlag, 1995. p.24–52.

ZHENG, C. Distributed Obstruction-Free Transactional Memory. In: 2002, Providence, Rhode Island, USA. **Anais...** Brown University, 2002.