Tiago Marques do Vale

Licenciado em Engenharia Informática

A Modular Distributed Transactional Memory Framework

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : João Manuel dos Santos Lourenço,
Prof. Auxiliar,
Universidade Nova de Lisboa

Júri:

Presidente:

Arguente: Luís Manuel Antunes Veiga

Vogal: João Manuel dos Santos Lourenço

Setembro, 2012

A Modular Distributed Transactional Memory Framework

Copyright © Tiago Marques do Vale, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Aos meus pais, pela oportunidade proporcionada.

Acknowledgements

I am thankful to my advisor, João Lourenço, not only for this opportunity and for introducing me to the world of research, but also for his guidance, advices and devotion during the elaboration of the dissertation. I would also like to extend my sincerest gratitude to Ricardo Dias for his effort, help and brainstorming, to whom I am greatly obliged. I deeply thank you both for supporting and pushing me forward in moments (and nights...!) of uncertainty and despair.

I am grateful to FCT (Fundação para a Ciência e Tecnologia, Ministério da Edução e Ciência) for kindly granting me with a scholarship in the scope of the Synergy-VM project (PTDC/EIA-EIA/113613/2009), and to Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, also for granting me with a two scholarships during the M.Sc. course.

To my co-workers and friends who frequented the room of the Arquitectura de Sistemas Computacionais group, for all the moments we shared.

I would also like to express my gratitude to Paolo Romano for his promptly help on issues with respect to the Appia Group Communication System, and to João Cachopo and Instituto Superior Técnico for providing access to a testing environment.

To my parents, I am heartily thankful for their support and guidance over the years, and specially for providing me with this opportunity. To Tânia, for her priceless love, support, patience and encouragement.

Finally, I wish to thank all my family and friends for being part of my life and their endless support.

Abstract

The traditional lock-based concurrency control is complex and error-prone due to its low-level nature and composability challenges. Software transactional memory (STM), inherited from the database world, has risen as an exciting alternative, sparing the programmer from dealing explicitly with such low-level mechanisms.

In real world scenarios, software is often faced with requirements such as high availability and scalability, and the solution usually consists on building a distributed system. Given the benefits of STM over traditional concurrency controls, Distributed Software Transactional Memory (DSTM) is now being investigated as an attractive alternative for distributed concurrency control.

Our long-term objective is to transparently enable multithreaded applications to execute over a DSTM setting. In this work we intend to pave the way by defining a modular DSTM framework for the Java programming language. We extend an existing, efficient, STM framework with a new software layer to create a DSTM framework. This new layer interacts with the local STM using well-defined interfaces, and allows the implementation of different distributed memory models while providing a non-intrusive, familiar, programming model to applications, unlike any other DSTM framework.

Using the proposed DSTM framework we have successfully, and easily, implemented a replicated STM which uses a Certification protocol to commit transactions. An evaluation using common STM benchmarks showcases the efficiency of the replicated STM, and its modularity enables us to provide insight on the relevance of different implementations of the Group Communication System required by the Certification scheme, with respect to performance under different workloads.

Keywords: Transactional Memory, Distributed Systems, Concurrency Control, Replication

Resumo

O controlo de concorrência tradicional com trincos é complexo e propenso a erros devido à sua natureza de baixo nível e falta de composicionalidade. A Memória Transacional por *Software* (MTS), herdada do mundo das Bases de Dados, afigura-se como uma alternativa excitante, onde o programador não lida explicitamente com mecanismos de baixo nível.

No mundo industrial, o *software* precisa de lidar com requerimentos tais como disponibilidade e escalabilidade, e a solução habitualmente consiste em construir sistemas distribuídos. Dados os benefícios da MTS em relação ao controlo de concorrência tradicional, a Memória Transacional Distribuída por *Software* (MTDS) está a ser investigada como uma alternativa atrativa ao controlo de concorrência distribuída.

O nosso objetivo a longo prazo é permitir que aplicações que explorem múltiplos fluxos de execução possam ser executadas num ambiente de MTDS de maneira transparente. Neste trabalho propomo-nos abrir caminho através da definição de uma infraestrutura modular para MTDS para a plataforma Java. Estendemos uma infraestrutura existente e eficiente para MTS com uma nova camada de *software* para criar uma infraestrutura para suporte de MTDS. Esta nova camada interage com a MTS local utilizando interfaces bem definidas, e permite a implementação de diferentes modelos de memória distribuída ao mesmo tempo que fornece uma interface familiar e não-intrusiva para as aplicações, ao contrário das outras infraestruturas para MTDS.

Utilizando a infraestrutura proposta, implementámos facilmente e com sucesso uma MTS replicada que utiliza protocolos de Certificação para confirmar transações. Uma avaliação efetuada com *benchmarks* comuns de MTS mostram a eficácia da MTS replicada, e a sua modularidade permite-nos proporcionar uma análise da relevância de diferentes implementações do Sistema de Comunicação em Grupo necessário aos protocolos de Certificação, em relação ao desempenho do sistema em diferentes cargas de trabalho.

Palavras-chave: Memória Transacional, Sistemas Distribuídos, Controlo de Concorrência, Replicação

Contents

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Problem . . . 2
1.3 Proposed Solution . . . 2
1.4 Contributions . . . 3
1.5 Publications . . . 3
1.6 Outline . . . 4

2 Related Work **5**

2.1 Transactional Model . . . 5
2.2 Software Transactional Memory . . . 6
2.2.1 Semantics . . . 7
2.2.2 Implementation Strategies . . . 9
2.2.3 Programming Model . . . 10
2.2.4 Frameworks . . . 11
2.3 Distributed Software Transactional Memory . . . 18
2.3.1 Distributed Commit and Memory Consistency . . . 18
2.3.2 Frameworks . . . 22
2.4 Summary . . . 27

3 TribuSTM **29**

3.1 Introduction . . . 29
3.2 External Strategy . . . 30
3.3 TribuSTM and the In-Place Strategy . . . 32
3.4 Metadata and Arrays . . . 34
3.5 Evalution . . . 39
3.6 Summary . . . 40

4 Distributed Software Transactional Memory with TribuSTM 41
4.1 Introduction 41
4.2 Putting the D in TribuDSTM 42
4.2.1 Distributed Transactions 43
4.2.2 Distributed Objects 45
4.3 On the Implementation of a Distributed STM 48
4.4 Implementing a Replicated STM 49
4.4.1 Communication System 49
4.4.2 Distributed Transactions 50
4.4.3 Distributed Objects 52
4.4.4 Bootstrapping 53
4.5 Summary 55

5 Evaluation 57
5.1 Implementation Considerations 57
5.2 Experimental Settings 58
5.2.1 On the Implementation of Total Order Broadcast 58
5.3 Red-Black Tree Microbenchmark 59
5.4 STAMP Benchmarks 62
5.4.1 Intruder 62
5.4.2 Genome 66
5.4.3 Vacation 67
5.5 Summary 68

6 Conclusion 71
6.1 Concluding Remarks 71
6.2 Future Work 72

List of Figures

2.1 Two memory transactions (taken from [HLR10]). 7
2.2 Two memory transactions producing a *write skew* (taken from [HLR10]). . 8
2.3 Two STM programming models. 11
2.4 Standard sequential Java programming model. 12
2.5 DSTM2 programming model. 13
2.6 JVSTM's programming model. 14
2.7 Deuce's programming model. 16
2.8 Certification-based protocols. 20
2.9 DiSTM nodes (taken from [KAJ+08a]). 22
2.10 Components of a D^2STM replica (taken from [CRCR09]). 23
2.11 GenRSTM's programming model. 24
2.12 HyFlow node architecture (taken from [SR11]). 25
2.13 HyFlow's programming model. 26

3.1 Example of the modifications made for the external strategy. 31
3.2 Algorithm implemented with both interfaces. 33
3.3 Example of the modifications made for the in-place approach. 35
3.4 The different array types. 36
3.5 A multiarray strucutre in the the efficient solution. 37
3.6 Performance comparison between array types. 39

4.1 Our framework's architecture overview. 42
4.2 Pseudo-code of the Non-Voting Certification algorithm. 50
4.3 Pseudo-code of the Voting Certification algorithm. 51
4.4 Pseudo-code of the replicated objects' serialization algorithm. 53
4.5 Bootstrapping with the `@Bootstrap` annotation. 54

5.1 Throughput on the Red-Black Tree benchmark. 60
5.2 Breakdown on the Red-Black Tree benchmark. 61

5.3 Execution time on the Intruder benchmark. 63
5.4 Execution breakdown on the Intruder benchmark. 64
5.5 Abort rate on the Intruder benchmark. 65
5.6 Execution time on the Genome benchmark. 67
5.7 Execution time on the Vacation benchmark. 68

List of Tables

3.1 Comparison between primitive and transactional arrays. 38

4.1 Operations provided by the Reflexive API. 44
4.2 Operations provided by the Actuator API. 45
4.3 Operations provided by the Distributed Object API. 46
4.4 Interface provided by the GCS to the DT. 49
4.5 Interface provided by the DT to the GCS. 49

5.1 Parameterization of the Red-Black Tree microbenchmark. 59
5.2 Parameterization of the Intruder benchmark. 63
5.3 Parameterization of the Genome benchmark. 66
5.4 Parameterization of the Vacation benchmark. 67

Introduction

1.1 Motivation

Parallel programming cannot be ignored any more. Multi-core computers became mainstream as we hit the ceiling regarding processor clock speed. But parallel programming is inherently harder than sequential programming, as the programmer must reason about multiple flows of execution instead of just one. The traditional lock-based concurrency control is complex and error-prone due to its low-level nature and *composability* challenges [HMPJH05], so a suitable abstraction is needed.

Software transactional memory, STM henceforth, has been around for over fifteen years now, since Shavit and Touitou's original paper [ST95] back in 1995. It is based on the transactional model, which originated in the database world, where the unit of execution is an *atomic* block consisting of one or more operations. This block has *all-or-nothing* semantics, in the sense that either all its operations succeed and execute as if they are one single indivisible and instantaneous operation, or none executes at all. STM became a real alternative to lock-based synchronization, even industrially, as Intel keeps improving its C++ STM compiler [Int12a], the GNU Compiler Collection (GCC) gets support for STM [Fou12b], and Hardware Transactional Memory support from IBM [IBM12] and Intel [Int12b] processors is a reality.

Research focus has recently been directed to the application of STM in the distributed setting and, as of today, there has been already substantial research in the distributed STM field. To that purpose, existing STM frameworks have been extended with communication layers, schedulers, and mechanisms and/or protocols for cache coherence,

replication and contention management, and object lookup. Only just now an effort has began in designing a modular framework to support distributed STM.

On top of that, the programming models offered by previously proposed frameworks [KAJ+08a, CRCR09, CRR11a, SR11] are intrusive.

DiSTM [KAJ+08a], which is built on top of DSTM2 [HLM06], requires the programmer to implement the interface of shared *atomic objects* with a combination of reflection and runtime class synthesis in object factories.

In D²STM [CRCR09], based on JVSTM [CRS06], the programmer must wrap objects in *versioned boxes*, containers that keep a sequence of values representing their history. In more recent work, the GenRSTM framework [CRR11a] also wraps objects in versioned boxes and distributed objects must extend an existing framework-provided class.

HyFlow [SR11] requires all distributed classes to implement a specific interface, in addition to an explicit indirection through a *locator* service when we need to access a distributed object, with which we trade the object's unique identifier for its reference.

1.2 Problem

In the current distributed STM frameworks, the offered programming models are intrusive and highly dependent on the chosen STM framework. This intrusiveness, with respect to the transactional model, is mainly due to the need to deal with the distributed setting, which is not transparently handled by the runtime. Coupled with the fact that the APIs are heterogeneous, this makes the decision of using a specific framework a rather serious and committing one as new applications become tied to the provided API, and *porting* of existing applications is tedious work.

It is then desirable to have available a modular framework to support distributed STM. This framework should be composed by various components, supporting different implementations, which can be easily replaced without having to rewrite code. Current distributed STM frameworks are specifically tailored for a concrete distributed memory model, *e.g.*, replicated, which can not be replaced, thus inhibiting the fair comparison of different distributed STM models under the same circumstances.

Also very important, since people are inherently adverse to change, the programming model provided by such framework should remain as familiar as possible, *i.e.*, the traditional sequential model.

1.3 Proposed Solution

In this dissertation we address the problem of implementing a modular, efficient, distributed STM system that provides a non-intrusive programming model. Unlike previous proposals, which are tied to a specific distributed memory model, in our system

multiple models can be employed.

A distributed STM framework can be naturally implemented by extending an existing STM framework. TribuSTM, an extension of Deuce, is an efficient STM framework for Java. To use TribuSTM as a starting point for our distributed STM framework, we first need to provide support for transactional arrays. We then extend TribuSTM to support distributed STM by defining a new layer of software below. This layer of software, called Distribution Manager (DM), encapsulates a distributed memory model and protocols to commit distributed transactions, and interacts with TribuSTM in order to provide distributed STM. This interaction is made according to clearly defined APIs that allow different implementations of the DM to be plugged in.

TribuSTM provides a non-intrusive programming model to applications, by rewriting the bytecode of application classes. Our extension to support distributed STM keeps the programming model non-intrusive, unlike other distributed STM frameworks, by performing additional instrumentation.

1.4 Contributions

This dissertation describes the following contributions:

- a novel solution for transactional arrays in TribuSTM, and the required bytecode instrumentation;
- the definition of a software layer that extends TribuSTM to support distributed STM in a modular fashion, by interacting with TribuSTM through clearly defined interfaces;
- an implementation of a replicated STM on the aforementioned software layer;
- an experimental evaluation of the implementation on a common micro-benchmark, as well as more complex benchmarks; and
- insight on the relevance of the Group Communication System implementation used to support replicated STM, in the context of performance under different workloads.

1.5 Publications

The contributions have been published prior to this dissertation. The transactional arrays for TribuSTM have appeared in the paper "Efficient Support for In-Place Metadata in Transactional Memory", Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par), 2012 [DVL12], which has been awarded as a *distinguished paper*.

The distributed STM framework was featured in the paper “Uma Infraestrutura para Suporte de Memória Transacional Distribuída”, Proceedings of the Simpósio de Informática (INForum), 2012 [VDL12].

1.6 Outline

The rest of this documents is organized as follows. Chapter 2 provides an introduction to previous work related to our contributions. Chapter 3 introduces the TribuSTM framework, describing in detail our solution for transactional arrays. Next, in Chapter 4 introduces our extension to TribuSTM to support distributed STM, and presents in detail the defined APIs and the implementation of a replicated STM using our framework. Chapter 5 presents the results of our experimental evaluation study. Finally, Chapter 6 concludes this dissertation with an overview of its main points and future work directions.

Related Work

In this Chapter we describe previous related work which contextualizes and introduces our proposal.

The Chapter is structured as follows. In § 2.1 we introduce the transactional model and its properties. Next, in § 2.2, we present Software Transactional Memory and its semantics, possible implementation strategies, and the programming models provided by the existing state-of-the-art frameworks. The last section, § 2.3, introduces Distributed Software Transactional Memory, along with the issues which rise from the distributed environment. We finish with a survey of the state of the art in distributed commit and memory consistency protocols for full replication, and the existing frameworks.

2.1 Transactional Model

General-purpose programming languages already support powerful abstractions for sequential programming, but unfortunately the same is not true for parallel programming, where explicit synchronization reigns. But in the database world, database management systems (DBMS) have been harnessing the performance of parallelism for years.

The main culprit is the concept of a *transaction*, which lies at the core of the database programming model, and frees the programmer from worrying about parallelism. A transaction is a sequence of operations that execute with *all-or-nothing* semantics, *i.e.*, either all operations succeed and execute as if they were one single indivisible and instantaneous action, or none executes at all. In the database world, a transaction is characterised by four attributes: (1) atomicity; (2) consistency; (3) isolation; and (4) durability. These are known as the ACID properties [GR92, HLR10].

Atomicity. Requires that all actions that comprise a transaction complete successfully for the transaction itself to be successful. If one of these actions fails, the entire transaction fails and all actions have no apparent effect.

Consistency. Demands that a transaction starting from a consistent state evolves the database to another consistent state, which is a valid state according to the rules defined for the data.

Isolation. Requires that two concurrent transactions do not interfere with each other, *i.e.*, one transaction is not allowed to observe the internal, possibly inconsistent states resulting from the other's operations. This property can be, and often is, relaxed to improve performance, allowing different kinds of interference to take place.

Durability. Requires that when a transaction commits, its result will be made permanent and available to following transactions, in spite of possible errors, crashes or power losses.

As such, a transaction seems to be an interesting alternative to explicit synchronisation. One could wrap a sequence of operations manipulating shared memory in a transaction, and the atomicity property would guarantee that all operations complete successfully or the transaction would abort. And the isolation property would ensure the produced result to be the same as if the transaction was the only one executing, akin to the sequential fashion.

2.2 Software Transactional Memory

In order to bring transactions to memory, we must consider the differences between memory transactions and database transactions. For instance, database transactions access data in disk, while *Transactional Memory* (TM) accesses data in memory. The durability property then requires the database transaction to record its changes permanently in disk, but TM is not durable in that sense, since the data in memory only exists as long as the program is running, so this property is dropped in TM.

The enforce of consistent state after a transaction execution is also not specified, because it is highly dependent on the semantics of each particular application. And since TM is pushing its way into an already rich and generic environment, it has to cope with various ways of data access. This means that data can be accessed in mixed ways, transactionally or not, as opposed to database transactions where transactions are the only route of data access.

Software Transactional Memory, STM henceforth, has been around for over fifteen years

```
transaction {
  x = x + 1
  y = y + 1
}
```

(a) Transaction 1 ($\mathcal{T}_1$).

```
transaction {
  while (x = y)
```

(b) Transaction 2 ($\mathcal{T}_2$).

Figure 2.1: Two memory transactions (taken from [HLR10]).

now, since Shavit and Touitou's original paper [ST95] back in 1995. It instantiates a programming model for general-purpose programming, based on the previously described transactional model (§ 2.1). It became a real alternative to lock-based synchronization, even industrially, as Intel keeps improving its C++ STM compiler [Int12a], the GNU Compiler Collection (GCC) gets support for STM [Fou12b], and Hardware Transactional Memory support from IBM [IBM12] and Intel [Int12b] processors is a reality.

2.2.1 Semantics

To understand STM, we start by introducing and describing its behaviour. Being based on the transactional model, the basic unit of work is a transaction. A transaction is a group of memory read and write operations that execute as a single indivisible and instantaneous operation. If transactions execute atomically[1], that means that theoretically it is as if only one transaction is executing at any given point in time.

Consider the two transactions in Figure 2.1, where initially $x = y = 0$[2]. $\mathcal{T}_1$ increments the value of both the x and y variables, while $\mathcal{T}_2$ will loop forever if $x \neq y$. Conceptually no two transactions execute simultaneously so this means there are two possible outcomes: (1) $\mathcal{T}_1$ executes before $\mathcal{T}_2$; or (2) $\mathcal{T}_2$ executes before $\mathcal{T}_1$. Either way, $\mathcal{T}_2$ would never loop because when it executes the outcome for case (1) is $x = y = 1$, and for case (2) is $x = y = 0$.

There are several definitions used to describe the semantics of transactions. Inherited from the database literature, the *Serializability* criteria [EGLT76, HLR10] states that the result of the parallel execution of transactions must be equivalent to *some* sequential execution of those transactions. This means that the sequential order that transactions seem to run in is not required to be the actual real-time order in which they do. As long as the system's implementation guarantees the result of their execution to be *serializable*, it is free to reorder or interleave them.

This freedom to rearrange to order of execution of transactions in the Serializability criteria motivates the stronger *Strict Serializability* [HLR10]. In Strict Serializability if $\mathcal{T}_1$ actually completes before $\mathcal{T}_2$ this has to be reflected in the resultant serialization order, something that is not required in Serializability.

[1] It is worth pointing out that describing something as atomic means that it appears to the system as if it occurred instantaneously, and as such has both the atomicity and isolation properties.

[2] In all examples, unless stated otherwise, consider all variables with the initial value of 0 (zero).

```
transaction {
  x = x + y + 1
}
```

(a) Transaction 3 ($\mathcal{T}_3$).

```
transaction {
  y = x + y + 1
}
```

(b) Transaction 4 ($\mathcal{T}_4$).

Figure 2.2: Two memory transactions producing a *write skew* (taken from [HLR10]).

Another criteria is *Linearizability* [HW90, HLR10]. It states that every transaction is to be considered as a single atomic operation that appears to execute at some unique point during its lifetime.

A weaker criteria that can also be used is *Snapshot Isolation* [RFF06, HLR10]. The key idea is that each transaction takes a memory snapshot at its start and then performs all read and write operations on this snapshot. This criteria allows increased concurrency in databases by allowing non-serializable executions to commit.

Figure 2.2 depicts an example of a possible non-serializable execution. Consider two transactions $\mathcal{T}_3$ and $\mathcal{T}_4$ both doing the same computation, except that $\mathcal{T}_3$ stores the result in x while $\mathcal{T}_4$ stores it in y. Under the Serializability criteria two results are possible, namely $\mathtt{x} = 1 \wedge \mathtt{y} = 2$ or $\mathtt{x} = 2 \wedge \mathtt{y} = 1$, if $\mathcal{T}_1$ executes before $\mathcal{T}_2$ and *vice-versa*, respectively. Snapshot Isolation admits a third result, $\mathtt{x} = 1 \wedge \mathtt{y} = 1$, in which both transactions begin with the same snapshot and commit their disjoint update sets. This anomaly is called *write skew*.

Lastly, the *Opacity* criteria [GK08, HLR10] provides stronger guarantees about the memory consistency during a transaction's execution. It states that all operations performed by every committed transaction appear as if they happened at some single, indivisible point during the transaction lifetime, no operation performed by any aborted transaction is ever visible to other transactions (including live ones), and that every transaction (even if aborted) always observes a consistent state of the system. This last property is very important, as a transaction executing over an inconsistent state might lead to unexpected and/or incorrect behaviours, possibly leading to fatal errors, *e.g.*, a memory access violation [LC07].

STM is pushing its way into the general-purpose programming world, and as such it has to cope with various ways of data access. This means that data can be accessed in mixed ways, transactionally or not. The semantic criteria just presented do not consider the behaviour result from transactional and non-transactional access to the same data. This behaviour is defined by two approaches called *Weak* and *Strong Atomicity* [BLM06, HLR10].

The former only guarantees transactional semantics between transactions, *e.g.*, transactions lose their isolation regarding non-transactional code. The latter also guarantees the semantics between transactional and non-transactional code, that is, even non-transactional accesses to shared data are executed as a transaction.

2.2.2 Implementation Strategies

In the previous section we presented an overview of STM semantics. We will now introduce the main design choices to be considered when implementing such semantics. A straightforward implementation of the described semantics would be to use a global lock that transactions would acquire before starting, releasing after commiting. We would indeed be implementing the semantics, but there would be no advantage whatsoever in using STM over any other concurrency control methods, as no concurrency would actually be exploited.

Since transactions execute concurrently, in parallel even if possible, care has to be taken to mediate concurrent access to shared data. A conflict *occurs* when two transactions perform two concurrent accesses on the same data and at least one of them is a write. The system need not *detect* the conflict as soon as it occurs, but it must take measures to *resolve* it after its detection, by aborting one of the transactions for example. Occurrence, detection and resolution are the three events that motivate the two approaches to concurrency control, namely *Pessimistic* and *Optimistic*.

In *Pessimistic Concurrency Control* [HLR10], all three events occur upon data access. As soon as a transaction accesses some shared data, the conflict is immediately detected and resolved. This is typically implemented using a lock per each piece of shared data which a transaction acquires upon access, preventing other transactions from accessing the data. Due to the use of locks, implementations must take care to ensure that transactions progress. In this particular case, it must avoid *deadlocks*. A set of processes[3] is deadlocked if each process in the set is waiting to acquire a lock that only another process in the set can release.

With *Optimistic Concurrency Control* [HLR10], the detection and resolution can happen after the conflict occurrence. The conflicting transactions are therefore allowed to continue, but somewhere down the road the system will deal with the conflict before, or when, one of the transactions tries to commit. Instead of deadlocks, these implementations must be aware that they can lead to *livelocks*. A livelock situation is similar to a deadlock, but the processes involved are not in a waiting state, but effectively preventing each other from progressing. For instance, if a write to `x` conflict is resolved by aborting one of the transactions, that same transaction may be restarted and upon writing to `x` again, cause the abortion of the first transaction.

While conflict detection is trivial in Pessimistic Concurrency Control due to the use

[3] In the context of this dissertation, *thread* and *process* may be used interchangeably, as its differences are not relevant.

of locks, systems using the Optimistic Concurrency Control can apply various strategies. For instance, if conflicts are detected upon data access (like in Pessimistic Concurrency Control), the system is said to use *Eager Conflict Detection* [HLR10]. Other alternative is to detect conflicts in a *validation phase*, where a transaction checks all previously read and written locations for concurrent updates. This validation can be triggered at any time, even multiple times, during the transaction's lifetime.

Inversely to the Eager Conflict Detection is *Lazy Conflict Detection* [HLR10]. Upon the attempt of a transaction to commit, its read and write sets are checked for conflicts with other transactions.

The kind of accesses that conflict can be exploited by the Optimistic Concurrency Control. If the system only takes into account conflicts between active and already committed transactions, it is said to have *Committed Conflict Detection* [HLR10], which is impossible in Pessimistic Concurrency Control due to the use of locks. Conversely, if it identifies conflicts between concurrent running transactions it is said to have *Tentative Conflict Detection* [HLR10].

The Isolation property states that transactions do not observe the intermediate state of one another. As such, the last design choice is related to how a transaction manages its tentative updates. The first approach is called *Eager Version Management* or *Direct Update* [HLR10]. It is called so because the transaction modifies the actual data in memory, while maintaining an *undo log* holding the values it has overwritten to restore them in case the transaction aborts. This scheme implies the use of Pessimistic Concurrency Control as the transaction must exclusively acquire the locations for itself, for it is going to update them.

The alternative approach must be holding the updated values somewhere if it is not writing them on the fly. It is called *Lazy Version Management*, or *Deferred Update* [HLR10]. Instead of an undo log, each transaction buffers its tentative writes in a *redo log*. This log must be consulted by the transaction's read operations to obtain the most fresh value if it has been updated. If the transaction commits, the contents of the redo log are written in their respective locations. If the transaction aborts, it only needs to discard the buffer.

2.2.3 Programming Model

So far we have discussed the semantics of STM (§ 2.2.1) and some dimensions of its implementation where various strategies can be used (§ 2.2.2). While this is valuable knowledge from the point of view of the programmer, the programming model provided to use STM is also very important.

As we have seen, STM coexists with other models in the vast world of general-purpose programming. As a direct consequence of this, it is clear that when inside a transaction, memory accesses are not regular reads and writes.

```
do {
  startTx()
  writeTx(x, readTx(x) + 1)
  writeTx(y, readTx(y) + 1)
} while (commitTx())
```

(a) Basic programming model.

```
atomic {
  x = x + 1
  y = y + 1
}
```

(b) Atomic block.

Figure 2.3: Two STM programming models.

For instance, considering that transactions are *isolated* and that multiple transactions can execute concurrently, if one transaction writes to memory location l and another concurrent transaction subsequently reads l, it would read the value written by the former, clearly violating the isolation property. This means that memory reads and writes are not ordinary accesses, in order to ensure isolation.

As such, we can assume that there are two operations to read and write while in a transactional context, `readTx(x)` and `writeTx(x, val)` respectively. The first returns the value of the variable `x` which the current transaction should have access to, while the latter writes `val` into variable `x`. There should also be an operation to commit the current transaction, namely `commitTx()`, and `startTx()` must initialise whatever is needed to start a transaction.

In its most basic form, the programmer must explicitly declare a transaction using the `startTx`, `{read,write}Tx`, `commitTx` loop pattern (Figure 2.3a). Alternatively, the system can provide a more friendly `atomic` block, as seen in Figure 2.3b, that implicitly wraps its contents to manage a transaction, and replace the memory accesses within the block by calls to `{read,write}Tx`.

2.2.4 Frameworks

As we know, theory and practice don't always hold hands. In the next paragraphs we analyse the frameworks that have been presented as a result of, and tool for, research. Like our own work, these frameworks are for the Java programming language.

To support our analysis, for each framework we provide an example tailored to highlight several idiosyncrasies of each programming model. The examples consist of a list data structure implemented with linked `Node` objects, with an `insert` operation that inserts a new `Node` at the first position of the list. This operation is to be executed as a transaction. In Figure 2.4 we can see the example implemented in the standard sequential Java programming model.

2.2.4.1 DSTM2

From the work of Herlihy *et al.* comes DSTM2 [HLM06]. This framework is built on the assumption that multiple concurrent threads share data objects. DSTM2 manages synchronization for these objects, which are called *Atomic Objects*. A new kind of thread

```
1  class Node {
2    int value;
3    Node next;
4
5    int getValue() {
6      return value;
7    }
8    void setValue(int v) {
9      value = v;
10   }
11   Node getNext() {
12     return next;
13   }
14   void setNext(Node n) {
15     next = n;
16   }
17 }
```

(a) `Node` class.

```
1  class List {
2    Node root = new Node();
3
4    boolean insert(int v) {
5      Node newNode = new Node();
6      newNode.setValue(v);
7      newNode.setNext(root.getNext());
8      root.setNext(newNode);
9    }
10 }
```

(b) `insert` method.

```
1 List list = ...;
2 int v = ...;
3 list.insert(v);
```

(c) Invoking `insert`.

Figure 2.4: Standard sequential Java programming model.

is supplied that can execute transactions, which access shared Atomic Objects, and provides methods for creating new Atomic Classes and executing transactions.

Perhaps the most jarring difference from the standard programming model lies on the implementation of the Atomic Classes. Instead of just implementing a class, this process is separated in two distinct phases:

Declaring the interface. First we must define an interface annotated as `@atomic` for the Atomic Class to satisfy. This interface defines one or more properties by declaring their corresponding *getter* and *setter*. These must follow the convention signatures `T getField()` and `void setField(T t)`, which can be thought as if defining a class field named `field` of type `T`. Additionally, this type `T` can only be *scalar* or atomic. This restriction means that Atomic Objects cannot have array fields, so an `AtomicArray<T>` class is supplied that can be used wherever an array of type `T` would be needed, in order to overcome this.

These methods, as implemented in the next phase, will play the roles of the previously

```
1  @atomic
2  interface INode {
3    int getValue();
4    void setValue(int v);
5    INode getNext();
6    void setNext(INode n);
7  }
```

(a) INode interface.

```
1  class List {
2    static Factory<INode> fact =
3      dstm2.Thread.makeFactory(INode.class);
4    INode root = fact.create();
5
6    void insert(int v) {
7      INode newNode = fact.create();
8      newNode.setValue(v);
9      newNode.setNext(root.getNext());
10     root.setNext(newNode);
11   }
12 }
```

(b) insert transaction.

```
1  List list = ...;
2  int v = ...;
3  dstm2.Thread.doIt(new Callable<Void>() {
4    public Void call() {
5      list.insert(v);
6      return null;
7    }
8  });
```

(c) Invoking insert.

Figure 2.5: DSTM2 programming model.

presented `{read,write}Tx` operations in their respective fields. In Figure 2.5a we define the `INode` interface on these terms.

Implementing the interface. The interface is then passed to a transactional factory constructor that returns a transactional factory capable of creating `INode` instances, which is charged with ensuring that the restrictions presented in the previous phase are met. This factory is able to create classes at runtime using a combination of reflection, class loaders, and the *Byte Code Engineering Library* (BCEL) [Fou12a], a collection of packages for dynamic creation or transformation of Java class files. This means that Atomic Objects are no longer instantiated with the `new` keyword, but by calling the transactional factory's `create` method.

In the example in Figure 2.5b, the transactional factory is obtained in line 2, and an atomic object is created in line 7.

Also important is how transactions are defined. In DSTM2 any method can be a transaction, and the latter are very similar to regular methods. The major difference is that Atomic Objects are instantiated by calling the factory's `create` method (see Figure 2.5b).

Lastly, in Figure 2.5c, we inspect how does invoking a method differ from invoking a transaction. DSTM2 supplies a new `Thread` class that is capable of executing methods as transactions. Specifically, its `doIt` method receives a `Callable<T>` object whose `call`

```java
class Node {
  VBox<Integer> value = new VBox<Integer>(new Integer(0));
  VBox<Node> next = new VBox<Node>(null);

  int getValue() {
    return value.get();
  }
  void setValue(int v) {
    value.put(v);
  }
  Node getNext() {
    return next.get();
  }
  void setNext(Node n) {
    next.put(n);
  }
}
```

(a) `Node` class.

```java
class List {
  Node root = new Node();

  @Atomic
  boolean insert(int v) {
    Node newNode = new Node();
    newNode.setValue(v);
    newNode.setNext(root.getNext());
    root.setNext(newNode);
  }
}
```

(b) `insert` transaction.

```java
List list = ...;
int v = ...;
list.insert(v);
```

(c) Invoking `insert`.

Figure 2.6: JVSTM's programming model.

method body will be executed as a transaction, wrapped in the `{start,commit}Tx` loop.

All things considered, DSTM2's programming model is very intrusive when compared to the sequential model in Figure 2.4. Atomic Classes cannot be implemented directly, instead an `@atomic` interface must be declared (Figure 2.5a). The instantiation of Atomic Objects is not done through the `new` keyword, but by calling the `create` method of the transactional factory (Figure 2.5b). And finally, to start a transaction is a rather verbose process. We wrap the transaction's body in the `call` method of a `Callable` object that is passed as argument to the `dstm2.Thread.doIt` method (Figure 2.5c).

Not apparent in Figure 2.5 but nonetheless important is the issue of arrays in Atomic Objects, as they need to be replaced by instances of the `AtomicArray<T>` class.

2.2.4.2 JVSTM

Cachopo *et al.* proposed *Versioned Boxes* as the basis for memory transactions [CRS06]. They leverage *Multiversion Concurrency Control* [BG83] to avoid the abortion of read-only transactions by using special locations that keep a tagged sequence of values – Versioned Boxes. This sequence of values is the history of the box, whose changes have been made by committed transactions and tagged accordingly.

Versioned Boxes enable the serialization of read-only transactions based on the values the read boxes held at the transactions' start. By only reading values consistent with the version at its start, a read-only transaction is serialized before any updating transactions that eventually commit while the read-only transaction is executing.

Implementing classes to be modified through transactions is more straightforward than in DSTM2, as the only difference lies in the use of the Versioned Boxes, which are implemented as the `VBox` class. Any object that we intend to share is wrapped in a `VBox`, as depicted in Figure 2.6a, and its accesses replaced by the use of the boxes' `get` and `put` methods. These methods will act as the `{read,write}Tx` operations, and `get` will access the expected version of the box according to the current transaction. It is worth clarifying that calls to either `get` or `set` outside the context of a transaction will execute as an on the fly created transaction comprised of only one operation, the read or write respectively, thus achieving *Strong Atomicity* (§ 2.2.1).

While the use of the *Decorator Pattern* [GHJV94] makes a lot of sense in this case, using primitive arrays presents some issues. Take `int[]` for example. It can be used as a `VBox<Integer>[]`, in which case the actual array is not under the control of the STM and therefore the programmer must take care of synchronization by his own means. This approach incurs in high overhead as each array position now holds an object and its history, instead of a simple `int`.

Instead of wrapping each array position with a `VBox`, the entire array can be versioned as one, as in `VBox<int[]>`. This alternative is only suitable for read-oriented workloads, as *disjoint* writes to the array will conflict, and keeping the array versions is very expensive because a complete copy of the array is kept for each version, even when only a single position was written.

Transaction definition, as seen in Figure 2.6b, is as simple as annotating the methods that should be executed as a transaction with `@Atomic`. These methods are then post-processed, having their bodies wrapped with the boilerplate transaction calls. Therefore, transaction invocation is no different than regular method invocation (Figure 2.6c).

JVSTM's programming model certainly is a step forward in terms of invasiveness compared to DSTM2's. Programming classes whose objects are to be accessed within a

```
1  class Node {
2    int value;
3    Node next;
4
5    int getValue() {
6      return value;
7    }
8    void setValue(int v) {
9      value = v;
10   }
11   Node getNext() {
12     return next;
13   }
14   void setNext(Node n) {
15     next = n;
16   }
17 }
```

(a) `Node` class.

```
1  class List {
2    Node root = new Node();
3
4    @Atomic
5    boolean insert(int v) {
6      Node newNode = new Node();
7      newNode.setValue(v);
8      newNode.setNext(root.getNext());
9      root.setNext(newNode);
10   }
11 }
```

(b) `insert` transaction.

```
1  List list = ...;
2  int v = ...;
3  list.insert(v);
```

(c) Invoking `insert`.

Figure 2.7: Deuce's programming model.

transaction is very similar to programming regular classes.The difference lies in wrapping fields that will potentially be accessed concurrently within `VBox`es, as depicted in Figure 2.6a. As such, access to these fields must be done through their `get` and `put` methods, as opposed to the usual read and assignment.

Transaction demarcation is done at method definition, using the `@Atomic` annotation as in Figure 2.6b, instead of at method calling as in DSTM2. The use of such annotation abstracts the `{start,commit}Tx` loop in a very compact way, akin to the `atomic` block (Figure 2.3b). But the requirement to access boxes through their specific methods (`get`, `put`) ultimately fails to hide the `{read,write}Tx` operations.

2.2.4.3 Deuce

Product of more recent work is the Deuce framework [KSF10]. Korland *et al.* aimed for an efficient Java STM framework that could be added to existing applications *without* changing its compilation process or libraries.

In order to achieve such non-intrusive behaviour, it relies heavily on Java Bytecode

manipulation using ASM [Con12], an all-purpose Java Bytecode manipulation and analysis framework that can be used to modify existing classes or dynamically generate classes, directly in binary form. This instrumentation is performed dynamically as classes are loaded by the JVM using a Java Agent [Ora12]. Therefore, implementing classes to be modified through transactions is no different from regular Java programming (Figure 2.7a), as Deuce will perform all the necessary instrumentation of the loaded classes.

To tackle performance-related issues, Deuce uses `sun.misc.Unsafe` [sun12], a collection of methods for performing low-level, unsafe operations. Knowing this is going to be relevant in the following paragraphs describing the instrumentation performed, namely for why fields are identified by the Deuce runtime the way they are. Using `sun.misc.Unsafe` allows Deuce to directly read and write the memory location of a field f given the $\langle \mathcal{O}, f_o \rangle$ pair, where $\mathcal{O}$ is an instance object of class $\mathcal{C}$ and f_o the relative position of f in $\mathcal{C}$. This pair uniquely identifies its respective field, thus it is also used by the STM implementation to log field accesses.

As a framework, Deuce allows to plug in custom STM implementation, by implementing a `Context` interface which provides operations whose semantics are that of `{start,read,write,commit}Tx`, presented in the beginning of § 2.2.3.

We now briefly present the manipulations performed by Deuce. For each field f in any loaded class $\mathcal{C}$, a synthetic constant field is added, holding the value of f_o. In addition to the synthetic field, Deuce will also generate a pair of synthetic accessors, a $\mathcal{G}_f$ *getter* and $\mathcal{S}_f$ *setter*. These accessors encapsulate the `{read,write}Tx` operations, by delegating the access to the Deuce runtime (the `Context` implementation). They pass along the respective $\langle$`this`$, f_o \rangle$ pair, so the runtime effectively knows which field is being accessed and can read and write its value using `sun.misc.Unsafe`.

Other important instrumentation is the duplication of *all* methods. For each method m Deuce will create a synthetic method m_t, a copy of method m, to be used when in the context of a transaction. In m_t, read and write accesses to any field f are replaced by calls to the synthetic accessors $\mathcal{G}_f$ and $\mathcal{S}_f$, respectively. Besides the rewriting of field accesses, method calls within m_t are also instrumented. Each call to any method m' is replaced by a call to its transactional synthetic duplicate m'_t. The original method m remains unchanged, to avoid any performance penalty on non-transactional code as Deuce provides *Weak Atomicity* (§ 2.2.1).

This duplication has one exception. Each method m^a annotated with `@Atomic` is to be executed as a transaction (Figure 2.7b). Therefore, after the creation of its m^a_t synthetic counterpart, m^a is itself instrumented so that its code becomes the invocation of m^a_t wrapped in the `{start,commit}Tx` transactional loop. The practical effect of this is that invoking a transaction is simply calling a method, as seen in Figure 2.7c, provided that the programmer annotates the method with `@Atomic`, of course.

In retrospective, Deuce is optimal regarding programming model intrusion, only requiring the `@Atomic` annotation when compared to the sequential model in Figure 2.4. Leveraging STM on an existing application using Deuce requires only the annotation of the desired methods, as all these transformations are performed behind the scenes dynamically at class loading.

2.3 Distributed Software Transactional Memory

Similarly to STM, *Distributed Software Transactional Memory* (DSTM) has been proposed as an alternative for distributed lock-based concurrency control. As STM proves to be a viable approach to concurrency control, in order to be used in the enterprise world it is faced with requirements such as *high availability* and *scalability*. And although STM has received much interest over the last decade in chip-level multiprocessing, only recently focus on the distributed setting has begun in order to enhance dependability and performance.

Several dimensions need to be addressed in order to enable DSTM. First and foremost some sort of *communication layer* is needed, but where will the data be *located*? How are transactions *validated* and *committed* in the distributed setting? Other dimensions that need not necessarily be explored include data *replication* and data and/or transaction *migration*. As we will see in this section, several answers to these questions have been proposed. The communication layers range from Group Communication Systems (GCS) to regular network messaging. Data is either centralized, distributed or fully replicated, with transaction validation and commit protocols tailored for each specific model.

Initial work in this field has been focused on the scalability requirement [MMA06, BAC08, KAJ+08a], while more recent investigation has shifted to replication and the associated memory consistency protocols [CRCR09, CRR10, CRR11b, CRR11c]. Some effort has also been made in designing DSTM frameworks, such as DiSTM [KAJ+08a], HyFlow [SR11], D^{2}STM [CRCR09] and GenRSTM [CRR11a], with the last two addressing replication.

In the following subsection we present an overview of the distributed commit and memory consistency protocols state of the art targeting full data replication.

2.3.1 Distributed Commit and Memory Consistency

While the transaction concept bridges the world of Databases and (D)STM, memory transactions' execution time is significantly smaller than database transactions. Memory transactions only access data in main memory, thus not incurring in the expensive secondary storage accesses characterizing the latter. Furthermore, SQL parsing and plan

optimization are also absent in (D)STM. On the other hand this increases the relative cost of remote synchronization, which should be minimized.

Nonetheless, literature on replicated and distributed databases represents a natural source of inspiration when developing memory consistency algorithms for DSTM. Certification-based protocols, in a full replication environment, allow the localization of transaction execution[4], only requiring synchronization among replicas at transactions' commit time. In the case of read-only transactions no remote synchronization is actually needed. For write transactions the key ingredients are deferring the updates (§ 2.2.2) of a transaction until it is ready to commit, and relying on a GCS providing a *Total Order Broadcast* (TOB) [DSU04] primitive to disseminate transactions at commit time.

The TOB primitive has the following properties[5]:

- *Validity*: if a correct process TO-broadcasts a message m, then it eventually TO-delivers m;
- *Uniform Agreement*: if a process TO-delivers a message m, then all correct processes eventually TO-deliver m;
- *Uniform Integrity*: for any message m, every process TO-delivers m at most once, and only if m was previously TO-broadcast by $sender(m)$; and
- *Uniform Total Order*: if processes p and q both TO-deliver messages m and m', then p TO-delivers m before m', if and only if q TO-delivers m before m'.

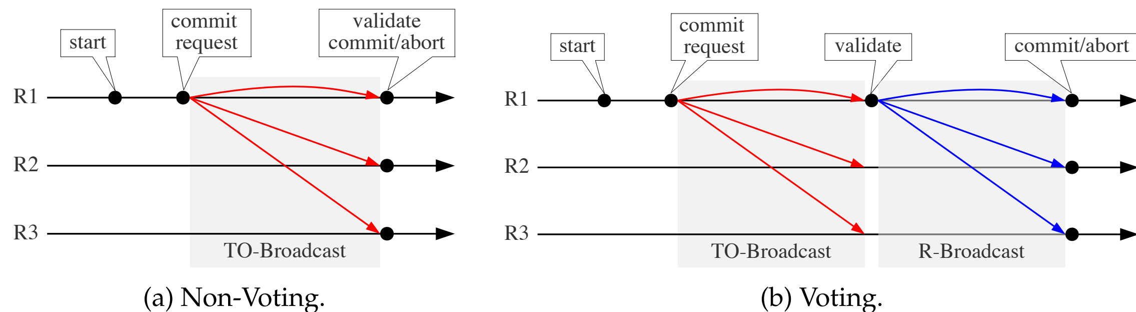
Being uniform means that the property does not only apply to correct processes, but also to faulty ones. For instance, with Uniform Total Order, a process is not allowed to deliver any message out of order, even if it is faulty. A broadcast primitive that satisfies all these properties except (Uniform) Total Order is called a *(Uniform) Reliable Broadcast*, (U)RB [DSU04].

We now present certification-based protocols from the literature.

Non-Voting Certification [AAES97]. When a transaction $\mathcal{T}$ executing at replica $\mathcal{R}$ enters the commit phase, it TO-broadcasts both its write set $\mathcal{WS}$ and read set $\mathcal{RS}$. This means that each replica is able to independently validate and abort or commit $\mathcal{T}$, as they are in possession of all the necessary information, *i.e.*, both $\mathcal{WS}$ and $\mathcal{RS}$. Note that given the total ordering of the deliveries, all replicas will process all transactions in the same order, so the result of any transaction's validation will be the same on all replicas. A sketch of this Certification protocol can be seen in Figure 2.8a.

[4] All transaction's operations execute locally in the single replica where the transaction began executing.

[5] We denote TO-broadcast and TO-deliver when broadcasting and delivering messages with the TOB primitive. R-broadcast and R-deliver are analogous but for the Reliable Broadcast primitive.



(a) Non-Voting. (b) Voting.

Figure 2.8: Certification-based protocols.

Voting Certification [KA98]. The previous scheme requires as single communication round to commit a transaction, by dissemination both $\mathcal{WS}$ and $\mathcal{RS}$. The latter is typically much larger than the former, thus this protocol explores the trade-off of exchanging potentially much smaller messages (since $\mathcal{RS}$, typically much larger than $\mathcal{WS}$, is not disseminated) at the expense of requiring two communication rounds instead of just one. It works as follows. When a transaction $\mathcal{T}$ executing at replica $\mathcal{R}$ reaches the commit phase, it *only* TO-broadcasts $\mathcal{WS}$. Unlike the Non-Voting Certification only $\mathcal{R}$ is capable of validating $\mathcal{T}$, as it is the only replica in possession of both $\mathcal{WS}$ and $\mathcal{RS}$. If $\mathcal{R}$ detects a conflict during the validation of $\mathcal{T}$, it R-broadcasts a message a (abort) notifying other replicas to discard $\mathcal{WS}$ and aborting $\mathcal{T}$ on $\mathcal{R}$. Else, a commit notification c is R-broadcasted instead, triggering the application of $\mathcal{WS}$ on all replicas. The notification messages a and c need only be reliably broadcasted (which doesn't ensure total ordering, thus being cheaper) because $\mathcal{T}$ was already serialized by the TO-broadcast of $\mathcal{WS}$. Figure 2.8b provides a delineation.

Bloom Filter Certification (BFC) [CRCR09]. As previously stated, the relative overhead of remote synchronization is much higher in DSTM. This extension of the Non-Voting scheme proposes the use of Bloom Filters to reduce the size of the broadcasted messages, as the efficiency of the TOB primitive is known to be strongly affected by the size of the exchanged messages [KT96, RCR08]. A Bloom Filter [BM03, CRCR09] is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible but false negatives are not, *i.e.*, a query returns "may be in the set" or "definitely not in the set". The Bloom Filter is used in this algorithm as follows. Before $\mathcal{T}$ is TO-broadcasted, $\mathcal{RS}$ is encoded in a Bloom Filter whose size is computed to ensure that the probability of a transaction abort due to a Bloom Filter's false positive is less than a user-tunable threshold. The validation of transactions checks whether their Bloom Filter contains any data item updated by concurrent transactions. This information (the data items updated by concurrent transactions) needs to be buffered by the system and periodically garbage-collected.

Asynchronous Lease Certification (ALC) [CRR10]. Certification schemes are inherently optimistic: transactions are validated on commit time and may be re-executed an unbounded number of times due to conflicts, leading to an undesirably high abort rate. This is particularly prominent in workloads comprising both short and long-running transactions, in which the latter may be repeatedly and unfairly aborted due to conflicts with a constant flow of short-running transactions. ALC tackles these issues using the concept of *asynchronous lease*. Informally, one can think of a lease as a token which gives its holder the privileges to manage a given subset of the whole data set. Leases are said to be asynchronous because they are detached from the notion of time, *i.e.*, leases are granted on an acquire/release-style protocol. The ALC algorithm works as follows. In order for a transaction $\mathcal{T}$ to commit, the replica $\mathcal{R}$ where $\mathcal{T}$ executed must first successfully obtain the leases for its accessed items. If $\mathcal{T}$ is found to have accessed outdated values, it is aborted and re-executed without relinquishing the leases. This ensures that $\mathcal{T}$ will not be aborted again due to remote conflicts, as no other replica can update the items protected by the held leases, assuming $\mathcal{T}$ deterministically accesses the same items as in the first execution.

Polymorphic Self-Optimizing Certification (PolyCert) [CRR11c]. The Voting and Non-Voting protocols are based on the trade off between communication steps and message size, respectively. Ergo both are designed to ensure optimal performance in different workload scenarios and they can exhibit up to $10\times$ difference in terms of maximum throughput [CRR11c]. To deal with this PolyCert supports the coexistence of the Voting and Non-Voting/BFC protocols simultaneously, by relying on machine-learning techniques to determine, on a per transaction basis, the optimal certification strategy to be adopted.

Speculative Certification (SCert) [CRR11b]. With high probability, messages broadcasted in a local-area network are received totally ordered (e.g., when network broadcast or IP-multicast are used). This property, called *spontaneous total order*, is exploited by the *Optimistic Atomic Broadcast* (OAB) primitive in order to deliver messages fast [PS03]. SCert leverages on the OAB's *optimistic delivery* (corresponding to the spontaneous total order) to overlap computation with the communication needed to ensure the total ordering. Informally, it works as follows. As soon as a transaction $\mathcal{T}$ is optimistically delivered, SCert speculatively certifies $\mathcal{T}$ instead of waiting for its final delivery as conventional certification protocols. If validation is successful, $\mathcal{T}$ is said to be *speculatively committed*, that is, *speculative versions* of the data items updated by $\mathcal{T}$ are created. Eventually $\mathcal{T}$ is *finally committed* (its versions are actually written) if the final delivery matches the optimistic. If it does not match, $\mathcal{T}$ is aborted unless it is still successfully certified in the new serialization order. Speculative versions of data items are immediately visible to new transactions, hence tentatively serializing these transactions after the speculatively-committed ones. This allows an overlap between computation and communication by

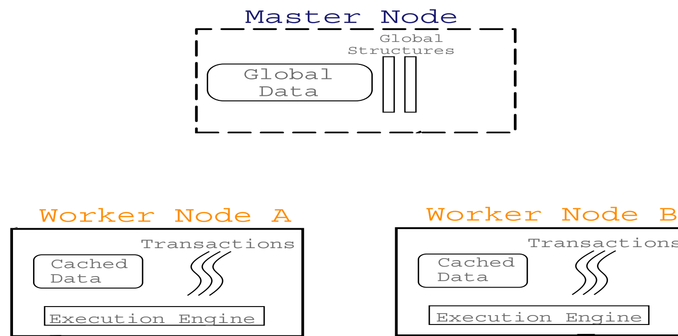


Figure 2.9: DiSTM nodes (taken from [KAJ+08a]).

certifying transactions while the AOB computes the final order, and to detect conflicts earlier (e.g., upon a read access, if a fresher speculative version of the data item exists) avoiding wasted computation and time on transactions doomed to abort.

2.3.2 Frameworks

With the recent thrust in DSTM investigation a handful of different memory consistency schemes have been proposed. It is no surprise that frameworks aimed at facilitating the development, testing and evaluation of these different protocols were also proposed. We now describe the existing DSTM frameworks by analysing the new components added to support the distributed settings and their programming model.

2.3.2.1 DiSTM

In [KAJ+08a] Kotselidis *et al.* designed a framework for easy prototyping of transactional memory coherence protocols called DiSTM, which is built on top of DSTM2 [HLM06]. In this work, one of the nodes acts as a *master node* where global data is centralised, while the rest of the nodes act as *workers* and maintain a cached copy of that global data (Figure 2.9). Three protocols are presented, whose objective is to maintain the coherence of these caches.

Communication is made on top of the ProActive framework [BBC+06]. The key concept of the ProActive framework is *Active Objects*. Each Active Object has its own thread of execution and can be distributed over the network.

Application-level objects are located in the master node but cached copies exist in all worker nodes as already stated. Transactions are validated at commit time and they update the global data stored in the master node upon successful validation. The master node then eagerly updates all cached copies, and any running transactions are aborted if they fail to validate against the incoming updates.

If the programming model of DiSTM differs from the one of DSTM2, in which it is based, they have not made it apparent in [KAJ+08a]. Even so, deducing from the fact

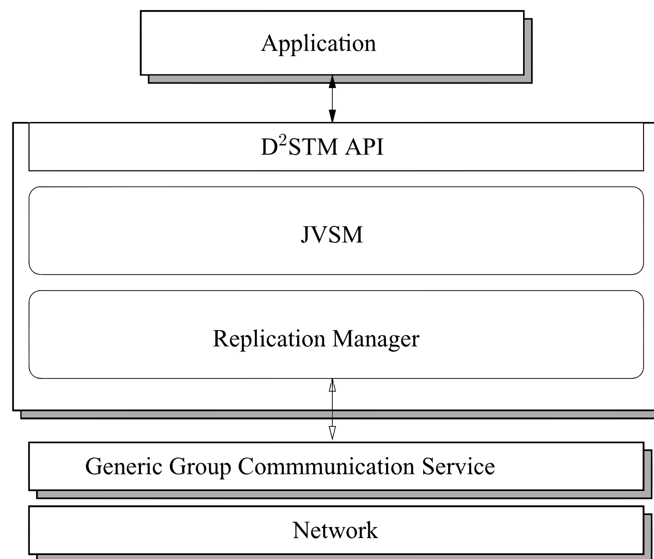


Figure 2.10: Components of a D^{2}STM replica (taken from [CRCR09]).

that data is centralized in the master node and worker nodes maintain local caches of all objects (in [KAJ$^+$08a] the authors refer to [KAJ$^+$08b] for details), I believe that it is safe to say that the programming model of DiSTM is the same as in DSTM2.

2.3.2.2 D^{2}STM

Couceiro *et al.* in D^{2}STM [CRCR09], based on JVSTM [CRS06], have the objective of leveraging replication not only to improve performance, but also to enhance dependability. Therefore data is replicated across all nodes of the distributed system, and their contribution features a *bloom filter-based* replica coordination protocol that enables high compression rates on the messages exchanged between nodes at the cost of a tunable increase in the probability of transaction abort due to false positives.

The components which constitute a D^{2}STM node can be seen in Figure 2.10. Communication is achieved through a group communication system implementing a *Total Order Broadcast* (TOB) [DSU04] primitive.

In between this communication layer and the STM lies the core component of this system, the replication manager, which implements the distributed coordination protocol required for ensuring replica consistency. It integrates with the STM, whose API was extended to accommodate so, by having the ability to inspect the internals of transaction execution, explicitly triggering transaction validation and atomically applying the write sets of remotely executed transactions.

D^{2}STM's programming model is not presented in [CRCR09], but since it is aimed at fully replicated systems we can assume that if all objects are replicated, its model is the same of JVSTM, on which it is based. If the only objects to be replicated are specified by

```
class Node extends GenRSTMObject {
  Box<Integer> value = STMRuntime.getRuntime().getBoxFactory().createBox(
    new Integer(0));
  Box<Node> next = STMRuntime.getRuntime().getBoxFactory().createBox(
    null);

  int getValue() {
    return value.get();
  }
  void setValue(int v) {
    value.put(v);
  }
  Node getNext() {
    return next.get();
  }
  void setNext(Node n) {
    next.put(n);
  }
}
```

(a) `Node` class.

```
class List {
  Node root = new Node();

  boolean insert(int v) {
    STMRuntime.getRuntime().begin();
    Node newNode = new Node();
    newNode.setValue(v);
    newNode.setNext(root.getNext());
    root.setNext(newNode);
    STMRuntime.getRuntime().commit();
  }
}
```

(b) `insert` transaction.

```
List list = ...;
int v = ...;
list.insert(v);
```

(c) Invoking `insert`.

Figure 2.11: GenRSTM's programming model.

the programmer, then there has to be some (minor) difference.

2.3.2.3 GenRSTM

GenRSTM [CRR11a], like D^{2}STM, is a framework for replicated STMs. From the perspective of the components added to support the replicated setting, it is identical to D^{2}STM and seems to be an enhanced version of the latter. Specifically, the STM layer can be exchanged as long as it provides an API like the one of JVSTM, based on boxes, and in his thesis TL2 [DSS06] was also used besides JVSTM.

In his Ph.D. thesis [Car11] several examples are supplied, thus the programming model is apparent. The fact that multiple backends can be used in the STM layer is

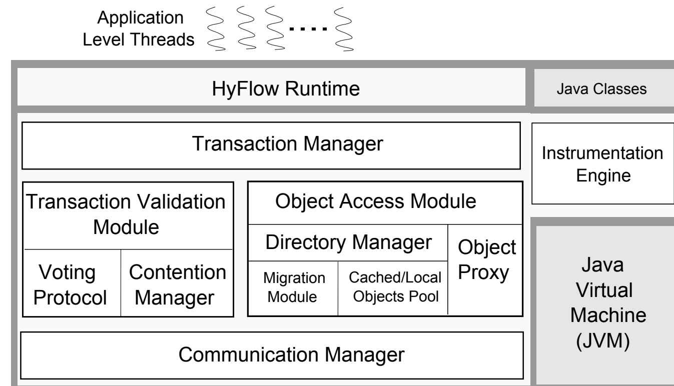


Figure 2.12: HyFlow node architecture (taken from [SR11]).

reflected on the creation of a new `Box`, now achieved through a `BoxFactory` class that returns the `Box` object of the STM implementation being used in the STM layer. A noteworthy detail is that the programmer explicitly marks which objects are replicated, by having their class extend the framework-supplied class named `GenRSTMObject` (Figure 2.11a).

Transactions are explicitly programmed with the `begin` and `commit` operations, as seen in Figure 2.11b, and object access is identical to JVSTM, using the boxes' `get` and `put`. Thus, the programming model is reminiscent of the basic programming model concept in Figure 2.3a.

2.3.2.4 HyFlow

HyFlow [SR11] is Java framework for DSTM, with pluggable support for directory lookup protocols, transactional synchronization and recovery mechanisms, contention management policies, cache coherence protocols, and network communication protocols.

Figure 2.12 sketches HyFlow's architecture. It is comprised of five main components:

Transaction Manager This component embodies the local STM algorithm;

Instrumentation Engine Modifies class code at runtime *à la* Deuce, for example, modifying annotated methods to support transactional behaviour;

Object Access Module The Object Access Module not only provides access to the objects owned by the current node, but is able to locate and send access requests to remote objects;

Transaction Validation Module Ensures data consistency, validating transactions upon their completion. It encapsulates the contention management policy and the distributed commit protocol; and

Communication Manager The Communication Manager enables the network communication between the various nodes of the system.

Since objects can be distributed over the network, normal references cannot be used to access them. As such, HyFlow demands that any distributed class implements the

```
class Node
implements IDistinguishable
{
  int value;
  String nextId;
  String id;

  Node(String id) {
    this.id = id;
  }
  Object getId() {
    return id;
  }
  @Remote
  int getValue() {
    return value;
  }
  @Remote
  void setValue(int v) {
    value = v;
  }
  @Remote
  String getNext() {
    return nextId;
  }
  @Remote
  void setNext(String n) {
    nextId = n;
  }
}
```

(a) Node class.

```
class List {
  //head sentinel node identifier
  String final HEAD = ...;

  @Atomic
  boolean insert(int v) {
    Locator locator =
      HyFlow.getLocator();
    Node root =
      (Node) locator.open(HEAD);
    String newNodeId = ...; //gen. UOID
    Node newNode = new Node(newNodeId);
    newNode.setValue(v);
    newNode.setNext(root.getNext());
    root.setNext(newNodeId);
  }
}
```

(b) insert transaction.

```
List list = ...;
int v = ...;
list.insert(v);
```

(c) Invoking insert.

Figure 2.13: HyFlow's programming model.

IDistinguishable interface, which consists of a getId() method that returns an unique object identifier (UOID). Thus, where before fields held regular references, now they must maintain these UOIDs.

Distributed classes can provide remote methods which can be invoked regardless of their objects' location, *à la* Java RMI. These methods are defined by the programmer, annotating them with @Remote. These details are depicted in Figure 2.13a.

In order to obtain a reference to a distributed object, a Locator instance from the Directory Manager component, encapsulating the directory lookup protocol, is used to retrieve objects given their UOID (Figure 2.13b, lines 6-7).

Transactions are defined as @Atomic-annotated methods akin to previous work we already presented, as seen in Figure 2.13b, and invoking them is analogous to calling a

method (Figure 2.13c).

Regarding HyFlow's programming model we believe that the additional intrusion introduced when compared to Deuce's (§ 2.2.4.3), specifically the unique identification of objects with the `IDistinguishable` interface and their retrieval through the `Locator` object, can be hidden from the programmer using bytecode instrumentation.

2.4 Summary

This Chapter 2 presented the foundation of our work. We started by introducing the transactional model and its properties (§ 2.1), which led to Software Transactional Memory (STM), described in § 2.2. We covered its semantics in § 2.2.1 and overview implementation strategies that can be applied to realise STM (§ 2.2.2). We also examined how can STM be used by the programmer by introducing it's programming model (§ 2.2.3) and the existing state-of-the-art frameworks and programming model they provide (§ 2.2.4).

In § 2.3 we introduced DSTM and the new issues that arise from applying STM to the distributed setting. We also overview the state of the art in distributed commit and memory consistency protocols for full replication, in addition to frameworks to support DSTM.

From all the existing STM frameworks, Deuce stands out as providing an extremely simple and intuitive programming model. Unfortunately, and as we will see in the next Chapter, it does not allow the efficient implementation of all STM algorithms, most notably multi-version algorithms.

It is also apparent that *no existing* DSTM framework provides a programming model as simple as Deuce's. Also, each DSTM framework is tied to a concrete distributed memory model, thus not allowing the fair comparison of DSTM, on the same scenario, under different distributed memory models.

TribuSTM

This Chapter presents TribuSTM, an extension of the Deuce Java STM framework. We introduce Deuce and motivate the need for TribuSTM in § 3.1. In § 3.2 we discuss the metadata placement strategy as provided by Deuce, and afterwards describe the new strategy featured in TribuSTM (§ 3.3). In § 3.4 we present our solution for n-dimensional transactional arrays with the new metadata placement strategy. We finish with the evaluation of our contribution in § 3.5 and conclude in § 3.6.

3.1 Introduction

Over the last years, several STM algorithms have been proposed. These algorithms associate information to each memory location (or object reference) accessed within a transaction. This information, which we will from this point on refer to as *metadata*, is specific to each algorithm and may be constituted by, *e.g.*, locks, timestamps or lists of values.

Metadata can be stored either in an external data structure that associates the metadata with the corresponding memory location (*out-place* or *external* strategy), or alongside the memory location (dubbed *in-place* strategy).

The external strategy can be implemented using a hash table that pairs the memory location to its metadata. If the table is pre-allocated we can avoid the overhead of dynamic memory allocation, paying instead the cost of evaluating the hashing function. At some point we might have to resize the table which typically is a very heavy operation, unless we opt not to resize, in which case we must live with the limitations imposed by not resizing, *e.g.*, have two or more distinct memory locations pairing with the same metadata. The in-place strategy, in object-oriented programming, is typically implemented with a twist of the Decorator design pattern [GHJV94] by wrapping the targeted object inside a

container object which holds both the original object and its associated metadata. While this allows a very direct and efficient access to the metadata, it is highly intrusive to application code, which has to be rewritten in order to use the decorator's class instead. This approach does not cope well with primitive and array types, with the first having to be replaced by their object counterparts while the latter is a very strict structure to which we can not add metadata.

Deuce (§ 2.2.4.3) is an efficient framework for the Java programming language which allows the implementation of different STM algorithms using an external strategy for storing metadata. This approach is suitable for algorithms whose metadata is not strongly tied to the associated memory location, *e.g.*, locks or timestamps. Algorithms such as TL2 [DSS06], whose metadata consists of a single lock, can be efficiently implemented by resorting to an hash table without collision avoidance and using a very fast hashing function. This means that two memory location may in fact be mapped to the same lock, incurring in false sharing, which results in transactions conflicting in practice even though they actually should not.

While metadata sharing is not fatal in these cases, it becomes so if we take into account metadata that is strongly tied to its associated memory location. Consider a multi-version algorithm, *e.g.*, JVSTM [CRS06], that associates a list of values with each memory location, building an history of that location's values. Each of these histories is strongly tied to the corresponding location, and sharing them is semantically wrong. Hence, and because we can not afford false sharing, the hash table needs to treat collisions but this imposes a significant and unacceptable performance overhead [DVL12].

We can conclude that using the external strategy is acceptable when metadata is not strongly tied to its memory location, that is, the relationship between a location and metadata can be N–1. If there must be a 1–1 relation, then the external strategy is unsatisfactory and the in-place strategy is preferred.

In this Chapter we present an extension to Deuce, named TribuSTM [DVL12], that enables STM algorithms to be implemented with the in-place strategy *without* changing the API provided to application programmers, allowing the efficient implementation of algorithms that could not be implemented so before while maintaining the transparency Deuce is characterized by. Both the external and in-place strategies are supported, making this extension flexible and fully backwards compatible with already existing implementations of STM algorithms. This in-place strategy is achieved without using the Decorator pattern.

3.2 External Strategy

Deuce provides the STM algorithms with an unique identifier for an object field, a pair $\langle \mathcal{O}, f^o \rangle$, where $\mathcal{O}$ is the object reference and f^o the field's logical offset of a field f within

$\mathcal{O}$. This pair has two use cases, namely (1) it uniquely identifies a field from an object, hence it can be used by the STM algorithms as a key to any map implementation to associate object fields with transactional metadata; and (2) using `sun.misc.Unsafe` it is possible to read and write directly to the associated memory location.

STM algorithms implement a `Context` interface, providing methods whose semantics are that of `{start,read,write,commit}Tx`, presented in Figure 2.3a. The transactional access methods (`{read,write}Tx`) have $\langle \mathcal{O}, f^o \rangle$ as parameter, allowing them to retrieve the associated metadata from an external mapping table. The STM algorithm programmer is completely free concerning transactional metadata implementation, as there is no specific class hierarchy to extend, or any rules to follow. In truth, the existing TL2 implementation that comes bundled with Deuce, at this time, implements the mapping table as an array of integers, each of them used as a versioned lock.

```
1  class C {
2    int[] f1;
3
4    String f2;
5
6
7
8
9
10
11
12   C() {
13     f1 = new int[2];
14     ...
15   }
16   @Atomic int m1() {
17     f2 = "a";
18     ...
19   }
20
21
22
23
24   int[] m2(int i) {
25     f1[i] = f1[i] + 1;
26     ...
27   }
28
29
30
31
32
33 }
```

(a) Before.

```
class C {                                1
  int[] f1;                              2
  static final long f1_o;                3
  String f2;                             4
  static final long f2_o;                5
                                         6
  static {                               7
    f1_o = ...;                          8
    f2_o = ...;                          9
  }                                      10
                                         11
  C() {                                  12
    f1 = new int[2];                     13
    ...                                  14
  }                                      15
  int m1() {                             16
    // retry loop                        17
                                         18
  }                                      19
  int m1(Context ctx) {                  20
    ctx.writeTx(this, f2_o, "a");        21
    ...                                  22
  }                                      23
  void m2(int i) {                       24
    f1[i] = f1[i] + 1;                   25
    ...                                  26
  }                                      27
  int[] m2(int i, Context ctx) {         28
    int aux = ctx.readTx(f1, i);         29
    ctx.writeTx(f1, i, aux + 1);         30
    ...                                  31
  }                                      32
}                                        33
```

(b) After.

Figure 3.1: Example of the modifications made for the external strategy.

To summarise, Deuce performs the following instrumentation of the Java bytecode. Let $\mathcal{C} = \{f_1, \ldots, f_i, m_1, \ldots, m_j\}$ be a class $\mathcal{C}$ with fields $\{f_x : 1 \leq x \leq i\}$ and methods $\{m_y : 1 \leq y \leq j\}$. For each field f_x a new field f_x^o is added, whose value is the logical offset of f_x. For each method m_y a modified version m_y^t is added, in which reads and writes have been replaced by transactional accesses through the runtime. If m_y is `@Atomic`-annotated, it is replaced by a retry loop calling m_y^t in the context of a new transaction. After all transformations we have

$$\mathcal{C} = \{f_1, f_1^o, \ldots, f_i, f_i^o, \ldots, m_1, m_1^t, \ldots, m_j, m_j^t\}$$

Figure 3.1 provides an example, with the modifications highlighted. We can see the corresponding f^o field (Lines 3 and 5 in Figure 3.1b) for each field in the original class (Lines 2 and 4 in Figure 3.1a). For each method in the original class (Lines 16 and 24 in Figure 3.1a), we can see the corresponding m^t version on Lines 20 and 28 in Figure 3.1b. Additionally, as `m1` is annotated with `@Atomic` (Line 16, Figure 3.1a), we can see that its code has been replaced with the transactional loop (Lines 16 and 17, Figure 3.1b).

3.3 TribuSTM and the In-Place Strategy

To eliminate the indirection introduced by the external mapping between a field f and it's metadata, henceforth denoted f^m, the runtime must provide the STM algorithm not with $\langle \mathcal{O}, f^o \rangle$, but with f^m itself. To accomplish this we directly inject f^m in $\mathcal{C}$, to be used instead of $\langle \mathcal{O}, f^o \rangle$. Algorithms following the in-place approach implement a slightly altered API (`ContextMetadata`) regarding the transactional access methods, which have as parameter f^m instead of $\langle \mathcal{O}, f^o \rangle$.

This approach contrasts with the Decorator pattern [GHJV94], where primitive types must be replaced with their object equivalents (*e.g.*, an `int` field is replaced by an `Integer` object). Our transformation keeps the primitive-typed fields untouched, simplifying the interaction with non-transactional code, limiting the code instrumentation and avoiding the overheads of autoboxing.

Since Deuce supports multiple STM algorithms (which have different metadata), and algorithms are chosen at execution time through parameterization, one can question how does the instrumentation process "know" which metadata classes to inject. Our solution for this issue is twofold. For algorithms using the in-place strategy, their transactional metadata classes must extend a common super class, `TxMetadata`. Objects from this class hold the corresponding $\langle \mathcal{O}, f^o \rangle$ to read and write the associated field. The algorithm's implementation must also specify its metadata classes through a `@InPlaceMetadata` annotation. The instrumentation process inspects the `@InPlaceMetadata` annotation of the chosen algorithm to be aware of what to inject.

```
1  class MyContext implements Context {
2    void writeTx(Object obj, long f_o, T val) {
3      // obtain the metadata from the table
4      // do whatever...
5    }
6
7    T readTx(Object obj, long f_o) {
8      // obtain the metadata from the table
9      // do whatever...
10   }
11   ...
12 }
```

(a) External metadata interface.

```
1  @InPlaceMetadata(class=MyMetadata)
2  class MyContext implements ContextMetadata {
3    void writeTx(TxMetadata f_m, T val) {
4      // do whatever...
5    }
6
7    T readTx(TxMetadata f_m) {
8      // do whatever...
9    }
10   ...
11 }
```

(b) In-place metadata interface.

Figure 3.2: Algorithm implemented with both interfaces.

An example of the differences between the external and in-place strategy interfaces can be seen in Figure 3.2. In this example, the instrumentation process is aware that the injected f^m fields should be of type `MyMetadata`, as specified in the `@InPlaceMetadata` annotation (Line 1, Figure 3.2b). In the in-place strategy interface (Figure 3.2b), the `{read,write}Tx` callbacks take the metadata as a direct parameter (Lines 3 and 7, Figure 3.2b), while the external strategy, in Figure 3.2a, takes the $\langle \mathcal{O}, f^o \rangle$ pair (Lines 2 and 7) and needs to access the external table to fetch the metadata.

Also please note that we informally use `T` where there should be a version of each method for every Java primitive type.[1]

To summarise, TribuSTM performs the following instrumentation of the Java bytecode. Let $\mathcal{C} = \{f_1, \ldots, f_i, m_1, \ldots, m_j\}$ be a class $\mathcal{C}$ with fields $\{f_x : 1 \leq x \leq i\}$ and methods $\{m_y : 1 \leq y \leq j\}$. For each field f_x, besides f_x^o, a new field f_x^m is added to reference the transactional metadata object associated with f_x. Method transformations are essentially the same as in § 3.2, with the notable exception of *constructors* and *static initializers*. The creation and initialization of the f_x^m metadata fields' objects are injected in

[1] `int, long, float, double, short, char, byte, boolean` and `Object`.

these. The transactional versions m_y^t are adapted to call the transactional access methods using f_x^m instead of $\langle O, f_x^o \rangle$. After the transformations we have

$$\mathcal{C} = \{f_1, f_1^o, f_1^m, \ldots, f_i, f_i^o, f_i^m, \ldots, m_1, m_1^t, \ldots, m_j, m_j^t\}$$

Figure 3.3 exemplifies the result of the transformations as if executing the algorithm in Figure 3.2b. Besides the f^o fields are described before, we can see the additional f^m fields (Lines 4 and 7, Figure 3.3b), and their initialisation (Lines 23 and 24). In Line 31 the `writeTx` callback in depicted, taking metadata as an argument. Modifications with respect to arrays (*e.g.*, Lines 2, 15-21, 34, 38) are covered in the next Section.

3.4 Metadata and Arrays

In § 3.2 we have seen how the runtime uses $\langle \mathcal{O}, f^o \rangle$ in the external metadata strategy to uniquely identify field f in object $\mathcal{O}$. Array cells are also uniquely identified by the array reference $\mathcal{A}$ and the index i of that cell, $\langle \mathcal{A}, i \rangle$, as the index i already represents the cell's logical offset within $\mathcal{A}$. Therefore, from the runtime's point of view, array cells and fields are indistinguishable.

In § 3.3 we described the in-place metadata approach and its implementation, injecting f^m inside classes. This poses a problem for arrays, specially arrays of primitive types, given their very strict structure: each array cell contains a single value of a well defined type.

A naive solution for storing in-place metadata in arrays. An ideal solution would be to have an array in which cells are *records* with two members, *value* for the actual stored data and the *metadata* member. While we can not change the array structure itself to match our requisite, we can manipulate the type of the array cells. Let $\mathcal{T}$ be the set of all Java primivite types. For each $t \in \mathcal{T}$, let $Cell_t = \{v, v^m\}$ be a class with fields v and v^m for the stored value and metadata, respectively, where v is of type t (Figure 3.4b). We replace all t`[]` by $Cell_t$`[]` (a transactional array) and adjust the code accordingly.

To implement this solution, in addition to the already performed bytecode modifications to support in-place metadata in objects (§ 3.3), the following are also applied:

- replace type in class field declarations;
- replace type in local variable declarations;
- update signature of methods receiving and/or returning (multi)arrays;
- replace non-transactional accesses to t`[`i`]` with $Cell_t$`[`i`]`$.v$;
- pass the array cell metadata, $Cell_t$`[`i`]`$.v^m$, to the transactional access callbacks;

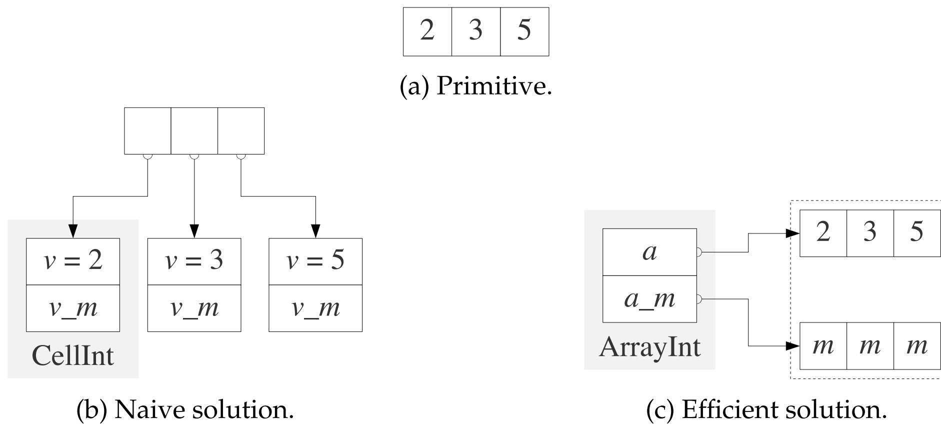
```
class C {
  int[] f1;


  String f2;








  C() {
    f1 = new int[2];






    ...


  }
  @Atomic int m1() {
    f2 = "a";
    ...
  }




  int[] m2(int i) {
    f1[i] = f1[i] + 1;
    ...
  }





}
```

(a) Before.

```
class C {
  ArrayInt f1;
  static final long f1_o;
  MyMetadata f1_m;
  String f2;
  static final long f2_o;
  MyMetadata f2_m;

  static {
    f1_o = ...;
    f2_o = ...;
  }

  C() {
    f1 = new ArrayInt();
    f1.a = new int[2];
    f1.a_m = new MyMetadata[2];
    for (int i = 0; i < 2; i++) {
      f1.a_m[i] =
        new MyMetadata(f1.a, i);
    }
    ...
    f1_m = new MyMetadata(this, f1_o);
    f2_m = new MyMetadata(this, f2_o);
  }
  int m1() {
    // retry loop

  }
  int m1(Context ctx) {
    ctx.writeTx(f2_m, "a");
    ...
  }
  ArrayInt m2(int i) {
    f1.a[i] = f1.a[i] + 1;
    ...
  }
  ArrayInt m2(int i, Context ctx) {
    int aux = ctx.readTx(f1.a_m[i]);
    ctx.writeTx(f1.a_m[i], aux + 1);
    ...
  }
}
```

(b) After.

Figure 3.3: Example of the modifications made for the in-place approach.



(a) Primitive.

(b) Naive solution.

(c) Efficient solution.

Figure 3.4: The different array types.

- replace type in array casts; and
- replace type in (multi)array initialization, and initialise each $Cell_t$.

Without low-level support, the overhead of the in-place strategy on arrays is inherent, as there is no getting around replacing a primitive array with an array of objects. Regardless, there is a more fundamental issue with this solution: the overhead imposed on non-instrumented (hence also non-transactional) code.

Consider application code invoking Java platform or third-party non-instrumented libraries. For example, the `Arrays.binarySearch(int[], int)` method from the Java platform. The Java core classes, and any other non-instrumented libraries, are oblivious to our transactional arrays, hence we need to recreate an `int[]` from the data in $Cell_{\texttt{int}}$`[]` and pass the `int[]` as argument to the method. The method itself is a black box from our point of view, therefore we do not know if any element in `int[]` was modified.[2] Unless we were to build some kind of black/white list with such information for *all* methods, the only solution is to copy the values from `int[]` back into $Cell_{\texttt{int}}$`[]`. All these memory allocation and copies significantly hamper the performance of non-transactional code, which should not be afflicted due to transactional-related instrumentation.

An efficient solution. The problem with the approach just described arises from the disappearance of the original primitive array, which is *completely* superseded by its transactional counterpart. Therefore, besides having per array cell metadata we also want to keep the original array ready for any non-transactional access, bypassing the need for the costly memory copy operations.

Our improvement is as follows. Let $\mathcal{T}$ be the set of all Java primivite types. For each $t \in \mathcal{T}$, let $Array_t = \{a, a^m\}$ be a class with a field a of type t`[]` (the original array), and a field a^m, an array of metadata objects (Figure 3.4c). Each metadata object in a^m`[`i`]`

[2] In this example we used the `binarySearch` method which does not modify the array, but in general we do not know.

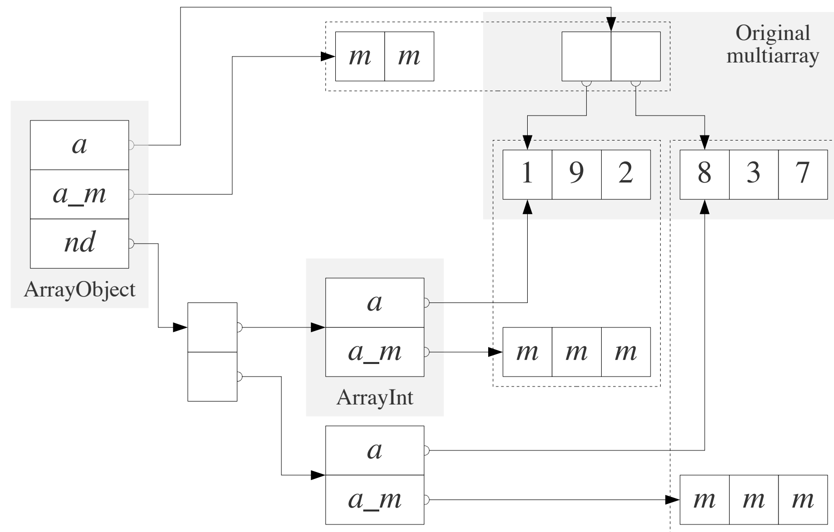


Figure 3.5: A multiarray strucutre in the the efficient solution.

is associated with the corresponding array cell a`[`i`]`. We replace all t`[]` by $Array_t$ and adjust the code accordingly. The bytecode modifications needed are:

- replace type in class field declarations;
- replace type in local variable declarations;
- update signature of methods receiving and/or returning (multi)arrays;
- replace non-transactional accesses to t`[`i`]` with $Array_t.a$`[`i`]`;
- pass the array cell metadata, $Array_t.a^m$`[`i`]`, to the transactional access callbacks;
- replace type in array casts; and
- after a (multi)array initialization, create $Array_t$, store the original (multi)array in $Array_t.a$ and initialise each metadata object in $Array_t.a^m$.

An example of the required transformations can be seen in Figure 3.3. In Lines 15, Figure 3.3b, we can see the creation of the $Array_{\texttt{int}}$ container. Line 16 depicts the $Array_{\texttt{int}}$ container storing a reference to the original array, and Lines 18-21 the initialisation of the metadata associated with each array cell. In both Lines 34 and 38 the return type has been changed from `int[]` to the $Array_{\texttt{int}}$ type.

Multidimensional arrays are also accommodated by this solution. Like previously, let $Array_{multi} = \{a, a^m, nd\}$ be a class with a field a of type `Object`, a field a^m (array of metadata) and a field nd (next dimension) of type $Array_t$`[]`. Field a references the first dimension of the original array and each metadata object in a^m`[`i`]` is associated with the array cell a`[`i`]`. Each nd`[`i`]` is a transacional array ($Array_t$) associated with the original

Table 3.1: Comparison between primitive and transactional arrays.

Arrays	Access n^{th} dimension	Objects	Non-transactional methods
Primitive	$n-1$ *derefs*	$\sum_{i=1}^{n} l^{i-1}$	-
Naive solution	$2n-1$ *derefs*	$\sum_{i=1}^{n} l^{i-1} + l^{i}$	$cr_p + 2(tr_t + l^n \textit{ copies}) + tr_p$
Efficient solution	$2n$ *derefs*	$\sum_{i=1}^{n} 3l^{i-1} + (i-1)$	1 *deref*

$$n \text{ (dimensions)}, \ l \text{ (length)}$$

$$cr_p = \sum_{i=1}^{n} l^{i-1} \textit{ allocs} \text{ (create primitive array)}$$

$$tr_p = \left(\sum_{i=1}^{n} l^{i-1}\right) - 1 \textit{ derefs} \text{ (traverse primitive array)}$$

$$tr_t = \left(\sum_{i=1}^{n} l^{i-1} + l^{i}\right) - 1 \textit{ derefs} \text{ (traverse naive transactional array)}$$

2nd-dimensional sub-array a`[`i`]`, and so forth (Figure 3.5).

Table 3.1 provides a comparison between primitive, the naive and efficient solution for arrays. For accessing any n^{th}-dimensional cell, in a primitive array it takes $n-1$ object dereferences (dereferencing all previous dimension arrays except the 1st). In the naive approach, it takes $2n-1$ dereferences because each cell is now a $Cell_t$, introducing an extra dereference per dimension. The efficient solution takes $2n$, with the extra dereferencing of initial $Array_{\texttt{Object}}$.

We now analyse how many objects does each approach need for an n-dimensional array. For simplicity's sake, let us assume that all arrays have the same length, l. Primitive arrays have $\sum_{i=1}^{n} l^{i-1}$ objects (each dimension's array cell is a reference to another array, except in the last dimension). The naive transactional arrays have the same arrays (l^{i-1}), plus an extra $Cell_t$ in every array cell (l^i). The arrays in the efficient solution again have l^{i-1} arrays (corresponding to the the original array which is kept), and as many $Array_t$ and metadata arrays $Array_t.a^m$, which totals $3l^{i-1}$. There are an additional $i-1$ arrays ($Array_{\texttt{Object}}.nd$).

At last, but not least, we consider the usage of each approach when passed as argument to a non-transactional method. Let us assume that the whole (multi)array is passed, instead of a specific sub-dimensional part. With primitive arrays, all we have to do is pass the array reference as argument. The naive approach requires a colossal amount of work, from creating an equivalent primitive array $\mathcal{A}$ (cr_p) while traversing the naive transactional array (tr_t) and copying each value to $\mathcal{A}$ (l^n). $\mathcal{A}$ is then passed as an argument, and after the method returns we must traverse both $\mathcal{A}$ (tr_p) and the naive transactional array (tr_t) to write back the values into the naive transactional array (l^n). With our proposed solution, we only need to dereference the root $Array_{\texttt{Object}}$ and then pass the reference in

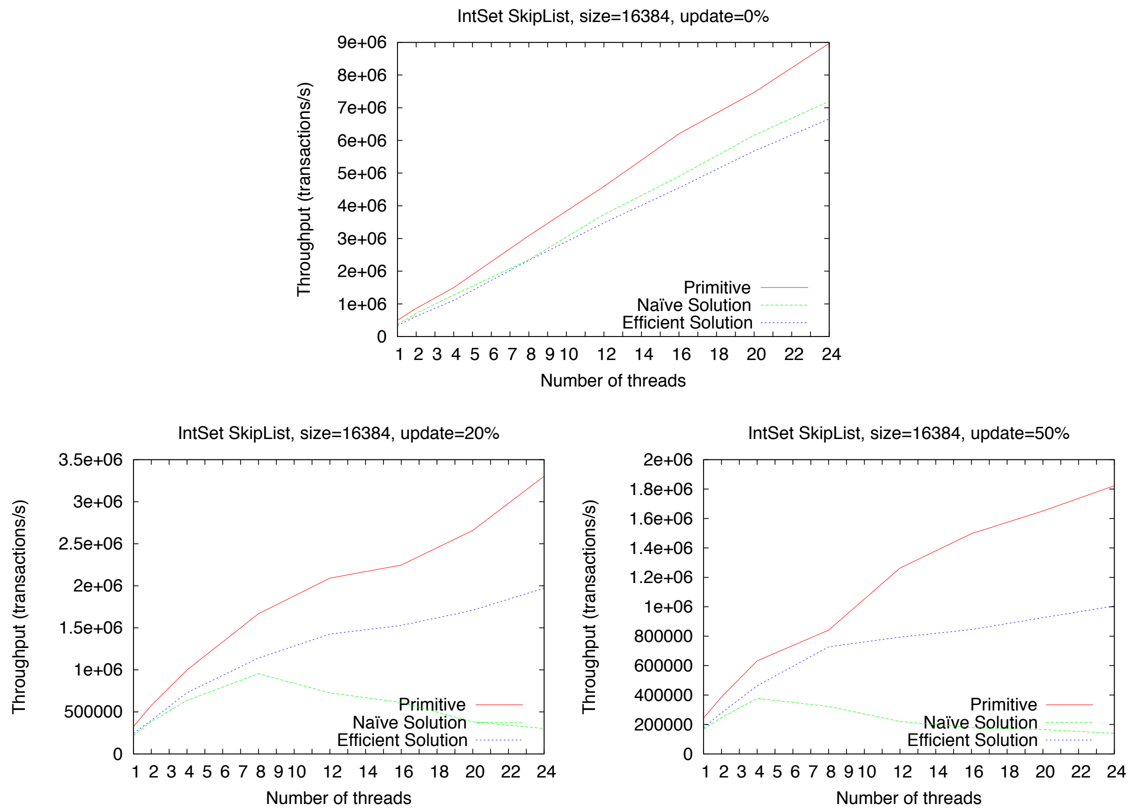


Figure 3.6: Performance comparison between array types.

the a field to the method, greatly improving upon the naive solution.

3.5 Evalution

To evaluate the efficient solution for transactional arrays, we compare the performance of the primitive, naive, and efficient solution, on the Skip List microbenchmark included in Deuce. The various layers of the skip list are implemented with arrays, which makes this benchmark relevant for our evaluation.

The benchmark was executed in a computer with four hexa-core[3] processors. Figure 3.6 shows the performance of the three array types under three different workloads. The top graph depicts the throughput (transactions per second) of the system with no write transactions (0% writes). The left graph shows the throughput with 20% writes, and the rightmost with 50%.

As expected, the external strategy with primitive arrays (Primitive) has the best performance and scalability, independently of the write rate. On the contrary, the naive approach (Naïve Solution) is shown to not scale beyond 8 threads with 20% updates, and 4 threads with 50% updates. This differs from the efficient solution (Efficient Solution) which scales gracefully with the number of threads.

[3] An hexa-core has 6 cores.

We believe that the scalability shown by the efficient solution's arrays as opposed to the naive arrays is due to its cache-friendlier nature. The naive approach completely disrupts the memory layout of the array, unlike the efficient approach which keeps the originally allocated array.

3.6 Summary

We presented TribuSTM, an extension to the Deuce Java STM framework. Deuce supports the implementation of STM algorithms following an external placement strategy for transactional metadata. This approach does not allow the efficient implementation of all kinds of algorithms, namely those which require a 1-1 mapping between metadata and the management memory locations. TribuSTM fills this gap by supporting in-place metadata without any change to the API the framework provides to the applications, by leveraging on bytecode instrumentation. Therefore, TribuSTM enables fair comparison between algorithms which previously add to be implemented in frameworks biased towards either of the metadata placement strategies.

In this dissertation we have contributed with a novel solution to allow in-place metadata in arrays. We began with a naive approach that imposed an unacceptable overhead for non-transactional code and, as shown in § 3.5, does not scale. We then proposed a solution that transparently supports transactional n-dimensional arrays that (1) impose negligible overhead for non-transactional code; and (2) delivers scalable performance.

Currently, the creation or structural modification of transactional arrays is not supported outside instrumented code, which is oblivious to the custom structure of the transactional arrays (and metadata). This issue remains an open direction for future work.

The contribution in this Chapter has appeared in the paper "Efficient Support for In-Place Metadata in Transactional Memory", Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par), 2012 [DVL12], which has been awarded as a *distinguished paper*.

4

Distributed Software Transactional Memory with TribuSTM

This Chapter presents an extension to TribuSTM (Chapter 3) in order to support Distributed Software Transactional Memory. We provide an overview in § 4.1, and proceed to a detailed presentation of our extension in § 4.2. In § 4.4 we describe one implementation of a replicated Software Transactional Memory using our framework. Finally, we conclude with a summary in § 4.5.

4.1 Introduction

In Chapter 2 we present the foundation of our work. We start by introducing the transactional model and its properties (§ 2.1), which led to Software Transactional Memory (STM), described in § 2.2. We cover its semantics in § 2.2.1 and present implementation strategies that can be applied to realise STM (§ 2.2.2). We end by examining how can STM be used by the programmer by introducing it's programming model (§ 2.2.3) and the existing frameworks (§ 2.2.4).

Despite initially being studied in the context of chip-level multiprocessing, STM's benefits over traditional concurrency control methods also make it an attractive model for distributed concurrency control. We introduce new questions that arise from Distributed STM (DSTM), such as how do the participants in the distributed system communicate, where is data and metadata stored and how to access them, and how to validade and commit transactions in a distributed setting (§ 2.3). In § 2.3.1, we overview the state of the art regarding how to commit transactions and maintaining memory consistency in an environment where data is replicated across all nodes of the system. And finally, as

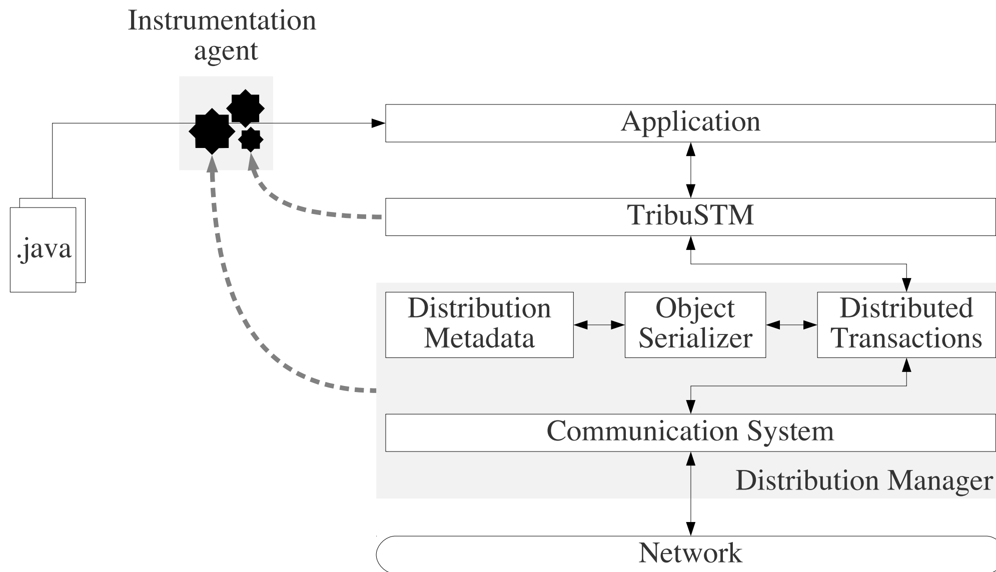


Figure 4.1: Our framework's architecture overview.

before, we end by analysing the existing framework solutions for DSTM (§ 2.3.2).

It is not contrived that one would build a DSTM infrastructure by extending an existing STM framework. In fact, that is what we did. From all the existing frameworks, the one with the most appealing characteristics, in our perspective, is our Deuce extension described in Chapter 3. It avoids any changes or additions to the Java Virtual Machine (JVM) and it does not require language extensions or intrusive APIs. Additionally, the in-place strategy provides a structured hierarchy for transactional metadata and we have provided specialized arrays. Furthermore, *none* of the existing DSTM frameworks feature a non-intrusive API, which is an objective of ours, and are only tailored for either a replicated environment or a fully distributed one.

In order to achieve this infrastructure we have to provide support for distributed objects, means to access them transparently and validating and committing transactions accessing these objects.

4.2 Putting the D in TribuDSTM

Figure 4.1 depicts the architecture of our system. It intends to provide a flexible framework for DSTM allowing different realisations of the two fundamental software layers of the architecture, specifically (1) local concurrency control; and (2) distributed commit and memory consistency.

Local concurrency control is taken care of by TribuSTM. Different STM algorithms have been shown to be better suited to different workload scenarios, and TribuSTM supports distinct algorithms transparently from the application's perspective.

By seeking distributed concurrency control through DSTM, the framework stack is

augmented with a distributed commit and memory consistency layer, the Distributed Manager (DM). It is responsible for establishing the distributed (possibly shared) memory through mechanisms for object distribution, and transaction execution on the distributed memory. Depending on the distributed memory strategy in use, different algorithms may be used to execute distributed transactions.

Inter-layer compatibility is achieved through well defined interfaces. The Application ↔ TribuSTM interface remains unchanged – methods are defined as transactions with the `@Atomic` annotation, and transactional versions of methods redirect read and write accesses to TribuSTM, and so forth.

To support distributed transactions, the DM should be aware of *events* triggered by the Application on TribuSTM, such as transactional accesses or commit requests, in order to take the necessary measures. The DM, as part of its support for distributed transactions and their system-wide commit, might also need to query or alter the state of the local STM, *independently* of the STM algorithm being used. These operations make up the bulk of the TribuSTM ↔ DM interface. Distributed objects are achieved with a combination of *metadata* concerning objects and the Java Serialization support. In a nutshell, to serialize an object consists in transforming its state in a sequence of bytes. The opposite process, deserialization, recreates the object from such sequence. This can be used to send objects through a network from one computer to another.

A quick note before proceeding to a more in-depth description of our system. STM algorithms associate metadata to each object's field to manage concurrency. Our support for distributed objects also associates metadata with each object to implement a desired distributed object model. These are clearly two distinct kinds of metadata, thus we shall refer to both as *transactional* and *distribution* metadata, respectively, to disambiguate if necessary.

4.2.1 Distributed Transactions

In the distributed setting, transaction operations are not necessarily strictly local. Depending on the distributed memory model being employed, a transactional read or write access, or any other operation, might require communication among the system's participants.

Consider, for example, a full replication scenario where transactions are committed using a Certification scheme as presented in § 2.3.1, Figure 2.8a. Replicas execute transactions locally in an optimistic fashion, only requiring synchronisation when a transaction attempts to commit. In this case the DM is not interested in the transactional read and write accesses, but the commit request triggers the TO-Broadcast of the transaction's read and write set to certify the transaction. If in a non-replicated environment instead,

Table 4.1: Operations provided by the Reflexive API.

Operation	Description
ONSTART($\mathcal{T}$)	Notifies of the start of transaction $\mathcal{T}$.
ONREAD($\mathcal{T}$, m)	Notifies of the read on transactional metadata m by transaction $\mathcal{T}$.
ONWRITE($\mathcal{T}$, m, v)	Notifies of the write of value v on transactional metadata m by transaction $\mathcal{T}$.
ONCOMMIT($\mathcal{T}$)	Notifies of the commit request issued from transaction $\mathcal{T}$.
ONABORT($\mathcal{T}$)	Notifies of the abort of transaction $\mathcal{T}$.

transactional read and write operations might trigger a request to the owner of the read-/written memory location, if such location is not local.

Besides actuating on events from TribuSTM, the DM also needs to retrieve information from the state of the STM, or even to modify it. Supposing the same scenario (full replication), in order for the DM to initiate the Certification protocol it must acquire (at least) the read and write set of the transaction, and update the STM state according to the latter if the validation was successful.

The interaction between the DM and TribuSTM is twofold. One aspect deals with the need for the DM to react to certain events from TribuSTM, while the second aspect requires the DM to query and/or alter the state of the local STM. These two distinct interactions between the DM and TribuSTM are accomplished through two different APIs.

In the first interface, the *Reflexive* API (Table 4.1), the DM reacts to events from TribuSTM using the the Observer design pattern [GHJV94]. It subscribes the relevant events according to the distributed memory model and protocol for distributed transactions which it is implementing. Specifically, the DM registers callbacks for transaction-related events such as transaction start (ONSTART), transactional accesses (ONREAD, ONWRITE), and commit and abort (ONCOMMIT, ONABORT).

The second interface, the *Actuator* API (Table 4.2), enables the DM to inspect the state of the local STM and act upon it. The DM can acquire an opaque representation – reminiscent of the Memento pattern [GHJV94] – of a transaction's state (CREATESTATE) which can be used to recreate the transaction (RECREATETX). The opacity of this representation is essential in order to support any STM algorithm implemented in TribuSTM. It is also possible for the DM to explicitly trigger the validation (VALIDATE) and apply the updates (APPLYWS) of a transaction.

When the DM reaction to an event triggers communication with remote nodes, it might need to wait for a response, thus blocking the execution. Execution resumes after the response is received, from which it is notified via callback ({START, READ, WRITE, COMMIT, ABORT}PROCESSED).

Table 4.2: Operations provided by the Actuator API.

Operation	Description
CREATESTATE($\mathcal{T}$) : $\mathcal{S}$	Returns a representation of transaction's $\mathcal{T}$ state, composed by its read set $\mathcal{RS}$, write set $\mathcal{WS}$, local identifier id, in addition to other opaque STM algorithm-dependent relevant information.
RECREATETX($\mathcal{S}$) : $\mathcal{T}$	Returns a transaction $\mathcal{T}$ recreated from state $\mathcal{S}$.
VALIDATE($\mathcal{T}$) : *bool*	Validates transaction $\mathcal{T}$ returning true or false if successful or not, respectively.
APPLYWS($\mathcal{T}$)	Applies all updates by transaction $\mathcal{T}$ on the local STM.
STARTPROCESSED($\mathcal{T}$)	Notifies that the start of transaction $\mathcal{T}$ has been processed.
READPROCESSED($\mathcal{T}$, v, a)	Notifies that the read from transaction $\mathcal{T}$ has been processed, yielding value v. If a is true a conflict as been detected, hence $\mathcal{T}$ must abort.
WRITEPROCESSED($\mathcal{T}$, a)	Notifies that the write from transaction $\mathcal{T}$ has been processed. If a is true a conflict as been detected, hence $\mathcal{T}$ must abort.
ABORTPROCESSED($\mathcal{T}$)	Notifies that the abort request from transaction $\mathcal{T}$ has been processed.
COMMITPROCESSED($\mathcal{T}$, c)	Notifies that the commit request from transaction $\mathcal{T}$ has been processed with outcome c, true or false if successfully committed or not, respectively.

Different implementations of the DM can have specific requirements for the Communication System (CS). For instance, the Certification scheme running on a fully replicated memory requires a Group Communication System (GCS) with support for a TO-Broadcast (Non-Voting and Voting) and R-Broadcast (Voting) primitives. However, if we have a purely distributed environment (every object has one and only one owner) and use a pessimistic approach, *i.e.*, lock objects upon access, we only need point-to-point communication. Therefore, unlike previous approaches to DSTM frameworks, the CS is not a first-class component like TribuSTM or the DM.

4.2.2 Distributed Objects

Application data accessed in the context of transactions can be purely distributed, or fully or partially replicated. In Chapter 3 we point out that, in general, STM algorithms associate algorithm-dependent transactional metadata to the managed memory. Likewise, one can envision that several distributed memory (or objects) strategies, such as fully or partially replicated, or even completely distributed, can also be carried out by combining some kind of distribution metadata with the memory locations.

For instance, in a fully replicated environment, we want to logically identify a memory location across the whole system. Each replica should therefore be able to identify a local memory location as being its representative of a global location. In this case the

Table 4.3: Operations provided by the Distributed Object API.

Operation	Description
GETMETADATA($\mathcal{O}$) : $\mathcal{M}$	Returns the metadata $\mathcal{M}$ associated with distributed object $\mathcal{O}$.
SETMETADATA($\mathcal{O}$, $\mathcal{M}$)	Associates the metadata $\mathcal{M}$ with distributed object $\mathcal{O}$.
REPLACEOBJECT($\mathcal{O}$) : $\mathcal{D}$	Delegates object $\mathcal{D}$ to be serialized instead of distributed object $\mathcal{O}$.
RESOLVEOBJECT($\mathcal{O}$) : $\mathcal{D}$	Delegates distributed object $\mathcal{D}$ to be deserialized instead of object $\mathcal{O}$.

metadata associated with each memory location should include a Unique Identifier (ID). A memory location with the same ID on different replicas represents the same global location. On the other hand, for a purely distributed memory, metadata might be composed of all the necessary information to execute a Remote Procedure Call (RPC) to the owner of the location in order to perform a transactional access.

Our framework's support for distributed objects is twofold. One aspect deals with the association of distribution metadata to objects and means to retrieve it. The other utilises the Java Serialization support to override how objects are sent to other participants in the system depending on the distributed memory model.

Metadata. The metadata itself can vary substantially depending on the distributed memory model employed, so the only requirement is that it implements a `DistMetadata` interface. To associate metadata with objects and retrieve it, one can think of several approaches.

For instance, define an abstract class containing a field of type `DistMetadata` and respective *getter* and *setter*, from which all application classes should inherit. The drawback of this strategy is that Java only supports single inheritance, that is, a class can only inherit from one super class. Thus, imposing a super class is intrusive and might be infeasible if the application's classes hierarchy is complex and using third-party libraries.

Alternatively, instead of forcing a super class one can define a `DistributedObject` interface consisting of only the *getter* and *setter* for the `DistMetadata` object (Table 4.3). This requires the application programmer to explicitly state her classes implement the interface `DistributedObject` and provide the implementation of the *getter* and *setter* repeatedly. To get around this, and in line with our objective of non-intrusiveness, our instrumentation agent automatically injects such code for the application programmer as follows. Let $\mathcal{C}^{\mathcal{I}} = \{\overline{f}, \overline{m}\}$ be class $\mathcal{C}$ implementing the set of interfaces $\mathcal{I}$, and $\overline{f}$ and $\overline{m}$ the set of fields and methods of $\mathcal{C}$, respectively. $\mathcal{C}$ is made to implement `DistributedObject`, and a new field f^{dm} of type `DistMetadata` is added, along with its companion *getter* and *setter*, $getMetadata$ and $setMetadata$ respectively, and their implementation. After all

transformations we have

$$\mathcal{C}^{\mathcal{I}\,\cup\,\texttt{DistributedObject}} = \{\overline{f}, f^{dm}, \overline{m}, getMetadata, setMetadata\}$$

Serialization. When using the serialization support offered by Java, by default an object's state is transformed in a sequence of bytes that can afterwards be transformed back into an identical object. Conveniently there is support for a fine manipulation of this process.

Of particular interest are the methods `writeReplace` and `readResolve` from the Java Serialization API. The first allows an object being serialized to designate *another* object, possibly even of a different class, to be serialized in its place, while the latter allows equivalent behaviour but with respect to deserialization instead. That is, the object designated by `writeReplace` is in fact the one whose state is transformed in a sequence of bytes (instead of the original), and the object designated by `readResolve` is the one whose state is reconstructed from the sequence of bytes and returned. The use of this operations needs to be carefully thought as to not violate the expected classes.

The key idea is that distributed objects are serialized in function of its distribution metadata, so different implementations of the DM might serialize objects differently according to their distributed memory model.

For example, consider the full replication environment where objects are associated with an identifier id – its distribution metadata! – that uniquely identifies the object system-wide. Using a Certification protocol to commit transactions, the transaction's read and write set are sent to all replicas, hence being serialized to be disseminated through the network. Using the default serialization provided by Java, when both sets are deserialized at each replica they will be populated with freshly created objects which *are not* the local representatives with identifier id on the replica. The correct behaviour would be to replace all objects o in the read and write sets with the existing local representatives r such that $o.id = r.id$. This way each replica manipulates the correct local representative of any distributed object. Other schemes might require different (de)serialization algorithms, *e.g.*, in a distributed memory setting, serializing a *proxy* object which redirects accesses to the real object.

To implement such behaviour, distributed objects are injected with the `writeReplace` and `readResolve` methods. The flexibility to (de)serialize them accordingly to the specific DM implementation is achieved by inserting hooks (REPLACEOBJECT, RESOLVEOBJECT, Table 4.3) in `writeReplace` and `readResolve` respectively, delegating the process to the DM.

To summarise, the following instrumentation of the Java bytecode is performed to support distributed objects. Let $\mathcal{C}^{\mathcal{I}} = \{\overline{f}, \overline{m}\}$ be class $\mathcal{C}$ implementing the set of interfaces $\mathcal{I}$, and $\overline{f}$ and $\overline{m}$ the set of fields and methods of $\mathcal{C}$, respectively. $\mathcal{C}$ is made to implement `DistributedObject`, and a new field f^{dm} of type `DistMetadata` is added, along with

the implementation of its companion *getter* and *setter*, $getMetadata$ and $setMetadata$ respectively. To conclude, the methods $writeReplace$ and $readResolve$ are implemented by delegating its functionality to the DM. The final result is

$$\mathcal{C}^{\mathcal{I} \,\cup\, \texttt{DistributedObject}} = \begin{Bmatrix} \overline{f}, f^{dm}, \\ \overline{m}, getMetadata, setMetadata, writeReplace, readResolve \end{Bmatrix}$$

4.3 On the Implementation of a Distributed STM

In this Section we reason on how to implement a distributed STM using the framework presented in § 4.2.

Consider that distributed metadata is composed of two pieces of information, (1) a unique identifier, id; and (2) the address of the node which owns the object associated with the metadata, $owner$. Let $id(\mathcal{O})$ and $owner(\mathcal{O})$ denote the id and $owner$ of a distributed object $\mathcal{O}$, respectively. If a distributed object $\mathcal{O}$ is created at node $\mathcal{N}$, $owner(\mathcal{O})$ is set to $\mathcal{N}$.

Let $host(\mathcal{T})$ denote the node where transaction $\mathcal{T}$ executes, and $id(\mathcal{T})$ the unique identifier of $\mathcal{T}$. Let $proxy_n(id)$ denote the proxy transaction of $\mathcal{T}$ on node n such that $id(\mathcal{T}) = id$, and consider, for simplicity, that every node contains a proxy transaction for every transaction which is currently executing in the system.

When a transaction $\mathcal{T}$ issues a read access on transactional metadata m (ONREAD, Table 4.1), if $owner(m) = host(\mathcal{T})$ the read access is a regular, local, access. Otherwise, $host(\mathcal{T})$ sends message $[id(\mathcal{T}), id(m)]$, henceforth rd, to $owner(m)$, and awaits response. When $owner(m)$ receives rd, it resolves $id(m)$ to m, issues a local read on m on behalf of $proxy_{owner(m)}(id(\mathcal{T}))$, and responds to $host(\mathcal{T})$ with message $[id(\mathcal{T}), v, a]$ where v is the value read and a is true if there has been a conflict. When $host(\mathcal{T})$ delivers the response message, the execution of $\mathcal{T}$ is resumed (READPROCESSED, Table 4.2). A write access behaves similarly.

When $\mathcal{T}$ requests to commit (ONCOMMIT, Table 4.1), $host(\mathcal{T})$ sends a message to all nodes to trigger the validation of $proxy_n(id(\mathcal{T}))$ on each node n (VALIDATE, Table 4.2). All nodes respond with the result of the validation. Once $host(\mathcal{T})$ has collected all validation results, if none of the validations failed, a message is sent to all nodes instructing the commit of each $proxy_n(id(\mathcal{T}))$. If at least one validation was unsuccessful, all nodes are instructed to discard their proxies and $\mathcal{T}$ aborts.

Informally, the key idea is that the read and write set of $\mathcal{T}$ are distributed across nodes, according to the locality of the accessed objects.

This is a simple approach, thus there are various optimisations to consider, *e.g.*, creating transaction proxies on demand only on strictly necessary nodes, caching to reduce network communication.

Table 4.4: Interface provided by the GCS to the DT.

Operation	Description
TOBCAST(m)	TO-Broadcasts message m.
RBCAST(m)	R-Broadcasts message m.
SELF(s) : *bool*	Returns true if sender s is the local replica, false otherwise.

Table 4.5: Interface provided by the DT to the GCS.

Operation	Description
ONTODELIVER(m, s)	Notifies of the delivery of message m which has been TO-Broadcasted by sender s.
ONRDELIVER(m, s)	Notifies of the delivery of message m which has been R-Broadcasted by sender s.

4.4 Implementing a Replicated STM

In the context of this dissertation we implemented, using the proposed framework, a replicated STM which targets an environment where objects are fully replicated among all participants, henceforth *replicas*, of the distributed system. Transactions are committed across the system using the Non-Voting Certification scheme as seen in Figure 2.8a. This replicated STM implementation is a concrete realisation of the DM component of our architecture (Figure 4.1).

Certification schemes rely on a Total Order Broadcast (TOB) primitive to disseminate transactions at commit time. The total ordering of transactions imposed by the TOB ensures 1-Copy Serializability, *i.e.*, that the transaction execution history across the whole set of replicas is equivalent to a serial transaction execution history on a single not replicated STM.

4.4.1 Communication System

Several Group Communication Systems (GCS) exist that provide the TOB primitive. We created a simple API between the Distributed Transactions (DT) component and the (Group) Communication System (Figure 4.1) in order to assess the impact of different providers for the TOB in the system evaluation.

From the GCS side (Table 4.4), it provides the TOB primitive (TOBCAST) and an operation to test whether a message was sent from the local replica (SELF). To be notified of incoming messages, the DT subscribes to the deliveries using the Observer pattern (ONTODELIVER, Table 4.5).

```
1: committed ← false

2: function COMMIT(𝒯)
3:     ONCOMMIT(𝒯)
4:     return committed

5: procedure COMMITPROCESSED(𝒯, c)
6:     committed ← c
7:     𝒯 was processed
```

(a) TribuSTM.

```
8: localTxs ← ∅

9: procedure ONCOMMIT(𝒯)
10:     localTxs ← localTxs[𝒯.id ↦ 𝒯]
11:     𝒮 ← CREATESTATE(𝒯)
12:     TOBCAST([𝒮])
13:     wait until 𝒯 is processed

14: procedure ONTODELIVER([𝒮], s)
15:     if SELF(s) then
16:         𝒯 ← localTxs(𝒮.id)
17:         localTxs ← localTxs \ 𝒯.id
18:     else
19:         𝒯 ← RECREATETX(𝒮)
20:     valid ← VALIDATE(𝒯)
21:     if valid then
22:         APPLYWS(𝒯)
23:     COMMITPROCESSED(𝒯, valid)
```

(b) Distributed Transactions component.

Figure 4.2: Pseudo-code of the Non-Voting Certification algorithm.

4.4.2 Distributed Transactions

Having specified the interface between the DT component and the GCS, we noew sketch both the implementation of the Non-Voting and Voting Certification, in Figures 4.2 and 4.3, with regard to the Reflexive API (Table 4.1), Actuator API (Table 4.2) and the DT ↔ GCS interface (Table 4.4 and 4.5).

Non-Voting Certification. Since Certification-based schemes are optimistic, transactions execute locally until commit time.

At that instant COMMIT is invoked on TribuSTM (Line 2), which delegates to the DT by issuing ONCOMMIT. The DT maintains, for local transactions, a mapping between the transaction's identifier and the transaction itself (Line 8). When a transaction attempts to commit, it is added to the map of local transactions (Line 10). We proceed by obtaining the state of the transaction which is subsequently TO-Broadcasted (Lines 11 and 12). At this point, this thread of execution (the application thread) waits for the transaction processing to finish.

Upon the delivery of the TO-broadcasted message, the ONTODELIVER callback is invoked by a thread from the GCS (Line 14). Depending on whether the received transaction state is local or remote, the DT either obtains the local transaction (Line 16) or recreates the remote transaction (Line 19). From this point they can be managed indistinguishably. We validate the transaction (Line 20) and if no conflicts are detected, apply the transaction's updates (Line 22). The transaction is now processed, thus we generate the appropriate notification (Line 23). On the replica where the transaction is local, this will signal the application thread (Lines 7 and 13) which returns the result of the commit request (Line 4).

```
1: localTxs ← ∅
2: pendingTxs ← nil: queue of ⟨s, S, r⟩
3: pendingResults ← ∅: set of ⟨s, id, r⟩

4: procedure ONCOMMIT(T)
5:     localTxs ← localTxs[T.id ↦ T]
6:     S ← CREATESTATE(T)
7:     S.readSet = ∅
8:     TOBCAST([S])
9:     wait until T is processed

10: procedure ONTODELIVER([S], s)
11:     tx ← ⟨s, S,WAIT⟩
12:     if ∃res ∈ pendingRes : res.s = s ∧ res.id = S.id then
13:         tx.r ← res.r
14:         pendingRes ← pendingRes \ res
15:     pendingTxs.queue(tx)
16:     PROCESSTX

17: procedure ONRDELIVER([⟨id, r⟩], s)
18:     if ∃tx ∈ pendingTxs : tx.s = s ∧ tx.S.id = id then
19:         tx.r ← r
20:     else
21:         pendingRes ← pendingRes ∪ ⟨s, id, r⟩
22:     PROCESSTX

23: procedure PROCESSTX
24:     if pendingTxs is empty then
25:         return
26:     tx ← pendingTxs.peek()
27:     if tx.r = COMMIT ∨ ABORT then
28:         pendingTxs.dequeue()
29:         if SELF(tx.s) then
30:             T ← localTxs(tx.S.id)
31:             localTxs ← localTxs\tx.S.id
32:         else
33:             T ← RECREATETX(tx.S)
34:         if tx.r = COMMIT then
35:             APPLYWS(T)
36:             result ← true
37:         else
38:             result ← false
39:         COMMITPROCESSED(T, result)
40:         PROCESSTX
41:     else
42:         if SELF(tx.s) ∧ tx.r=WAIT then
43:             T ← localTxs(tx.S.id)
44:             if VALIDATE(T) then
45:                 valid ← COMMIT
46:             else
47:                 valid ← ABORT
48:             RBCAST([⟨tx.S.id, valid⟩])
```

Figure 4.3: Pseudo-code of the Voting Certification algorithm.

Voting Certification. Voting Certification explores the trade off of broadcasting smaller messages at the expense of requiring two communication rounds to commit a transaction. Transactions to be processed are identified by the tuple $\langle s, \mathcal{S}, r\rangle$, where s is the replica where the transaction executed (the only replica which can validate the transaction), $\mathcal{S}$ is the transaction's state and r its validation result. Messages containing the validation result of transactions are identified by the tuple $\langle s, id, r\rangle$, where s is analogous, id is the transaction's identifier and r is either COMMIT or ABORT if the validation was successfull or not, respectively.

When a transaction attempts to commit, the procedure is similar to the Non-Voting protocol except that the read set is not broadcasted (Line 7, Figure 4.3).

Upon the delivery of the TO-broadcasted transaction (ONTODELIVER), we check if we have already received the transaction's validation result (Lines 12-14). Since messages R-broadcasted, such as the transaction validation result, need not be totally ordered, a message m R-broadcasted by replica $\mathcal{R}$ might be delivered earlier than a message that $\mathcal{R}$ TO-broadcasted before m. If the validation result has already been received the received transaction's tuple is updated accordingly (Line 13). The tuple is then enqueued for processing, and we process any pending transaction (Lines 15 and 16, respectively). The order in which transactions arrive, and thus are enqueued, defines their serialization order.

Upon the delivery of the validation message (Line 17), we check if we have already received the corresponding transaction, and if so update its tuple with the result of the validation (Lines 18 and 19). If the corresponding transaction has not yet been received, we buffer the validation message (Line 21). We then process any pending transactions (Line 22).

Processing pending transactions works as follows. A transaction is processed if (1) its validation result has already been delivered (Line 27); or (2) it is a local transaction waiting to be validated (Line 42).

We peek the next transaction to be processed (Line 26) and if in the presence of case (1), we dequeue the transaction, recreate the transaction if it is remote (Line 33), and commit if validation was successful (Line 35). The behaviour is identical to the Non-Voting scheme. In this case we keep processing transactions in the queue, for they might also be ready (Line 40).

In case (2), the transaction to be processed is local and is waiting to be validated. We validate the transaction (Line 44) and R-broadcast the result of its validation on Line 48.

4.4.3 Distributed Objects

When the transaction state is TO-Broadcasted by the Non-Voting protocol (Figure 4.2, Line 12), its contents are serialized. The full replication serialization strategy devised guarantees that each replica manipulates its own representatives of the replicated objects.

To identify objects across the system, the distributed metadata assigned to each one is a Universally Unique IDentifier (UUID [uui12]). This UUID is assigned lazily as needed, meaning that objects that never cross the boundaries of a single replica are unaffected.

The serialization algorithm can be seen in Figure 4.4 and works as follows. When an object is being serialized, it invokes OBJECTREPLACE (Line 2). As metadata is assigned lazily on demand, there are two possibilities: either the object already has already been assigned a UUID (Line 4) or not (Line 6). An object with an associated UUID is said to be *published*, or *private* otherwise.

If the object is already published, its UUID oid is nominated to be serialized in its place (Line 5). Type consistency is maintained because the deserialization of oid when delivered on each replica will return the local representative on the replica, *i.e.*, the object o such that GETMETADATA(o) $= oid$ (Line 20)[1]. Clearly, if every replica already possesses a local representative of the object being serialized it would be a waste of resources to disseminate the whole object graph.

If the object is private, a fresh UUID is generated (Line 7) and assigned to the object (Lines 8 and 9). In this case the original object itself is serialized. When it is delivered at each replica, the object is marked as published on the replica (Line 17) and deserialized.

The behaviour on the replica from which the object originated is slightly different, as the object has already been marked as published (Line 9). Therefore we deserialize the

[1]Recall that the `readResolve` method designates a delegate to be deserialized instead of the original object.

```
1: memory ← ∅

2: function OBJECTREPLACE(𝒪)
3:     oid ← GETMETADATA(𝒪)
4:     if ∃ oid then
5:         return oid
6:     else
7:         oid ← generate fresh UUID
8:         SETMETADATA(𝒪, oid)
9:         memory ← memory[oid ↦ 𝒪]
10:        return 𝒪

11: function OBJECTRESOLVE(𝒪)
12:     obj ← memory(oid)
13:     if ∃ obj then
14:         return obj
15:     else
16:         oid ← GETMETADATA(𝒪)
17:         memory ← memory[oid ↦ 𝒪]
18:         return 𝒪
```

(a) Object Serializer component.

```
19: function READRESOLVE(oid)
20:     return memory(oid)
```

(b) Distributed Metadata component.

Figure 4.4: Pseudo-code of the replicated objects' serialization algorithm.

existing object instead (Line 14).

After a thorough analysis of this algorithm one might question the implications of using the map in Line 1. By keeping references to both the distributed metadata and the objects themselves, it prevents them from being garbage-collected when unreachable by the application, thus incurring in memory leaks.

The solution we implemented is twofold. First, we used the `java.util.WeakHashMap` implementation, in which keys (and not values) are held by *weak references*. Informally, a weak reference is a reference which does *not* prevent their referents from being garbage-collected.

This still does not solve the problem because (1) the value objects themselves strongly reference the distributed metadata (which are the map keys), therefore preventing them from being garbage-collected; and (2) the values themselves are strongly referenced, preventing them from being garbage-collected. This is solved by wrapping the value objects in weak references before being put in the map. This way when an object is unreachable by the application it can be garbage-collected from the map, which in turn makes the associated key also unreachable, thus also being garbage-collected.

4.4.4 Bootstrapping

In the previous Section we described the implementation of a fully replicated STM which uses a Certification scheme to commit transactions, using the clearly defined APIs of our framework.

```
1  class Benchmark {
2    @Bootstrap(id = 1)
3    private static RBTree tree;
4
5    @Atomic
6    public void createTree() {
7      if (tree == null)
8        tree = new RBTree();
9    }
10
11   public static void main(String[] args) {
12     ...
13     createTree();
14     // from this point on, tree references the same object in every replica
15     ...
16   }
17 }
```

Figure 4.5: Bootstrapping with the `@Bootstrap` annotation.

The execution model in this environment can either be (1) the same program executing in all replicas; or (2) possibly different programs executing in each replica, but manipulating (some of) the same objects. We have seen in § 4.4.3 that object identifiers are dynamically generated and assigned on demand. If every object is assigned a unique identifier, the managed objects are disjoint. We are in the presence of a classic bootstrapping problem. For example, consider a widely used microbenchmark in the literature, the Red-Black Tree. When the microbenchmark is executed in each replica, it must handle the *same* tree in every replica. Technically, this translates to the tree reference metadata having the *same identifier* in all replicas.

We address this issue using the `@Bootstrap` annotation. It can be used to denote that a number of fields are semantically the same, *i.e.*, are replicas of each other, hence their value is *always* the same. In the example of the Red-Black Tree microbenchmark, the field which references the tree is a use case that fits naturally – we clearly want all replicas to be handling the same tree.

The annotation is used as follows. Its target are fields, and it takes a parameter `id` whose value serves as a *seed* in the identifier generation. Using equal seeds deterministically yields the same identifier. Annotating a field overrides the default process of identifier generation and object publication. Instead of dynamically generating an identifier and locally publishing the field's transactional metadata only at serialization time, it assigns the deterministic identifier to the metadata and locally publishes it immediately.

This gives the practical effect of that field's transactional metadata being already replicated *a priori* with the same identifier in all replicas, thus keeping the field's value consistent across the system when updated (in a transactional context, of course).

Figure 4.5 sketches an example using the annotation. The `tree` field is semantically the same in all replicas and it is initialised in the `createTree` transaction.

Alternatively, if “same program executing on all replicas” is the only execution model considered, we can circumvent the need for the programmer to provide an `id` argument by generating a seed based on the qualified name of the class and the field’s line number.

Consider that the `Benchmark` class in Figure 4.5 belongs to the `x.y` package. Since `tree` field is in Line 3, the implicit seed would be `x.y.Benchmark3`. A seed generated using this strategy is guaranteed to be unique for any field in any class, but all nodes generate seeds accordingly, under the assumption that all nodes have the same “version” of all classes.

4.5 Summary

In this Chapter we described how we augmented the software from Chapter 3 to evolve from a centralized system to a distributed one. In § 4.2 we presented our framework’s architecture, its layers and the clearly defined API between them, and how we extend the local STM layer to support distributed transactions. We also show how we achieve the flexible support for distributed objects in a completely transparent way to the application.

In § 4.3 we reasoned on how to implement a distributed STM using the APIs of our framework.

We also provided, in § 4.4, a description of the implementation, in our infrastructure, of a fully replicated STM using a Certification protocol to maintain consistency when committing transactions.

This Chapter’s contribution was featured in the paper “Uma Infraestrutura para Suporte de Memória Transacional Distribuída”, Proceedings of the Simpósio de Informática (INForum), 2012 [VDL12].

5

Evaluation

This Chapter reports the results of an experimental study aimed at evaluating the performance of a real replicated STM system implemented in the TribuDSTM framework (as described in § 4.4), in face of a variety of workloads with different characteristics.

We begin with some considerations with respect to the modularity of our framework, and the easy, quick, and clean implementation of the Certification protocols in § 5.1. Next, we present the experimental settings in § 5.2. Then, § 5.3 describes the results obtained with a common microbenchmark, the Red-Black Tree. Several STAMP benchmarks and their results are presented in § 5.4.

5.1 Implementation Considerations

We would like to note that implementing both Certification-based protocols was a fairly easy task given the rich APIs we have defined between TribuSTM and the Distribution Manager (DM) layer. In reality, the actual implementations closely resemble the presented pseudo-code in Figures 4.2 and 4.3. The Non-Voting protocol is implemented in under 100 lines of code (LOC) and the Voting protocol less than 200 LOC. These can indistinguishably use the three different Group Communication Systems (GCS) employed in the the following experiments.

The local STM algorithm, the Certification protocol and the GCS implementations, can all be replaced and combined as desired without any burden, by simple parameterization at execution time. The ease of replacing different components of our architecture highlights its modularity.

5.2 Experimental Settings

All the experiments presented in this Chapter were performed in a cluster of 8 nodes, each one equipped with a Quad-Core AMD Opteron 2376 at 2.3Ghz, 4×512KB cache L2, and 8GB of RAM. The operating system was Debian 5.0.8, with the Linux 2.6.26-2-amd64 kernel, and the nodes were interconnected via a private gigabit ethernet. The max send buffer was set to 640KB and the max receive buffer to 25MB as needed by the JGroups configuration. The installed Java Platform was version 6, specifically OpenJDK Runtime Environment (IcedTea6 1.8.3, package 6b18-1.8.3-2lenny1).

In our infrastructure, the local STM layer uses the TL2 algorithm [DSS06]. The DM implementation is the Non-Voting Certification described in § 4.4, in which the consistency of the replicated objects when committing transactions is maintained with the Non-Voting Certification scheme. Read-only transactions are processed *strictly locally* on the replica they executed. With regard to the underlying Group Communication System (GCS) providing the Total Order Broadcast (TOB) primitive needed by the Non-Voting Certification protocol, three different implementations were considered: Appia [app12], JGroups [jgr12] and Spread [spr12]. Switching between GCS is done by parameterization when executing the target program, hence no code rewriting whatsoever is needed.

Appia has been used as the GCS in related work [CRCR09, CRR10, CRR11a, CRR11b]. Its configuration was borrowed from freely available[1] GenRSTM source code, and provides uniform total order through the `SequencerUniform` protocol.

JGroups is a well-known toolkit used in several projects, *e.g.*, JBoss [jbo12]. It was configured according to the UDP configuration from the freely available repository[2], in addition to the `SEQUENCER` protocol which provides non-uniform total order.

Spread differs from the previous in its client-server architecture and being implemented in C, instead of Java. A Spread daemon was deployed on each node. All daemons belong to the same segment, used the *vanilla* configuration, and our framework used the Java API provided by Spread. Message type was set to `SAFE_MESS`, which guarantees uniform total order.

The results are obtained from five runs of each experiment configuration, dropping the highest and lowest result, and averaging the remaining three.

An instance of the replicated STM was deployed on each node. Therefore, in this Chapter node and replica are synonyms.

5.2.1 On the Implementation of Total Order Broadcast

The three chosen GCS rely on different implementations of the TOB primitive.

[1] http://code.google.com/p/genrstm
[2] https://github.com/belaban/JGroups

Appia provides uniform total order using a fixed sequencer [GLPQ10]. Informally, in the fixed sequencer algorithm a single node is assigned the role of sequencer and is responsible for the ordering of messages. Any node n wanting to broadcast message m first unicasts m to the sequencer, which then broadcasts m on behalf of n. Intuitively, this algorithm should not be fair with regard to message ordering. The sequencer should have the upper hand, as it does not need to unicast its messages to itself.

JGroups provides non-uniform total order also using a fixed sequencer. The relaxation of the uniformity properity is likely to allow higher throughput, and even more unfairness, because it does not require all nodes to send back acknowledgements to the sequencer [GLPQ10]. This contrasts with Appia which has the uniform property, requiring the acknowledgements and thus the sequencer more quickly becomes a bottleneck [GLPQ10].

Spread, yet again unlike the others, implements TOB with a privilege-based protocol [GLPQ10]. In a nutshell, privilege-based protocols rely on the idea that senders can broadcast messages only when they are granted the privilege to do so. The privilege to broadcast (and order) messages is granted to only one node at a time, but this privilege circulates from node to node in the form of a token [GLPQ10]. This approach is most likely to achieve fairness, at the expense of throughput, as each node is allowed equal opportunities to disseminate its messages.

5.3 Red-Black Tree Microbenchmark

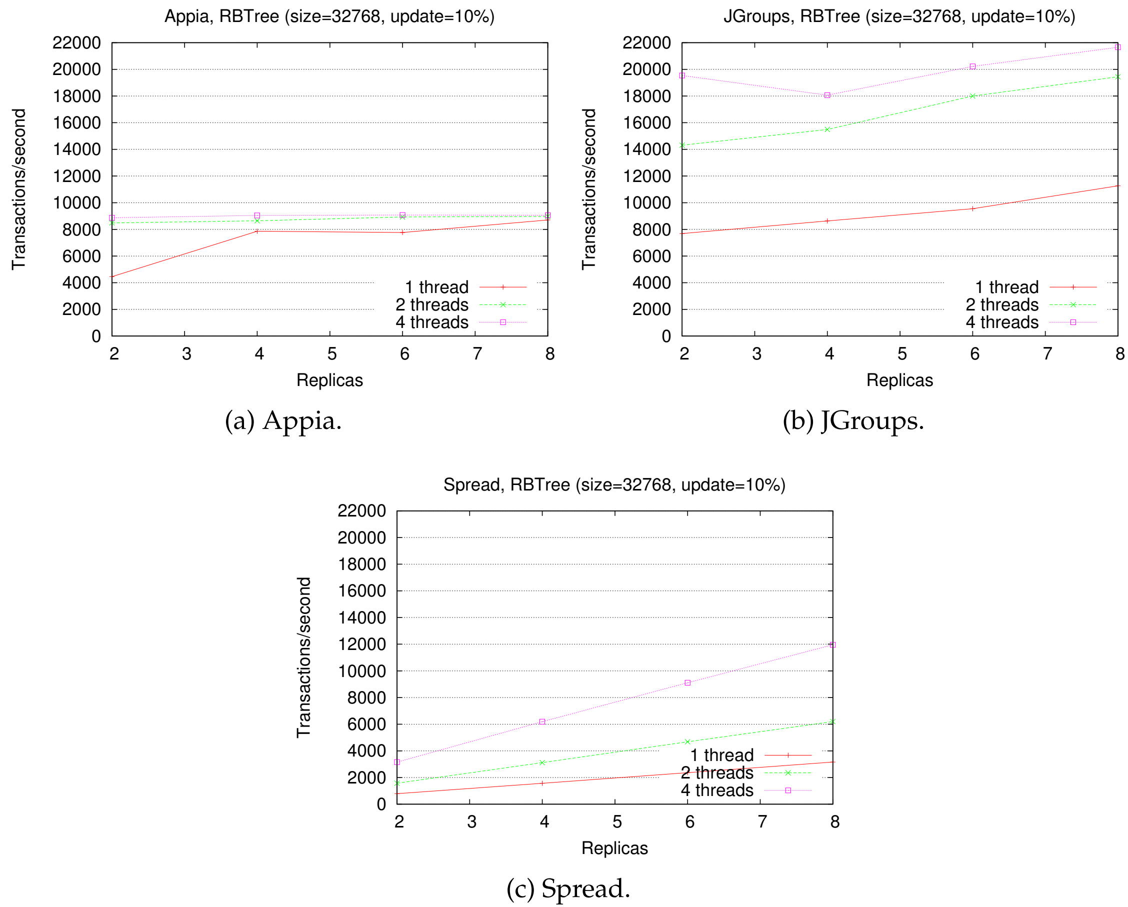
Table 5.1: Parameterization of the Red-Black Tree microbenchmark.

Initial size (-i)	**Value range (-r)**	**Write transactions % (-w)**
32 768 (2^{15})	131 072 (4× initial size)	10

To evaluate our framework, we start by considering a common microbenchmark in the literature, the Red-Black Tree. It is composed of three type of transactions (1) insertions, which add an element to the tree (if not already present); (2) deletions, which remove an element from the tree (if present); and (3) searches, which search the tree for a specified element. Insertions and deletions are said to be *write* transactions.

The microbenchmark was parametrized according to Table 5.1. The tree was populated with 32 768 pseudo-randomly generated values, ranging from 0 to 131 072, thus having an height of 15. Each thread executed 10% of write transactions.

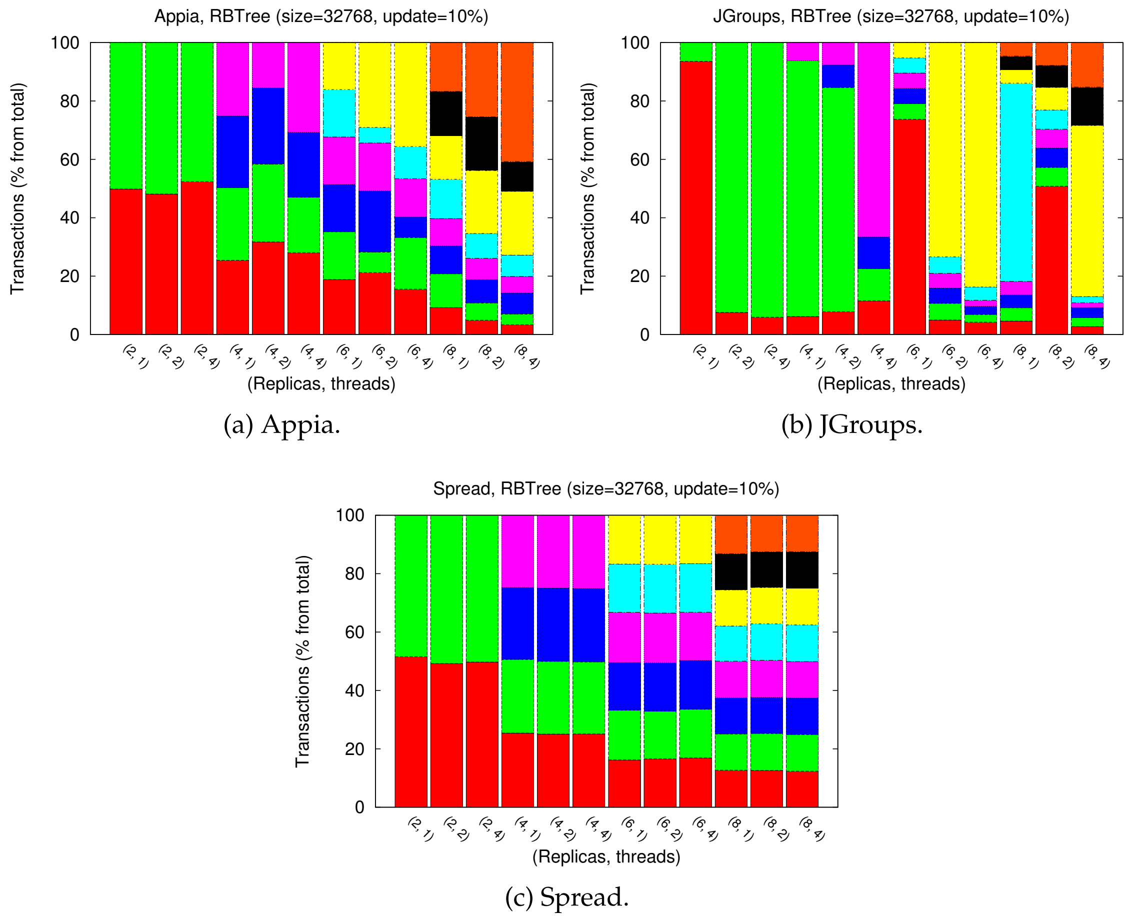
The workload is characterised by very small, fast, transactions and contention is very low.



(a) Appia. (b) JGroups.

(c) Spread.

Figure 5.1: Throughput on the Red-Black Tree benchmark.



(a) Appia.

(b) JGroups.

(c) Spread.

Figure 5.2: Breakdown on the Red-Black Tree benchmark.

Figure 5.1 shows the throughput of our system (higher is better) on the microbenchmark, varying the number of replicas and the number of threads per replica. Unsurprisingly, JGroups (Figure 5.1b) achieves the best performance of the three for every combination, due to both the fixed sequencer implementation and the uniformity relaxation.

Appia (Figure 5.1a) quickly peaks at around half the throughput of JGroups, which can be explained by the requirements of the uniform property.

Finally, in Figure 5.1c, we have the throughput of the system using Spread. The linear scalability displayed is consistent with the idea that the algorithm implemented by Spread achieves fairness. Since every node is provided with equal opportunities to broadcast and order messages, the system scales either with more threads per node, or when nodes increase, not incurring in the bottleneck of a fixed sequencer.

To assess the accuracy of our intuitions, in Figure 5.2 we breakdown the contribution of each replica in the overall throughput. Each color represents one replica. As expected, Spread (Figure 5.2c) achieves fairness as each replica contributes equally to the system's total throughput. This has the practical effect of each replica waiting, on average, the same time for their transactions to be delivered for certification.

In JGroups (Figure 5.2b) we can observe the exact opposite. There is always one replica which dominates the system (most likely the sequencer). In practice, this translates to the sequencer having its messages consistently ordered before everyone else's, therefore waiting a negligible amount of time for their delivery.

Appia (Figure 5.2a) exhibits a mixed behaviour. When the intra-replica concurrency is low, it behaves approximately fair. But as the number of threads grown, the sequencer starts to dominate.

The behaviour of each GCS remains consistent in the following experiments. Spread is fair, JGroups is completely unfair, and Appia is moderately fair if intra-replica concurrency is low, but the sequencer dominates as more threads are added to each replica.

5.4 STAMP Benchmarks

In this Section we evaluate our system with a more complex set of benchmarks from the STAMP suite [MCKO08]. The following experiments allow us to test the system with workloads substantially different from the previous microbenchmark.

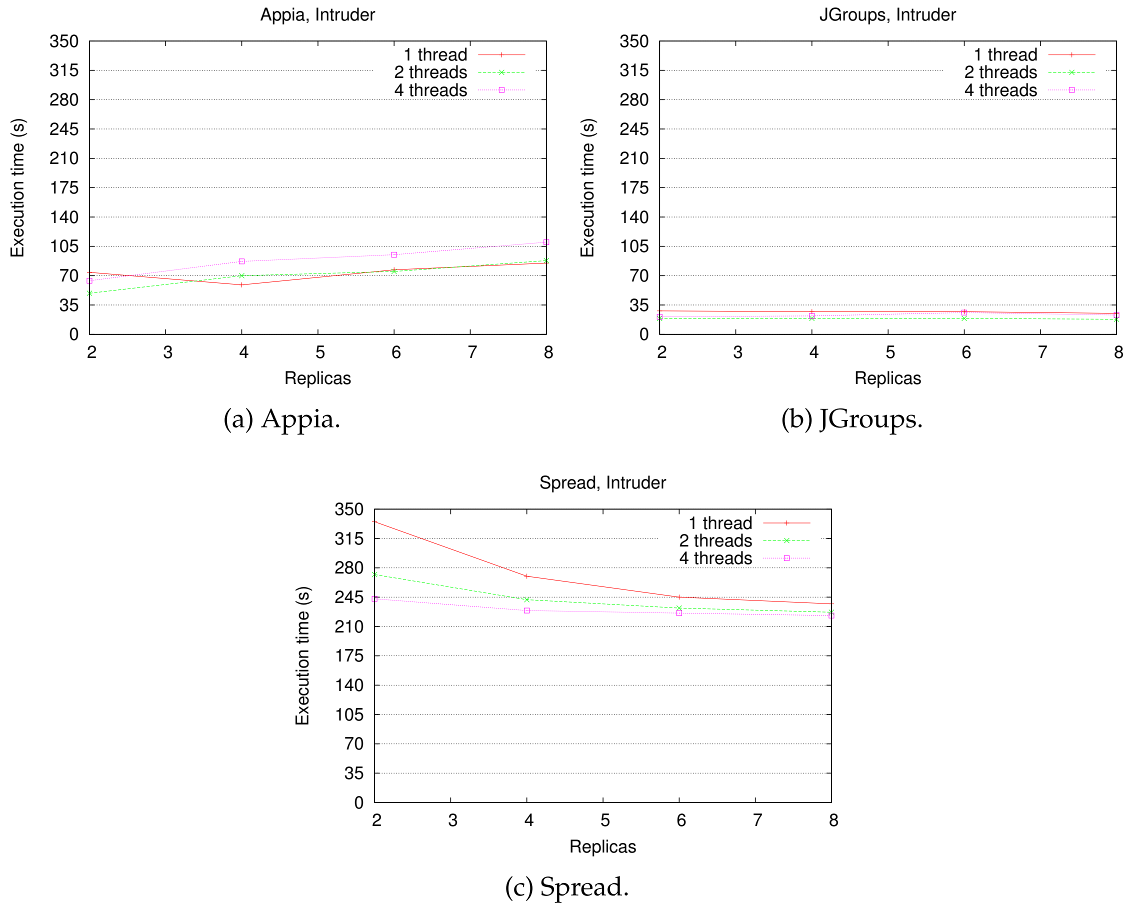
5.4.1 Intruder

The Intruder benchmark simulates the detection of network intrusions.

Each thread repeatedly executes 3 phases. The first phase basically involves a simple FIFO queue from which threads pop a packet. In the second phase threads add the packet to a dictionary (implemented by a self-balancing tree) that contains lists of packets that

Table 5.2: Parameterization of the Intruder benchmark.

Attacks (-a)	Packets (-l)	Flows (-n)	Seed (-s)
10	4	2 048	1



(a) Appia. (b) JGroups.

(c) Spread.

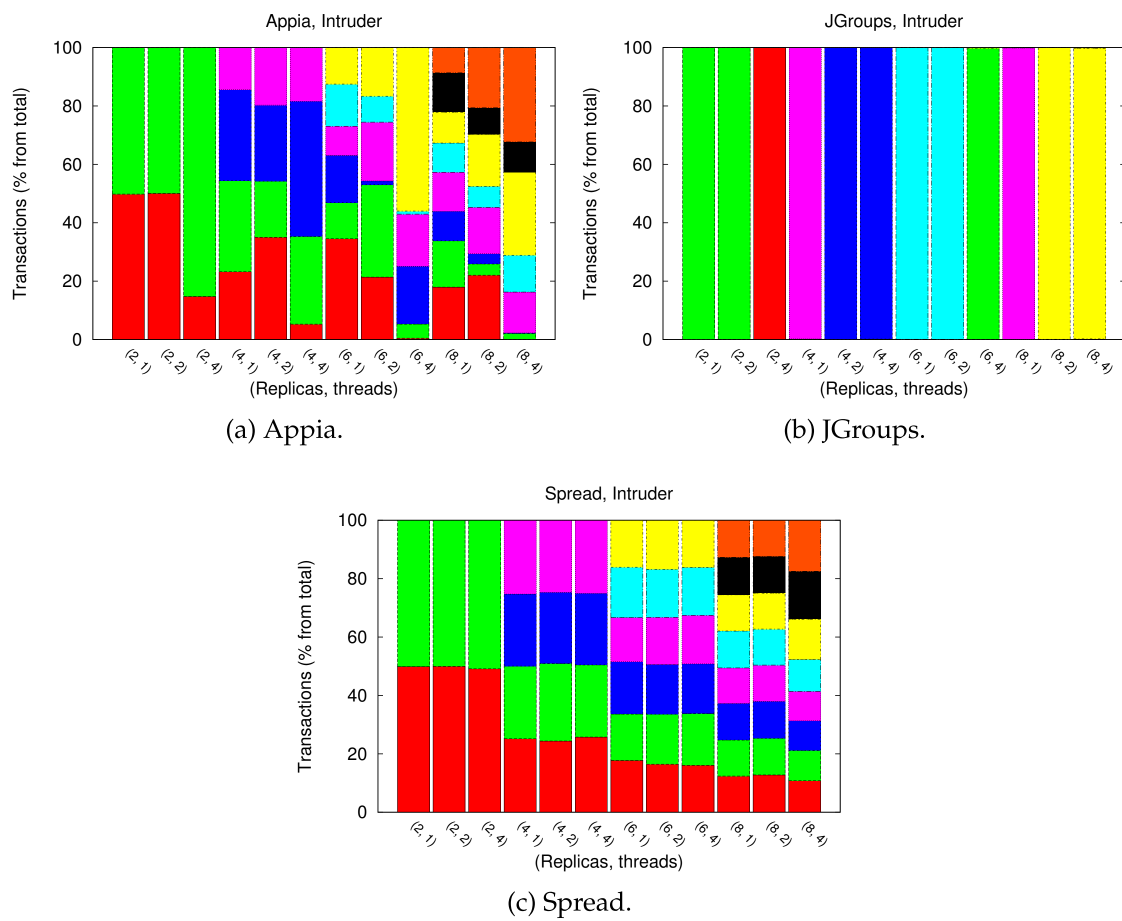
Figure 5.3: Execution time on the Intruder benchmark.

belong to the same flow. If all the packets of a flow have been delivered, they are reassembled and added to the completed packets FIFO queue. The final phase consists of taking a reassembled packet from the FIFO queue and checking if it has been compromised.

The benchmark was parameterized according to Table 5.2. There were 2 048 flows with 4 packets each, and 10 of the flows had been attacked.

Transactions under this configuration are small and fast, and the workload is highly contented, due to both of the FIFO queues and the rebalancing of the tree in the reassembly phase. Thus, this workload distinguishes itself from the the Red-Black Tree's in the contention level.

In Figure 5.3 we have the execution time of the benchmark (lower is better) using each GCS, and varying the number of threads and replicas. As a highly contended workload,

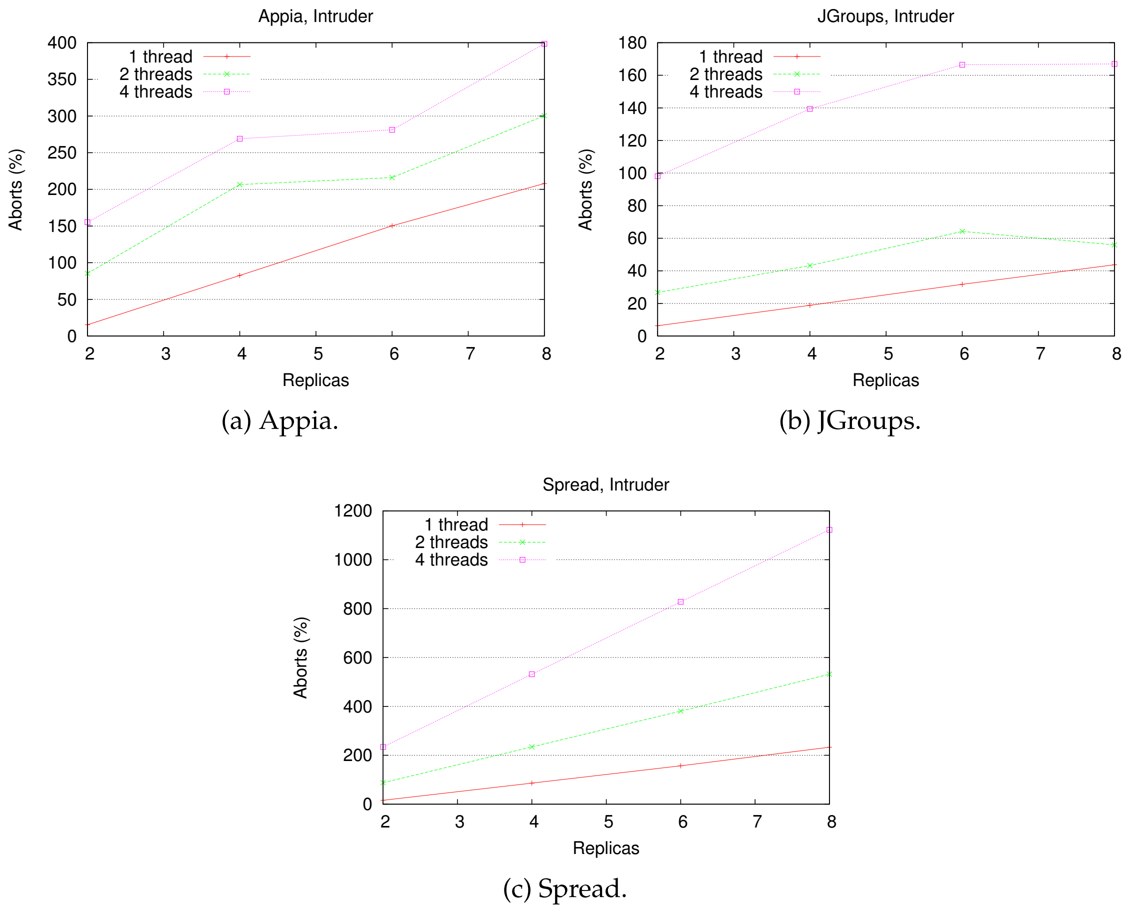


(a) Appia.

(b) JGroups.

(c) Spread.

Figure 5.4: Execution breakdown on the Intruder benchmark.



(a) Appia.

(b) JGroups.

(c) Spread.

Figure 5.5: Abort rate on the Intruder benchmark.

Table 5.3: Parameterization of the Genome benchmark.

Gene length (-g)	Segment length (-s)	Total segments (-n)
256	16	16 384

it is expected that the system performs poorly. Using Appia (Figure 5.3a) the system performance degrades as the total number of threads rises, which proportionally increases the abort rate (Figure 5.5a).

However, JGroups (Figure 5.3b) exhibits constant execution times independently of the replica-thread combinations. This suggests that the sequencer single-handedly executes all the transactions of the benchmark, which is confirmed by the transaction breakdown in Figure 5.4b. The lower abort rate of JGroups (Figure 5.5b) when compared to Appia's (Figure 5.5a) and Spread's (Figure 5.5c) strengthens that, not only did the sequencer execute all committed transactions, these transactions were consistently ordered before any other transactions from the other replicas which experienced incredible latency.

Finally, when using Spread (Figure 5.3c) the benchmark takes around $7\times$ more time to complete than JGroups. This is not unexpected, given the highly contended workload and the fact that fairness is achieved at the expense of throughput. Still, the system shows light scalability for less than 8 threads (in total).

5.4.2 Genome

The Genome benchmark performs gene sequencing, from the bioinformatics domain. Genome assembly is the process of taking a large number of DNA segments and matching them to reconstruct the original source genome.

The program consists of several steps which are executed sequentially, but inside each step several threads execute concurrently. But since the steps are sequential, threads wait for each other when advancing from one step to the next. The last step is completely sequential (it is executed by a single thread), and there is one step which is a mix of concurrent and sequential parts.

The benchmark was parameterized according to Table 5.3. This workload is radically different from both the Red-Black Tree's and Intruder's. Overall, transactions are of moderate length (with regard to the number of operations) and there is little contention. Unlike the previous benchmarks, in Genome data is partitioned among threads. Threads execute a sequence of steps in synchrony, *i.e.*, threads must wait for each other when advancing from step a to step b. Thus, the benchmark exploits intra-step concurrency, but it is ultimately bounded by the synchronization when advancing from step to step.

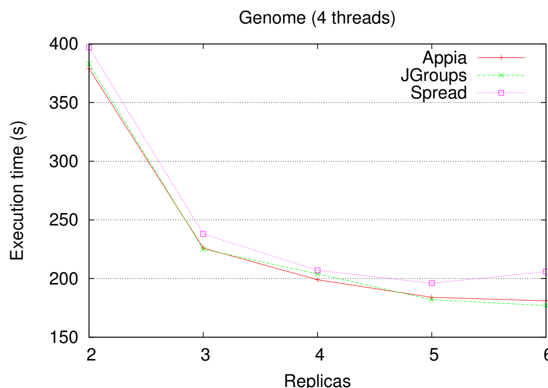


Figure 5.6: Execution time on the Genome benchmark.

Table 5.4: Parameterization of the Vacation benchmark.

Operated items (-n)	Accessible records (-q)	User transactions (-u)	Records (-r)	Sessions (-t)
2	90	98	16 384	4 096

With such workload, specially due to the synchronization between steps, it is expected that the differences between the GCS become negligible. In fact, that is the observed behaviour in Figure 5.6, which depicts the execution time of the Genome benchmark using each GCS and varying the number of replicas. As expected in a low contention environment, the system shows scalability.

5.4.3 Vacation

The Vacation benchmark emulates a travel reservation system.

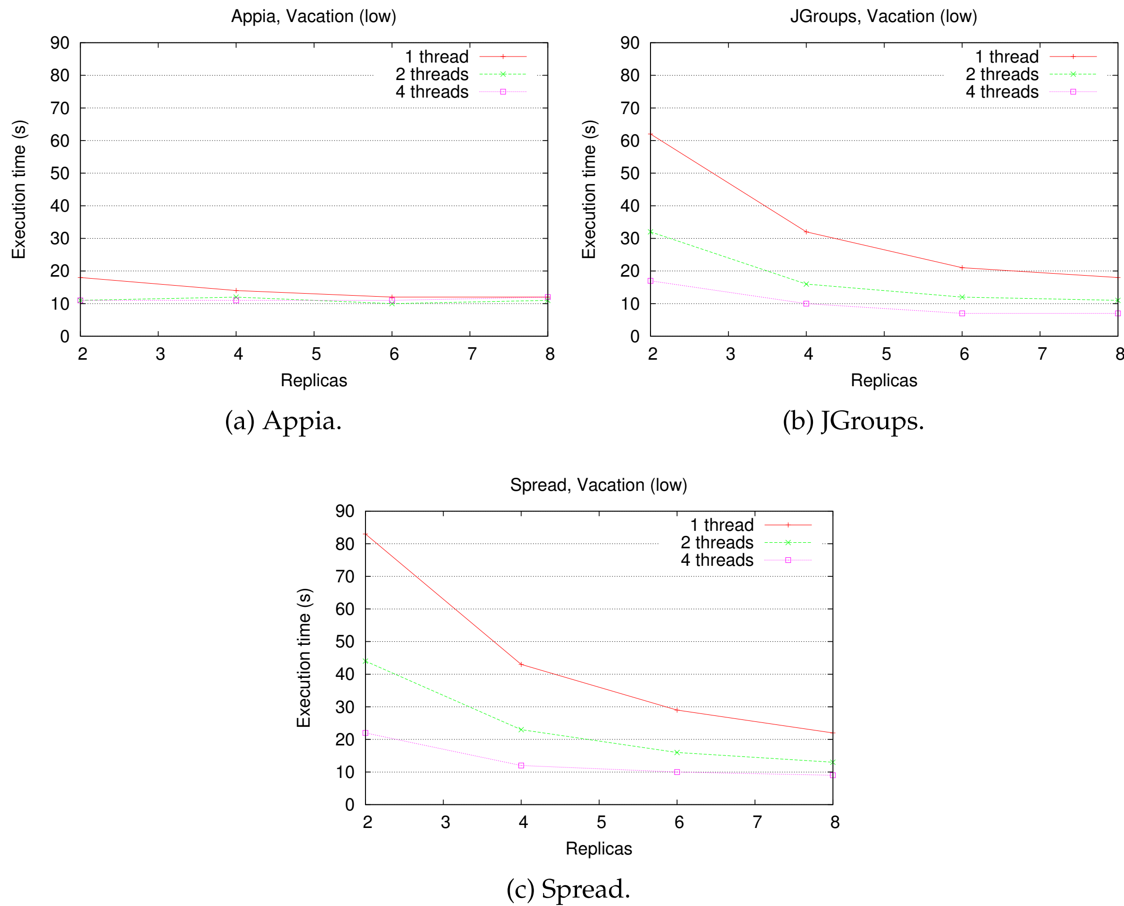
This application implements an online transaction processing system. The system is implemented as a set of trees that keep track of customers and their reservations for various travel items. During the execution of the workload, several client threads perform a number of sessions that interact with the travel system's database. In particular, there are three distinct types of sessions: reservations, cancellations, and updates [MCKO08].

Each of these client sessions is enclosed in a coarse-grain transaction to ensure validity of the database. Consequently, transactions are of moderate size.

The benchmark was parameterized according to Table 5.4, which is the low contention configuration presented in [MCKO08]. The database had 16 384 records of each reservation item, and clients performed 4 096 sessions. Of these sessions, 98% reserved or cancelled items and the remainder created or destroyed items. Sessions operated on up to 2 items and were performed on 90% of the total records.

This workload is similar to Genome's considering that each thread has its own work to perform. Thus, the complete bias of JGroups towards the sequencer should not yield great performance comparing to Spread, since the sequencer can not "steal" the work from the remaining replicas. But unlike Genome, the whole thread execution path is

concurrent – there are no sequential sections or *rendezvous* among threads. Thus, it is expected that



(a) Appia. (b) JGroups.

(c) Spread.

Figure 5.7: Execution time on the Vacation benchmark.

In fact, as can be seen in Figures 5.7b and 5.7c, JGroups' performance is only slightly better than Spread's. Appia, in Figure 5.7a, achieves the best performance of the three GCS due to its intermediate nature with respect to JGroups and Spread. While Appia is not as unfair as JGroups, replicas do not experience the higher latency, on average, that the privilege-based protocol implemented by Spread imposes to ensure fairness.

5.5 Summary

In this Chapter we presented the results of an experimental evaluation of the implementation of a replicated STM described in § 4.4.

In § 5.2 we stated the settings under which our experiments were executed, and provided an in-depth analysis of the three different GCS used: Appia, JGroups and Spread.

The Red-Black Tree microbenchmark results are presented in § 5.3. We show that our replicated STM shows great scalability in this benchmark, until the GCS is maxed out. The workload in this microbenchmark is characterised by fast transactions, the majority of which are read-only (this is the only executed benchmark with read-only transactions) and thus do not require remote communication. The contention is low.

In § 5.4 we evaluate our system under more complex workloads from the STAMP suite, specifically Intruder, Genome and Vacation. Intruder is a highly contented workload with fast transactions. These characteristics are adverse to the optimistic nature of the transaction certification procedure, as confirmed by the observed abort rate.

In Genome, transactions are more consuming than in Intruder, and contention is low. But the program is composed of a sequence of steps that must be executed sequentially, *i.e.*, the steps themselves benefit from multiple threads, but threads must wait for each other when advancing to the next step. Under these circumstances the differences of the three GCS become negligible due to the synchronization between steps, as observed in the experimental results.

The Vacation benchmark, like Genome, features more lengthy transactions with low contention, but each thread's execution path is concurrent – threads do not wait for each other in order to complete their work.

All things considered, we believe that the presented results highlight our success in building an efficient, flexible, DSTM framework that provides a much more familiar programming model than the existing alternatives.

Conclusion

6.1 Concluding Remarks

Software Transactional Memory (STM) systems have become a powerful mechanism to develop concurrent programs, by preventing the programmer from dealing explicitly with concurrency control mechanisms. STM algorithms associate metadata with the memory locations accessed during a transaction's lifetime, and this metadata may be stored in-place, near the associated memory location, or externally, on a map which pairs memory cells with metadata. The implementation techniques for these two approaches are very different and each STM framework is usually biased towards one of them, only allowing the efficient implementation of STM algorithms following that approach, hence inhibiting the fair comparison with STM algorithms falling into the other.

TribuSTM is a Java STM framework which allows to implement STM algorithms using both approaches, thus enabling a fair comparison between the algorithms while, at the same time, providing a non-intrusive interface to applications. In this dissertation we propose a novel, efficient, solution to support arrays under the in-place approach.

Despite initially being studied in the context of chip-level multiprocessing, the benefits of STM over traditional concurrency control methods also make it an attractive model for distributed concurrency control. However, the existing Distributed STM (DSTM) frameworks are tied to a specific distributed memory model and provide an intrusive interface to applications. This dissertation addresses the problem of building a modular DSTM framework which transparently supports different distributed memory models, while providing a non-intrusive interface to applications.

We propose an extension to the TribuSTM framework to support DSTM. This extension cooperates with TribuSTM using clearly defined APIs, which allows to implement different distributed memory models and the associated protocols to support distributed transactions. We also keep the programming model non-intrusive, unlike other DSTM frameworks. This paves the way for the fair comparison of different DSTM models under the same workloads, *e.g.*, using the same benchmarks with minimal-to-node code changes required.

Using our proposed, general, framework, we provide an implementation of a replicated STM which used a Certification-based protocol to commit distributed transactions. We evaluated the replicated STM, with three different Group Communication Systems (GCS), under different workloads using well-known benchmarks. These GCS provide the Total Order Broadcast primitive required by the Certification scheme. Our evaluation (1) shows the performance of the replicated STM implemented with our proposed framework; and (2) provides insightful information on the relevance of the GCS implementation under different workloads.

In conclusion, the modular DSTM framework presented allowed us to easily implement an efficient replicated STM, which unlike all other distributed STMs, provides a non-intrusive interface to the applications.

6.2 Future Work

Interesting directions for future work include:

- Currently, our solution for transactional arrays does not support the creation or structural modification outside instrumented code. This shortcoming presents a challenging, technical, problem that can be investigated;
- Implementation of the state-of-the-art algorithms of replicated STM distributed commit and memory consistency (presented in § 2.3.1) on our replicated STM, and subsequent evaluation;
- Implementation of a distributed STM, as sketched in § 4.3, and possible comparison with replicated STM;
- Partial replication is still a challenging topic, one which has been largely unexplored in the context of distributed STM. This framework can aid in such research;
- Under a distributed STM environment, support data and thread/transaction migration considering the affinity between data items and threads/transactions;
- Applying the distributed STM model to cloud environments.

Bibliography

[AAES97] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par)*, volume 1300 of *Lecture Notes in Computer Science*, pages 496–503, 1997.

[app12] APPIA Communication Framework. http://appia.di.fc.ul.pt, September 2012.

[BAC08] Robert Bocchino, Vikram Adve, and Bradford Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 247, February 2008.

[BBC+06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. Programming, Composing, Deploying for the Grid. In *Grid Computing: Software Environments and Tools*, pages 205–229. Springer, 2006.

[BG83] Philip Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

[BLM06] Colin Blundell, E. Lewis, and Milo Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), February 2006.

[BM03] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[Car11] Nuno Carvalho. *A Generic and Distributed Dependable Software Transactional Memory*. PhD thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, 2011.

[Con12] OW2 Consortium. ASM. `http://asm.ow2.org`, January 2012.

[CRCR09] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D^{2}STM: Dependable Distributed Software Transactional Memory. In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 307–313, November 2009.

[CRR10] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Asynchronous Lease-Based Replication of Software Transactional Memory. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, Lecture Notes in Computer Science, pages 376–396, 2010.

[CRR11a] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. A Generic Framework for Replicated Software Transactional Memories. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA)*, pages 271–274, August 2011.

[CRR11b] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. SCert: Speculative certification in replicated software transactional memories. In *Proceedings of the Annual International Systems and Storage Conference (SYSTOR)*, May 2011.

[CRR11c] Maria Couceiro, Paolo Romano, and Luís Rodrigues. PolyCert: Polymorphic Self-Optimizing Replication for In-Memory Transactional Grids. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, pages 309–328, December 2011.

[CRS06] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, December 2006.

[DSS06] David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[DVL12] Ricardo J. Dias, Tiago M. Vale, and João M. Lourenço. Efficient Support for In-Place Metadata in Transactional Memory. In *Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 589–600, 2012.

[EGLT76] Kapali Eswaran, Jim Gray, Raymond Lorie, and Irving Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[Fou12a] Apache Software Foundation. Apache Commons Byte Code Engineering Library. http://commons.apache.org/bcel, January 2012.

[Fou12b] Free Software Foundation. GNU Compiler Collection. http://gcc.gnu.org/wiki/TransactionalMemory, January 2012.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[GK08] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 175–185, February 2008.

[GLPQ10] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Transactions on Computer Systems*, 28(2):1–32, July 2010.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, first edition, 1992.

[HLM06] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 41, pages 253–262, October 2006.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*, volume 5. Morgan & Claypool Publishers, second edition, December 2010.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 48, June 2005.

[HW90] Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[IBM12] IBM. IBM BlueGene/Q data sheet. http://public.dhe.ibm.com/common/ssi/ecm/en/dcd12345usen/DCD12345USEN.PDF, September 2012.

[Int12a] Intel. Intel C++ STM compiler. http://www.software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition, January 2012.

[Int12b] Intel. Transactional Synchronization in Haswell. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell, September 2012.

[jbo12] JBoss Application Server. http://www.jboss.org/jbossas, September 2012.

[jgr12] JGroups - A Toolkit for Reliable Multicast Communication. http://www.jgroups.org, September 2012.

[KA98] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 156–163, 1998.

[KAJ+08a] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. DiSTM: A software transactional memory framework for clusters. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 51–58, 2008.

[KAJ+08b] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Investigating software Transactional Memory on clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–6, April 2008.

[KSF10] Guy Korland, Nir Shavit, and Pascal Felber. Deuce: Noninvasive Software Transactional Memory in Java. *Transactions on HiPEAC*, 5(2), 2010.

[KT96] M. Frans Kaashoek and Andrew S. Tanenbaum. An Evaluation of the Amoeba Group Communication System. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 436–447, 1996.

[LC07] João M. Lourenço and Gonçalo Cunha. Testing patterns for software transactional memory engines. In *Proceedings of the ACM Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, page 36, New York, New York, USA, July 2007. ACM Press.

[MCKO08] Chí Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 35–46, 2008.

[MMA06] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 198–208, March 2006.

[Ora12] Oracle. The `java.lang.instrument` package. http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html, January 2012.

[PS03] Fernando Pedone and André Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79—-101, 2003.

[RCR08] Paolo Romano, Nuno Carvalho, and Luís Rodrigues. Towards distributed software transactional memory systems. In *Proceedings of the Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, September 2008.

[RFF06] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[spr12] The Spread Toolkit. http://www.spread.org, September 2012.

[SR11] Mohamed Saad and Binoy Ravindran. HyFlow: a high performance distributed software transactional memory framework. In *Student Research Posters of the International Symposium on High Performance Distributed Computing (HPDC)*, pages 265–266, June 2011.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, August 1995.

[sun12] `sun.misc.Unsafe`. http://www.docjar.com/docs/api/sun/misc/Unsafe.html, January 2012.

[uui12] A Universally Unique IDentifier (UUID) URN Namespace. http://www.ietf.org/rfc/rfc4122.txt, September 2012.

[VDL12] Tiago M. Vale, Ricardo J. Dias, and João M. Lourenço. Uma Infraestrutura para Suporte de Memória Transacional Distribuída. In *Proceedings of the Simpósio de Informática (INForum)*, pages 177–189, 2012.