João André Almeida e Silva

Licenciado em Engenharia Informática

Partial Replication in Distributed Software Transactional Memory

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadores :	Hervé Miguel Cordeiro Paulino, Prof. Auxiliar, Universidade Nova de Lisboa João Manuel dos Santos Lourenço, Prof. Auxiliar, Universidade Nova de Lisboa

	Júri:
Presidente:	Prof. Doutor João Alexandre Carvalho Pinheiro Leite
Arguente:	Prof. Doutor Paolo Romano
Vogal:	Prof. Doutor Hervé Miguel Cordeiro Paulino

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2013

Partial Replication in Distributed Software Transactional Memory

Aos meus pais.

Acknowledgements

I am thankful to my thesis advisers, Hervé Paulino and João Lourenço, not only for this opportunity and for introducing me to the world of research, but also for their guidance, advices and devotion during the elaboration of this thesis. I would also like to extend my sincerest gratitude to Ricardo Dias and, specially, to Tiago Vale for their effort, help and brainstorming, to whom I am greatly obliged.

I am grateful to Departamento de Informática, Faculade de Ciências e Tecnologia, Universidade Nova de Lisboa for kindly granting me with three scholarships during the M.Sc. course.

To my co-workers and friends who frequented the room of the Arquitectura de Sistemas de Computadores group (and not only), specially Helder Martins, Joana Roque, Andy Gonçalves, Diogo Sousa, João Martins, Lara Luís and Laura Oliveira, for all the moments of work, fun, procrastination and despair we shared. None of this would be the same without their presence.

I would also like to express my gratitude to the Wroclaw Centre for Networking and Supercomputing (WCSS) for providing me access to a testing environment.

To my parents, Luís and Glória, I am heartily thankful for their support and guidance over the years, and specially for providing me with this opportunity. To my sister, Sara, for her support and understanding in my bad humour days.

To all the people in my Scout group (Agrupamento 719 Arrentela), specially Patrícia Correia, Hélder Marques, Joana Coelho and Sara Silva, for their support when I needed the most and for all the moments of fun we shared, I am very thankful.

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

This work was partially supported by the Centro de Informática e Tecnologias da Informação (CITI), and by the Fundação para a Ciência e Tecnologia (FCT/MCTES) in the scope of the research project PTDC/EIA-EIA/113613/2009 (Synergy-VM).

Abstract

Distributed software transactional memory (DSTM) is emerging as an interesting alternative for distributed concurrency control. Usually, DSTM systems resort to data distribution and full replication techniques in order to provide scalability and fault tolerance. Nevertheless, distribution does not provide support for fault tolerance and full replication limits the system's total storage capacity. In this context, partial data replication rises as an intermediate solution that combines the best of the previous two trying to mitigate their disadvantages. This strategy has been explored by the distributed databases research field, but has been little addressed in the context of transactional memory and, to the best of our knowledge, it has never before been incorporated into a DSTM system for a general-purpose programming language. Thus, we defend the claim that it is possible to combine both full and partial data replication in such systems.

Accordingly, we developed a prototype of a DSTM system combining full and partial data replication for Java programs. We built from an existent DSTM framework and extended it with support for partial data replication. With the proposed framework, we implemented a partially replicated DSTM.

We evaluated the proposed system using known benchmarks, and the evaluation showcases the existence of scenarios where partial data replication can be advantageous, e.g., in scenarios with small amounts of transactions modifying fully replicated data.

The results of this thesis show that we were able to sustain our claim by implementing a prototype that effectively combines full and partial data replication in a DSTM system. The modularity of the presented framework allows the easy implementation of its various components, and it provides a non-intrusive interface to applications.

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

Resumo

A memória transacional por *software* distribuída (MTSD) é reconhecida como uma alternativa interessante para controlo de concorrência distribuído. Estes sistemas fazem uso de técnicas de distribuição e replicação total de dados, a fim de atingir requisitos como escalabilidade e tolerância a falhas. No entanto, a distribuição dos dados não oferece tolerância a falhas e a replicação total limita a capacidade de armazenamento total do sistema. Neste contexto, a replicação parcial de dados aparece como uma solução intermédia que combina o melhor das duas anteriores tentando mitigar as suas desvantagens. Esta estratégia tem sido explorada na área de investigação das bases de dados distribuídas, mas tem sido pouco abordada no contexto da memória transacional e, tanto quanto sabemos, nunca antes foi incorporada num sistema de MTSD para uma linguagem de programação de propósito geral. Assim, nós defendemos que é possivel integrar replicação total e parcial de dados em tais sistemas.

Como tal, desenvolvemos um protótipo de um sistema de MTSD para programas Java que combina replicação total e parcial. Como ponto de partida utilizámos uma infraestrutura já existente que foi estendida com suporte para replicação parcial. Com a infraestrutura proposta, implementámos um sistema de MTSD replicado parcialmente.

O sistema proposto foi avaliado com *benchmarks* conhecidos, e a avaliação mostra a existência de cenários onde a replicação parcial pode ser vantajosa, e.g., em cenários com pequenas quantidades de transações que modificam dados replicados totalmente.

Os resultados desta tese mostram que fomos capazes de sustentar a nossa hipótese através da implementação de um protótipo que efetivamente combina replicação parcial e total de dados num sistema de MTSD. A modularidade da infraestrutura apresentada permite a fácil implementação dos seus vários componentes e fornece uma interface não intrusiva para as aplicações.

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

Contents

List of Figures

List of Tables

1

Introduction

This thesis addresses partial data replication in distributed software transactional memory (DSTM) systems. Although this technique has already been used and experimented in distributed database management systems (DBMSs), there are significant differences which justify our work.

In this chapter we motivate this thesis, presenting the layout of our approach, and listing the main contributions achieved.

1.1 Motivation

Nowadays, distributed software architectures are a widely used solution either by the distributed nature of applications, given the increasing use of mobile devices and the ubiquity of the Internet, or by the need to ensure quality of service regarding availability and latency, which usually leads to the use of multiple data centers where data is replicated and geographically dispersed.

The services provided by these architectures commonly address requirements such as availability and fault tolerance using data distribution and replication techniques. Data distribution appeared as a means of workload distribution. This technique maximizes the system's total storage capacity, but provides no fault tolerance support, since the failure of a single node causes inevitable data loss. To that extent, full data replication appeared as a technique to provide fault tolerance on top of workload distribution. It maximizes data survival, but limits the system's total storage capacity to the capacity of the node with fewer resources. In this context, partial data replication is a more recent solution that tries to find a middle ground between distribution and full replication. To that end, each node replicates only a subset of the system's data.

Geo-replicated systems usually use a combination of both full and partial data replication. These systems, comprised by multiple data centers, have an integral copy of the entire data set in each data center and inside the data centers use partial replication, dispersing data among the available nodes. This combination provides fault tolerance, both inter and intra data center, and high availability. These systems manage huge amounts of data and usually require the management of concurrent accesses to shared data, thus requiring concurrency control mechanisms.

In this context, DSTM is emerging as an interesting research topic. It builds from the application of software transactional memory (STM), proposed in 1995 by Shavit and Touitou [ST95], to the distributed setting, presenting an alternative for distributed concurrency control.

STM is based on the transactional model, which was originally proposed in the database world. Its basic unit of work is an *atomic* action encasing one or more operations (accessing shared memory) that execute with an *all-or-nothing* semantics, i.e., either all operations succeed and execute as a single indivisible and instantaneous action, or none execute at all.

More recently, research in the STM field has addressed requirements such as scalability and dependability, bringing attention to the distributed setting [Cou+09; Kot+08; Man+06]. To that purpose, existing STM systems have been extended with communication layers, protocols for memory consistency, replication and distribution management, and so forth, creating DSTM systems. Many of these systems have been developed tackling just some of these components, and only recently some DSTM frameworks addressed all components through modularity [Car+11a; SR11; Val12]. The majority of the proposed frameworks are tied to an unique distributed memory model, e.g., data distribution or full data replication. Both models allow systems to scale since the workload can be distributed among multiple nodes, and full data replication provides fault tolerance since data is replicated. Then, a distributed memory model that combines the best of both worlds would be very desirable. Nonetheless, all of this must be weighted against the costs involved in synchronization, after all, where there is replicated data there has to be consistency.

1.2 Problem

Full data replication requires data updates to be propagated to all the nodes in the system, since each node has a full copy of the system's data and that replicated data must be coherent between nodes. This solution raises scalability issues, considering that, most likely, it will not scale in systems with many nodes. Furthermore, it may also become infeasible to have a full copy of the system's data in every node. As data grows, the load on each node's virtual memory management may require systematic use of swapping, degrading the system's overall performance. In fact, at some point, nodes may simply not have the resources required to store all the system's data.

In recent past, *partial data replication* [Alo97] has emerged. With this distributed memory model, each data item is replicated on a subset of the system's nodes, and no node has all the system's data. In a system with these characteristics, a transaction necessarily involves multiple nodes, more precisely, the nodes that have a copy of the data items accessed by the transaction. Since different transactions may access different data items, the set of nodes involved in each transaction are not necessarily the same, although they may not be disjoint. This can improve the scalability of replicated systems since updates only need to be applied to a subset of the system's nodes, allowing these to handle independent parts of the workload in parallel. Since updates only need to be sent to a subset of the system's nodes, less messages are sent through the network and the synchronization cost becomes smaller. Considering this, we can leverage from locality inside data centers (where data is usually partially replicated and require strong consistency properties), as the propagation of updates can be further confined to a subset of the system's nodes and not disseminated throughout the whole data center.

To the best of our knowledge, until this moment there are no implementations of DSTM systems harnessing partial data replication. Thus far, DSTM research has only addressed data distribution and full replication. Some partial replication protocols have already been proposed [Pel+12a; Pel+12b] and were applied in the context of data-grids (with support for transactions), but not in a DSTM system. It is then desirable to explore partial data replication in the context of DSTM, allowing the assessment of to what extent it is feasible and effective in this context and in which contexts of use (workloads, number of nodes, etc.) partial data replication is a better option than other strategies.

When tackling partial data replication in a DSTM system for a general-purpose programming language, we found the following main challenges: (1) what data items should be partially replicated; (2) how the partially replicated data items should be partitioned among the system's nodes; (3) how to identify and access remote data items; and (4) how to validate transactions in this type of environments. We propose solutions to these challenges throughout this document.

1.3 Proposed Solution

In this thesis we address the problem of implementing a modular DSTM system for a general-purpose programming language, namely Java, providing partial data replication as a possible distributed memory model. We claim that it is possible to combine both full and partial data replication in a DSTM system for a general-purpose programming language.

In our system, we use both full and partial data replication in a collaborative manner, allowing the programmer to decide what data should be full or partially replicated. Our main goal is to provide a non-intrusive programming model to applications and a modular framework allowing the implementation of its various components (particularly the data partitioning algorithms and the distributed validation protocols) supporting different combinations of both full and partial replication.

A DSTM framework with support for partial data replication can be naturally implemented by extending an existing DSTM framework. So, our system grows from previous work, namely TribuDSTM [Val12], a modular DSTM framework for Java that targets fully replicated environments. We extend TribuDSTM with new partial replication-specific components, regarding data and nodes' partitioning, and a new kind of distribution metadata that enable the system to process (remote) read operations of objects that are not locally replicated.

TribuDSTM provides a non-intrusive programming model, by rewriting the bytecode of application classes. Our extension to support partial data replication continues to be non-intrusive to applications, by performing additional instrumentation.

1.4 Contributions

This thesis describes the following contributions:

- The definition of software components that extend TribuDSTM [Val12] to support partial data replication in a modular fashion;

- An implementation of a DSTM system with support for partial data replication based on the aforementioned extensions, creating a system that offers a combination of full and partial data replication;

- An implementation of an algorithm, from the literature (SCORe [Pel+12a]), for distributed transactions validation and commit in transactional memory (TM) systems harnessing partial data replication; and

- An experimental evaluation of the implementation on a common microbenchmark, as well as on more complex benchmarks, allowing initial insights on the contexts of use (workloads, number of nodes, etc.) that best suit partial data replication.

1.5 Publications

Some of the contributions have been published prior to this thesis. The DSTM framework was featured in the article "Replicação Parcial com Memória Transacional Distribuída", Proceedings of the Simpósio de Informática (INForum), 2013 [Sil+13].

1.6 Outline

In this chapter, we introduced our work. We started by presenting the motivations and the layout of our proposed solution. We also presented the contributions featured in this thesis.

The rest of this document is organized as follows. In Chapter 2, we present the work related with this thesis. We begin by introducing the reader to the transactional model and its properties. Next, we lay the basis of STM, whose semantics and implementation strategies we present afterwards. Later, we discuss the existing techniques that enable DSTM, analysing the existing protocols and exploring new ideas from recent research. Chapter 3 introduces the TribuSTM framework and its evolution to the distributed setting, TribuDSTM. Since TribuDSTM was the starting point of our work we dedicate this chapter to the overviewing of both systems. Next, in Chapter 4, we describe the extensions done to TribuDSTM in order to support the partial data replication distributed memory model. We also provide a detailed overview of our implementation of a partially replicated STM using the extended framework. In Chapter 5, we present the results of our experimental evaluation study. Finally, Chapter 6 concludes this thesis with an overview of its main points and future work directions.

2

Related Work

In order to better understand the contents of this document it is essential to be aware of the subjects relevant to our work. To this extent, in this chapter we describe the work that more closely relates to this thesis.

We start by introducing the concept of *transaction* and its properties, in Section 2.1. Then, in Section 2.2, we describe the concept of STM which aims at bringing transactions from database into memory. We continue by presenting its semantics and implementation strategies. Lastly, in Section 2.3, we describe DSTM as an alternative for distributed concurrency control. We introduce the issues that arise from the distributed setting, some mechanisms that support that setting, followed by a survey of the state of the art in distributed commit and memory consistency protocols, for both full and partial data replication environments. Finally, we still do an overview of the state of the art in DSTM frameworks.

2.1 Transactional Model

Nowadays, general-purpose programming languages have powerful abstractions for sequential programming, but that is not the case when it comes to concurrent programming where explicit synchronization is required, making programs very cumbersome and error-prone.

For years, databases have been taking advantage of the performance achieved through parallelism. Most DBMS make use of *transactions* to free the programmer from synchronization concerns.

Transactions are in the core of the database programming model. A database transaction is an abstraction that encapsulates a sequence of operations (over a database), that execute with an *all-or-nothing* semantics, i.e., either all operations succeed and execute as a single indivisible and instantaneous action or none execute at all. In the database world a transaction is characterized by four attributes: *atomicity*, *consistency*, *isolation* and *durability*, known as the ACID properties [GR92; HR83]:

Atomicity Either every operation in a transaction succeeds (and the transaction succeeds) or none will, i.e., if one operation fails, the entire transaction fails and the database is left unchanged;

Consistency The database is always in a consistent state (the meaning of consistent state is entirely application dependent). The commit of a transaction will bring the database from a valid state to another valid state, according to all defined rules;

Isolation Concurrent transactions are unaware of each other's presence. The effects of a transaction that is in progress are hidden from the remainder, i.e., a transaction is not allowed to observe the internal, and possibly inconsistent, state of other transactions; and

Durability Once a transaction has been committed, its results are persistent in the database (are stored in persistent storage, e.g., hard disk) and are made available to the following transactions.

In 1977, Lomet explored the notion of an atomic action as a method of process structuring [Lom77]. He observed that these properties make transactions an interesting alternative to explicit synchronization. If a sequence of operations accessing shared memory was encapsulated in a transaction, the atomicity property would guarantee that all operations succeed or the transaction would fail. In turn, the isolation property would guarantee that each transaction would run with the illusion that it is the only one executing at that time, like in the sequential model.

2.2 Software Transactional Memory

The transaction concept is vastly used in the database world, but in order to bring this concept into memory we must consider the differences between these two worlds.

Database transactions access persistent data (e.g., in a hard disk), but memory transactions access data in volatile memory. The durability property requires database transactions to record their changes permanently into persistent storage, but memory transactions are not durable in that sense, since data in memory only exists as long as the program is running. So this property is dropped when tackling transactions in memory.

Data inside a database can only be modified using transactions in the DBMS. In turn, memory transactions modify data from a process' memory. Since the DBMS has exclusive control over the database data it can enforce the consistency property. But when dealing with transactions in memory the "transactional system" does not control the memory exclusively, so data can be accessed in a non-transactional way, thus possibly making the memory inconsistent.

TM was originally coined by Herlihy and Moss in 1993 [HM93]. They defined a transaction, in this context, to be a finite sequence of machine instructions, executed by a single process, satisfying two properties: *serializability*, i.e., the steps of one transaction never appear to be interleaved with the steps of another transaction, and *atomicity* (as explained in Section 2.1). In this proposal, the authors introduced new primitive instructions for accessing memory in a transactional way. The TM system was implemented by modifying a processor's cache and cache controller. Thus being the first proposal of hardware transactional memory (HTM).

Two years later, in 1995, Shavit and Touitou proposed a new approach [ST95], based on the hardware transactional synchronization methodology of Herlihy and Moss. The authors introduced STM, a programming model for general-purpose programming, based on the transactional model described previously.

2.2.1 Semantics

To better understand STM we start by introducing its behaviour and presenting some related terminology.

Being based on the transactional model, a transaction is the basic unit of work. In the context of TM, a transaction is a sequence of memory read and write operations that execute as a single indivisible and instantaneous operation, thus executing atomically. That means that theoretically it is as if one and only one transaction is executing at any given point in time. A transaction that completes successfully *commits* and one that fails *aborts*. The set of locations that a transaction has read from is referred as its *read set*, and the set of locations that it as written to as its *write set*.

Consider the two transactions in Figure 2.1, where initially $x = y = 0$[1]. T_1 increments the value of both x and y variables, while T_2 will loop forever if $x \neq y$. Conceptually no two transactions execute simultaneously, so this means there are two possible outcomes of the concurrent execution of these two transactions: (1) either T_1 executes before T_2, or (2) T_2 executes before T_1. Either way, T_2 would never loop because when it executes, the possible outcomes are

$$\begin{aligned} x = y = 1 &\quad \text{for case (1)} \\ x = y = 0 &\quad \text{for case (2)} \end{aligned}$$

A programming abstraction with a simple and clean semantics helps the programmer to better understand the programming construct. However, transactions have no agreed semantics [Har+10] although some attempts have been made [Aba+08; GK08; MG08]. Being so, there are several definitions used to describe the semantics of transactions.

[1] Throughout this document, unless stated otherwise, variables are assumed to be initialized with the value 0 (zero).

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

(a) Transaction 1 (T_1).

```
transaction {
  while (x ≠ y) {}
}
```

(b) Transaction 2 (T_2).

Figure 2.1: Two memory transactions (taken from [Har+10]).

Since the database world and the TM world share the concept of transaction, many database semantics can be applied to TM. The basic correctness condition for concurrent transactions is *serializability* [Esw+76; Har+10]. It states that the concurrent execution of transactions must be equivalent to some sequential execution of those same transactions. Even though serializability requires that transactions appear to run in a sequential order, that order does not need to be the real-time order in which they run, i.e., the system is free to reorder or interleave transactions, as long as the resultant execution remains serializable. Consider, for instance, two transactions, T_A and T_B, being that T_A executes before T_B. Respecting serializability, transaction T_A can appear to run after transaction T_B (even if its execution comes completely before T_B).

With this freedom comes the need for a stronger criterion, the *strict serializability* criterion [Har+10]. It requires that if a transaction completes before another starts, then the first must occur before the second in the equivalent sequential execution, e.g., if transaction T_A actually completes before transaction T_B starts, in every serialization order, T_A must appear before T_B.

Another criterion is *linearizability* [Har+10; HW90]. It states that, considering a transaction as a single operation, each transaction must appear to execute atomically at some point during its lifetime.

Weaker correctness criteria allow higher concurrency between transactions, by allowing non-serializable executions to commit. Snapshot isolation (SI) [Dia+11; Har+10; Rie+06] is one of those criteria. The key idea is that each transaction takes a memory snapshot at its start and then performs all read and write operations on its snapshot.

Figure 2.2 illustrates an example of a possible non-serializable execution. Consider two transactions, T_3 and T_4, which evaluate the same expression

$$x + y + 1$$

writing the subsequent result to different memory locations, labelled x and y, respectively. Under serializability, two results are possible

$$\begin{array}{ll} x = 1 \text{ and } y = 2 & \text{if } T_3 \text{ executes before } T_4 \\ x = 2 \text{ and } y = 1 & \qquad\text{otherwise} \end{array}$$

SI admits a third result which would be impossible under serializability

$$x = 1 \text{ and } y = 1$$

in which both transactions begin with the same snapshot and commit their disjoint update sets. This anomaly is called a *write skew*.

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

(a) Transaction 3 (T_3).

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

(b) Transaction 4 (T_4).

Figure 2.2: Two memory transactions producing a *write skew* (taken from [Har+10]).

These correctness criteria, taken from databases, can be applied to TM and they provide some intuition for the semantics of TM systems, but they fall short in two main areas: (1) they specify how committed transactions behave, but they do not define what happens while a transaction runs, and (2) criteria such as serializability assume that the database mediates on all access to data, and so they do not consider cases where data is sometimes accessed by non-transactional code.

One last correctness criterion is *opacity* [GK08; Har+10], that provides stronger guarantees about the consistency of the values read by a transaction. Opacity can be viewed as an extension to the classical database serializability criterion with the additional requirement that even non-committed transactions are prevented from accessing inconsistent states. It states that (1) all operations performed by every committed transaction appear as if they happened at some single, indivisible point during the transaction lifetime, (2) no operation performed by any aborted transaction is ever visible to other transactions (including live ones), and that (3) every transaction (even if aborted) always observes a consistent state of the system.

As stated above, the presented criteria do not consider the behaviour resultant from transactional and non-transactional access to the same data. But as STM is pushing its way into the general-purpose programming world, it has to cope with these mixed-mode accesses to data. This behaviour is defined by two concepts called *weak* and *strong atomicity* [Blu+06; Har+10].

Weak atomicity is a semantics in which transactions are atomic only with respect to other transactions, i.e., their execution may be interleaved with non-transactional code. Whereas strong atomicity is a semantics in which transactions execute atomically with respect to both other transactions and non-transactional code, treating each operation appearing outside a transaction as its own singleton transaction.

In short, serializability provides higher concurrency than linearizability, because it allows more freedom in reordering the operations of a transaction. Snapshot isolation is good for short read-write transactions (that conflict minimally) and long read-only transactions, but the semantics is not as intuitive as the others. Regarding non-transactional access to data, there is a trade-off. Strong atomicity provides strong consistency guarantees in exchange for implementations with less performance, whereas weak atomicity provides weaker consistency guarantees in exchange for implementations with better performance.

2.2.2 Implementation Strategies

Even for simple TM systems, many implementations are possible. Next, we introduce the main design choices to be considered when implementing semantics as the ones presented previously.

Since transactions execute concurrently, in parallel if possible, there is the need to control the concurrent access to shared data. A conflict occurs when two transactions operate on the same data item and at least one of them is a write (SI has a slightly different definition of conflict, since it circumvents read-write conflicts). The conflict is detected when the TM system determines that the conflict has occurred and takes measures to resolve it, executing some action to avoid the conflict, e.g., by delaying or aborting one of the conflicting transactions. These three events (conflict, detection and resolution) can occur at different times (but not in a different order) and be managed in different ways.

All the definitions in this section are based in [Har+10].

2.2.2.1 Concurrency Control

With *pessimistic concurrency control*, all three events occur upon data access. When a transaction accesses some shared data item, the system detects a conflict and resolves it. This type of concurrency control allows a transaction to claim exclusive ownership of data prior to proceeding, preventing other transactions from accessing it. This is usually achieved using a lock per piece of shared data. Implementations that use locks should ensure that transactions progress. In particular, *deadlocks* should be taken care of. A deadlock is a situation in which two or more competing actions hold at least a lock and request additional locks which are being held by another action (in a circular chain), and thus neither ever does, e.g., transaction T_A holds a lock L_1 and requests lock L_2, while transaction T_B holds L_2 and requests L_1.

With *optimistic concurrency control*, conflict detection and resolution can happen after the conflict's occurrence. This type of concurrency control allows multiple transactions to access data concurrently and to continue running even when they conflict, as long as the TM system is able to detect and resolve these conflicts before a transaction commits. Instead of deadlocks, these implementations should take care of *livelocks*. A livelock is similar to a deadlock, in the sense that prevents the involved actions to progress. However, in such cases the state of these actions constantly changes. For example, transaction T_A writes to x, then conflicts with transaction T_B which forces T_B to be aborted, whereupon T_B may restart, write to x, forcing T_A to be aborted, none of them progressing.

If a certain application's workload has many long-living transactions, then pessimistic concurrency control can be advantageous: once a transaction acquires the locks it needs, it will be able to run to completion, and it will not be continuously aborted due to a stream of short-living transactions. However, if the workload is more homogeneous (in terms of transaction's duration), optimistic concurrency control can be a better choice because it avoids the cost of locking and can increase concurrency between transactions. Other aspect is contention. Workloads with high contention favour pessimistic approaches while low contention favour optimistic ones.

2.2.2.2 Conflict Detection

With pessimistic concurrency control, conflict detection is straightforward due to the use of locks, since a lock can only be acquired when it is not already held by another thread-/entity. But, in systems with optimistic concurrency control, conflict detection can have various implementations. For instance, a conflict can be detected upon data access (like with pessimistic concurrency control). This approach is called *eager conflict detection*. Another implementation strategy is to detect conflicts in a *validation phase*, at which point a transaction checks all previously read and written locations for concurrent updates. Validation can occur at any time, or even multiple times, during a transaction's lifetime.

Contrary to eager conflict detection, *lazy conflict detection* detects conflicts on commit. Upon the attempt of a transaction to commit, its read and write sets are checked for conflicts with other transactions.

With regard to the kind of accesses that are treated as conflicts, optimistic concurrency control can be exploited in two ways. The system can identify conflicts only between concurrent running transactions. If so, the system uses *tentative conflict detection*, e.g., if transaction T_A has read x and transaction T_B writes to x, that is a conflict, even before either transactions commit. On the other hand, if the system considers conflicts between active and committed transactions, it uses *committed conflict detection*, e.g., in this case, T_A and T_B can continue in parallel, and a conflict is only detected when one of them commits.

Usually, eager mechanisms are coupled with tentative conflict detection, while lazy mechanisms are coupled with committed conflict detection.

There is a trade-off between mechanisms that avoid wasted work (i.e., where a transaction executes work that is eventually aborted) and mechanisms that avoid lost concurrency (i.e., where a transaction is stalled or aborted, even though it would eventually commit). Eager conflict detection tries to avoid wasted work by detecting conflicts early, but it can cause livelocks. Lazy conflict detection allows for higher concurrency, but allows for more wasted work.

2.2.2.3 Version Management

Since transactions cannot observe each other's intermediate state, according to the isolation property, TM systems require mechanisms to manage the tentative updates that transactions do. The first approach is *eager version management*. It is also known as *direct update*, since a transaction directly modifies the data in memory, and maintains an *undo-log* with a backup of the original values, allowing the original values to be restored if ever the transaction aborts. This strategy requires the use of pessimistic concurrency control, since transactions must acquire exclusive access to the memory locations they update.

The second approach is *lazy version management*. Also known as *deferred update* because the updates are delayed until the transaction commits. This approach maintains its tentative updates in a per transaction *redo-log* that stores the transaction's tentative updates. This log must be consulted by the corresponding transaction's read operations to obtain the most fresh value if it has been updated. When the transaction commits, it updates the actual memory locations from its redo-log. If the transaction aborts, the redo-log is simply discarded.

Eager version management requires the use of locks, which must be held until the transaction commits. With lazy version management, if a transaction commits, the new values have to be written into memory (with the difference that the transactional reads must consult the redo-log). In a workload with many long-living transactions, the eager approach is likely to be more advantageous since locks must be held for the corresponding memory locations until the end of the transactions, not allowing other (short-living) transactions to access these memory locations (and continuously abort the long transactions).

2.3 Distributed Software Transactional Memory

In the last decade, STM has been the subject of increasing research and it has proven that it is a viable alternative for concurrency control. As the TM and the database worlds share the concept of transaction, STM began to adopt (and adapt) the existing solutions used in databases to main memory management. And in order to use STM in the enterprise world, where systems are subjected to heavy workloads and unexpected failures, it faces requirements such as *scalability*, *high availability* and *fault tolerance*. For those reasons, the distributed counterpart of STM (DSTM) has also been proposed as an alternative for distributed concurrency control. And although STM has received much interest over the last decades, only recently the distributed setting has begun to draw attention, in order to enhance dependability and performance.

Supporting DSTM raises several issues. First of all, the distributed nature of the underlying physical support requires some sort of *communication layer*. And then, some key questions arise. Will data be located in every single node or just in a subset of these? How should transactions be *validated* and *committed*? Several approaches have been proposed to these questions. The communication layer can range from a group communication system (GCS) to regular network messaging. Data is either centralized, distributed or fully replicated, with transaction validation and commit protocols tailored for each specific approach.

As noted, STM systems share some resemblance with DBMS. Both are built from the transaction concept and control concurrent data accesses automatically. However, they also have some significant differences.

The execution time of memory transactions is significantly smaller than database transactions since they only access data in memory, thus not incurring in the expensive persistent storage accesses done by the latter [Rom+08]. Moreover, SQL parsing and plan optimization are absent in STM. In distributed DBMS all this makes the remote synchronization somewhat expensive. In turn, in DSTM systems the remote synchronization needs to be optimized, in order not to become too big of an overhead (regarding the execution time of a memory transaction being usually small).

Nonetheless, replicated and distributed databases can be a source of inspiration when developing DSTM systems, regarding memory consistency algorithms, replication techniques, and so on. Therefore, we now present some replication techniques that, even though they were originally proposed for the database field, can be applied when tackling the distributed setting of STM.

2.3.1 Support Mechanisms

Data distribution consists in partitioning data among the multiple nodes of the system without resorting to replication, thus increasing the amount of data that the system can store, and dividing the workload by the same nodes. Rarely, a transaction needs to access all of the system's data, so it does not need to be validated by every node. This allows the system to scale more easily, but on the other hand, the failure of a node can lead to the loss of that node's data.

Data replication is a technique that consists in creating and maintaining multiple copies of the system's data across multiple nodes, called *replicas*. By maintaining consistent replicas, one can increase the system's availability and fault tolerance, and simultaneously improve the system's performance by splitting the workload among the replicas.

Replication and distribution allow for systems to scale since the workload can be distributed among multiple nodes. Furthermore, it enables parallelism, naturally at the expense of synchronization costs in order to ensure data coherence between replicas.

The replication and distribution of STM systems across multiple machines require some support mechanisms regarding communication and remote validation and commit. We now present some popular mechanisms that support these techniques.

2.3.1.1 Two Phase Commit

The two phase commit (2PC) algorithm [Gra78] is the simplest algorithm for coordinating all the processes that participate in a distributed transaction, to decide whether to commit or abort the transaction. It can be seen as a type of consensus protocol. The process that submits the transaction works as the coordinator and the remainder as participants.

As the name suggests, the algorithm has two phases. In the first phase, also called voting/prepare phase, the coordinator announces the transaction to the participants. When the participants receive the transaction's read and write sets, each one tries to validate the transaction. Then, each participant replies to the coordinator with its vote (yes to commit, no to abort). The voting phase ends when the coordinator receives all the participants' votes. In the second phase, also called decision/commit phase, there can be two outcomes. If all the participants voted yes, the coordinator sends a commit message to all of them. Each participant completes its transaction and sends an acknowledge message back to the coordinator. The coordinator completes the transaction when it receives all the participants' acknowledge messages. In turn, if any of the participants voted no, the coordinator sends an abort message to all them. Each of the participants locally aborts its transaction and sends an acknowledge message back to the coordinator, which in turn, aborts the transaction when it receives all the participants' acknowledge messages.

At each node, all the data modified by the transaction must be "locked" in the first phase and "unlocked" at the end of the second phase.

The big disadvantage of this algorithm is that it is a blocking algorithm. Being so, if the coordinator fails in-between the two phases of a commit operation, some participants may never receive the response to its voting. Thus, preventing further transactions to commit. The three phase commit (3PC) algorithm [Ske82] adds one more phase to the protocol in order to avoid this situation.

2.3.1.2 Group Communication System

A GCS [Cho+01] is a software layer that offers communication primitives and provides some guarantees about reliable communication. These systems ensure that all the members in a group (of processes) receive a copy of the message sent to the group, usually allowing different ordering guarantees. These guarantees assure that all group members receive the same set of messages and every member agrees in the global message delivery order.

We now present the guarantees and properties of some communication primitives offered by common GCSs.

Atomic Broadcast Atomic broadcast (ABcast), also known as total order broadcast (TOB) [Déf+04], is a group communication primitive that allows the emission of a message to all registered processes with the guarantee that all of them will agree on the delivered set of messages and on the order in which they are delivered. ABcast can be defined through the `TO-broadcast` and `TO-deliver` primitives[2]. This primitive has the following properties:

[2] 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, and `A-multicast` and `A-deliver` are analogous but for the atomic multicast primitive.

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 the 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-broadcasted by *sender(m)*[3]; 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, e.g., with uniform integrity, a process is not allowed to deliver a message *m* twice, even if it is faulty.

Reliable Broadcast A broadcast primitive that satisfies all the properties stated above except uniform total order (i.e., that provides no ordering guarantees) is called reliable broadcast (RBcast) [CT96; Déf+04]. Its properties are defined through the `R-broadcast` and `R-deliver` primitives.

Optimistic Atomic Broadcast The ABcast primitive offers strong ordering guarantees, but that comes at a cost. Its implementation is very costly, because it needs a great number of communication steps and messages, thus introducing a significant latency in delivering the messages. The main goal of optimistic atomic broadcast (OABcast) [PS98] is to minimize the latency problems of ABcast. The idea is that, with high probability, messages broadcasts in a local area network are received totally ordered (e.g., like when network broadcast or IP-multicast are used). This is called the *spontaneous total order* property. Messages are *optimistically delivered* to the application as soon as they are received, so the application overlaps the messages' processing with the communication needed to ensure the messages' total order. Later, the messages are *finally delivered* (TO-delivered) in their final order. So, OABcast has an additional primitive, `Opt-deliver`, that delivers a message optimistically to the application. This primitive has the following properties:

Termination If a process TO-broadcasts a message *m*, then it eventually Opt-delivers *m*;

Global Agreement If a process Opt-delivers a message *m*, then all the processes eventually Opt-deliver *m*;

Local Agreement If a process Opt-delivers a message *m*, then it eventually TO-delivers *m*;

[3] We assume that every message *m* can be uniquely identified, and carries the identifier of its sender, denoted by *sender(m)*.

Global 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'; and

Local Order A process first Opt-delivers a message *m* and only then it TO-delivers *m*.

Figure 2.3 shows the advantage of OABcast against ABcast. If the spontaneous order is equal to the final order, it is possible to save time. And the probability of the final order to be equal to the spontaneous order is very high when all nodes are interconnected in the same local network segment [Rod+06].

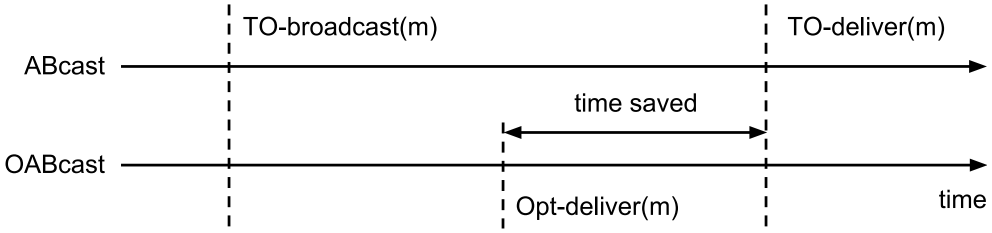


Figure 2.3: Optimistic atomic broadcast (adapted from [Rui11]).

Atomic Multicast The atomic multicast (AMcast) [Sch+10] is a variation of the ABcast primitive. While ABcast sends messages to all processes, AMcast allows the sending of messages to a subset of the existing processes. Similarly to ABcast, AMcast also guarantees that all the processes will agree on the delivered set of messages and on the order in which they are delivered. AMcast can be defined through the `A-multicast` and `A-deliver` primitives. For every message *m*, *m.dst* denotes the group of processes to which *m* is multicasted. This primitive has the following properties:

Validity If a correct process *p* A-multicasts a message *m*, then eventually all correct processes $p' \in m.dst$ A-deliver *m*;

Uniform Integrity For any process *p* and any message *m*, *p* A-delivers *m* at most once, and only if $p \in m.dst$ and *m* was previously A-multicasted;

Uniform Agreement If a process *p* A-delivers a message *m*, then eventually all correct processes $p' \in m.dst$ A-deliver *m*; and

Uniform Prefix Order For any two messages *m* and m' and any two processes *p* and p' such that $\{p, p'\} \subseteq m.dst \cap m'.dst$, if *p* A-delivers *m* and p' A-delivers m', then either *p* A-delivers m' before *m* or p' A-delivers *m* before m'.

2.3.2 Full Replication Environment

Replication is a natural way to deal with failures: if one replica fails, another one takes over. It offers fault tolerance and it also allows to increase the system's throughput by splitting the workload among the replicas. However, the challenge is how to keep the replicas consistent.

In a full replication environment, every node replicates all the system's data items and there is the need for algorithms that ensure data coherence across all these nodes.

We now present some replication techniques inspired in the database and distributed systems' literature. We also present an overview of some memory consistency protocols, the state of the art targeting full replication (that were already applied in the DSTM context). Some of the techniques presented ahead do not have a direct application in the DSTM context, but are presented for completion's sake.

2.3.2.1 Primary-Backup Replication

In this algorithm, also known as passive replication [AD76; Bud+93], one replica is designated as the primary and all the others as backups (at any given time, there is only one primary replica in the system). Clients make requests by sending messages only to the primary replica. Then, the primary replica updates all the backup replicas. If the primary replica fails, then a fail-over occurs and one of the backup replicas takes over.

The updates can happen in two ways: synchronously or asynchronously. In the first case, the primary replica stays blocked until all backup replicas respond with an acknowledge message, announcing that the update has been applied. In the second case, the updates can be deferred until a time that is best suited for the system (e.g., less activity in the system).

The main advantages of this technique are its simplicity, the fact that it involves less redundant processing and is less costly than the techniques presented ahead. On the other hand, requests can be lost when the primary replicas is overloaded with requests (additional protocols must be employed to retry such lost requests).

This technique was extensively applied in the database context and in distributed systems in general, but not in the DSTM context; since there is only one primary replica, that replica will become the system's bottleneck not allowing the system to scale.

2.3.2.2 State Machine Replication

This approach involves all the replicas in the processing of requests, thus having no centralized control. In this approach, also known as active replication [Sch90], client requests are broadcasted to all replicas, which process the request and in the end send the response back to the client. The client can wait for the first, a majority (voting) or all responses.

The main advantage of this technique is that it completely masks the effects of failures (by voting) to the client. On the other hand, it requires a communication primitive that guarantees that all replicas receive requests in the same order (like some of the primitives discussed in Section 2.3.1.2) and, so, the complexity is hidden by the communication primitive. Another disadvantage is the fact that this technique only works with deterministic requests, i.e., given the same initial state and a request sequence, all replicas will produce the same response sequence and end up in the same final state. Otherwise, replicas can diverge and the system becomes inconsistent.

The AGGRO system [Pal+10] combines active replication with the OABcast communication primitive (Section 2.3.1.2) to build a DSTM system.

2.3.2.3 Certification-based Replication

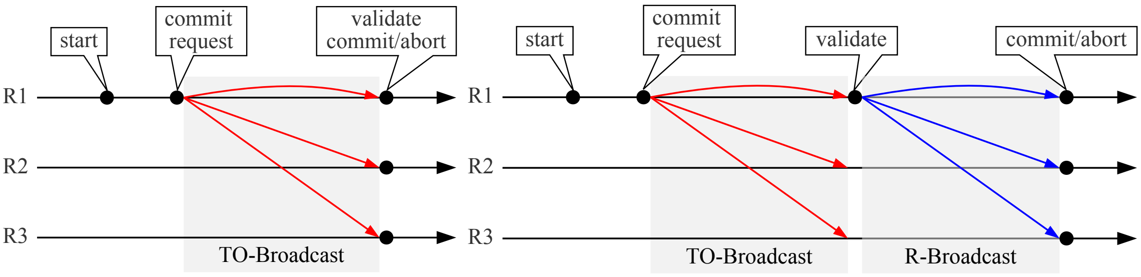
Certification-based protocols in a full replication environment allow the localization of transaction execution. A transaction executes locally in a single replica, only requiring synchronization among all other replicas at commit time. In fact, read-only transaction do not need remote synchronization at all. For write transactions the main idea is to defer the updates (Section 2.2.2.3) of a transaction until the latter is ready to commit, and rely on GCS primitives for communication.

Next we present some certification protocols. Some inspired in the database literature, others designed specifically for the DSTM context.

Non-Voting Certification When a transaction T, executing at replica R, enters its commit phase, it TO-broadcasts both its write and read sets [Agr+97]. Thus, enabling all other replicas to independently validate, and abort or commit T, as they are in the possession of all the necessary information, i.e., both the write and read sets. As the total order of the deliveries is guaranteed (by the TOB primitive), all replicas will process all transactions in the same order, making the outcome of any validation step the same on all replicas. Figure 2.4a illustrates this protocol.

Voting Certification In non-voting certification protocols, to commit a transaction, both write and read sets are disseminated in a single communication round. In common workloads the write set is typically much smaller than the read set. So, this protocol explores the trade-off of exchanging potentially smaller messages (the read set, typically much larger than the write set, is not disseminated) at the expense of requiring two communication rounds (instead of just one) [KA98]. When a transaction T, executing at replica R, enters its commit phase, it only TO-broadcasts its write set. Thus, only R is capable of validating T, as it is the only replica with both the write and read sets. If R detects a conflict during the validation of T, it R-broadcasts an abort message notifying the other replicas to discard T's write set and aborts T on R. Otherwise, a commit message is R-broadcasted instead, activating the application of T's write set on all replicas, as shown in Figure 2.4b. Both the abort and commit messages only need the reliable broadcast primitive (which does not ensure total ordering, thus being cheaper) because T was already serialized by the TO-broadcast of its write set.

Bloom Filter Certification Bloom filter certification (BFC) is an extension/variant of the non-voting protocol that uses bloom filters to reduce the size of the broadcasted messages [Cou+09], as the efficiency of the TOB primitive is known to be strongly affected by the



(a) Non-voting certification.

(b) Voting certification.

Figure 2.4: Certification-based protocols (taken from [Val12]).

size of the exchanged messages [Rom+08]. This protocol was devised with DSTM in mind, and its first application was in the DSTM context. A bloom filter [Blo70; BM03] is a space-efficient probabilistic data structure that is used to test if an element is a member of a set. False positives are possible but false negatives are not, i.e., a query returning true actually means "may be in the set", and a query returning false means "definitely not in the set". Before transaction T TO-broadcasts its write and read sets, the read set is encoded in a bloom filter whose size was computed to ensure that the probability of a transaction abort due to a false positive (when querying the bloom filter) is less than a user-tunable threshold.

Asynchronous Lease Certification Certification-based protocols are inherently optimistic, since transactions are only validated at commit time and no bound is established on the number of times that a transaction will have to be re-executed due to conflicts, sometimes leading to undesirable high abort rates. This is obvious in workloads that contain both short and long-running transactions, where the latter ones may be repeatedly and unfairly aborted due to conflicts with a stream of short-running transactions. Asynchronous lease certification (ALC) tackles these problems using the notion of asynchronous lease [Car+10]. A lease can be seen as a token that grants its owner privileges over the management of a subset of a transaction's dataset. Leases are asynchronous because they are detached from the notion of time, i.e., leases are granted in an acquire/release base. In order for a transaction T to commit, the replica R where T executed must have established a lease for the accessed data items prior to proceed with its validation. If T is found to have accessed outdated data, it is aborted and re-executed without releasing the lease. This ensures that T will not be aborted again due to remote conflicts, as no other replica can update the data items protected by the held lease, provided that T deterministically accesses the same set of data items as the first execution.

Polymorphic Self-Optimizing Certification Non-voting protocols disseminate both the transaction's write and read sets, allowing each replica to locally certify transactions. This is optimal in terms of communication steps. Voting protocols avoid broadcasting the transaction's read set, thus drastically reducing the size of the disseminated message.

On the other hand, this makes a second communication step necessary. Both protocols incur in a trade-off between communication steps and message size, and thus they are designed to ensure optimal performance in different workload scenarios (they can exhibit up to 10x difference in terms of maximum throughput [Cou+11]). To deal with this dichotomy polymorphic self-optimizing certification (PolyCert) supports the coexistence of the voting and non-voting/BFC protocols simultaneously [Cou+11], in the DSTM context, by relying on machine-learning techniques to determine, on a per transaction basis, the optimal certification strategy to be adopted.

Speculative Certification With high probability, messages broadcasts in a local area network are received totally ordered. The spontaneous total order property used by the OABcast primitive is exploited in order to allow fast delivery of messages (Section 2.3.1.2). Speculative certification (SCert) leverages on the OABcast's optimistic delivery to overlap computation with the communication needed to ensure the messages' total order [Car+11b]. Once a transaction reaches its commit phase, it is first locally validated and then its write and read sets are disseminated to all replicas by means of an OABcast. As soon as a transaction T is optimistically delivered, SCert speculatively certifies T instead of waiting for its final delivery, as conventional certification protocols. If validation if successful, T is said to be speculatively committed, i.e., the validation creates speculative versions of the data items updated by T. Eventually T is finally committed if the final delivery matches the optimistic (its speculative versions are actually written). If it does not match, T is aborted unless it is still successfully certified in a new serialization order. Speculative versions of data items are immediately visible to new transactions, hence tentatively serializing the transactions after the speculatively committed ones. This allows an overlap between computation and communication by certifying transactions while the OABcast computes the final order and to detect conflicts earlier (e.g., as soon as a transaction T is speculatively committed, any other local transaction that was serialized before T and that has read, or reads, a data item updated by T is immediately aborted) avoiding wasted computation and time on transactions doomed to abort.

2.3.3 Partial Replication Environment

Full replication forces the result of every transaction to be propagated to every replica in the system. That may not scale with the increase of the number of replicas. It may also become infeasible to have a full copy of the system's data when that data grows, simply because the replicas may not have enough space to store it. In fact, the system is limited by the replica with less resources.

Partial replication refers to a replication approach in which replicas do not replicate all the system's dataset [Alo97]. Instead, each replica only stores a subset of the given dataset and that subset can have copies in different replicas. The main idea of partial replication is that not all replicas have to process a transaction. A transaction only has to be sent to the set of replicas that replicate the data items it accesses. Thus, less messages are sent through the network and the coordination cost between replicas becomes smaller [Cou+05]. Unlike full replication, partial replication can increase access locality and reduce the number of messages exchanged between replicas [Cou+05]. Partial replication can be seen as a combination of the advantages of full replication and distribution while trying to diminish their disadvantages.

2.3.3.1 Classification of Partial Replication Protocols

Partial replication protocols can be classified into three categories, according to what information about transactions replicas need to store [Sch+06; Sch+10]:

Non-genuine For every submitted transaction T, all replicas store information about T, even if they do not replicate data items read or written by T;

Quasi-genuine For every submitted transaction T, correct replicas that do not replicate data items read or written by T permanently store not more than the identifier of T; and

Genuine For every submitted transaction T, only replicas that replicate data items read or written by T exchange messages to certify T.

Non-genuine protocols go against the main idea of partial replication, since every replica has to be involved in the processing of every transaction, which leads to the communication overhead problems of full replication.

2.3.3.2 Partial Replication in the DBMS Context

Partial replication in distributed systems, mainly databases, is a recent technique. In 1997, Alonso [Alo97] addressed this problem and described under what conditions can partial replication be supported using group communication primitives. In the following years, several solutions were presented.

Sousa et al. [Sou+01] presented an approach based in the 2PC algorithm (Section 2.3.1.1), named *resilient atomic commit*. Since replicas hold a partial copy of the system's dataset, they cannot decide to commit a transaction based only on the local validation; they should also consider data items stored in other replicas and decide on a common basis. In this approach, when a transaction enters its commit phase, it sends its write and read sets to all replicas using the OABcast communication primitive (Section 2.3.1.2). Similarly to 2PC, each replica does a local validation (involving every data item accessed by the transaction for which that replica holds a copy) and then sends back its vote. Every replica broadcasts its vote and starts gathering votes from the remainder until it can reach a decision. Because of the OABcast primitive, if the final order of the transaction is different from the tentative order the algorithm starts once again.

In 2006, Schiper et al. [Sch+06] followed a different approach. In this algorithm, the transaction's write and read sets are sent to all replicas using the RBcast primitive (Section 2.3.1.2), not guaranteeing total order. It uses a consensus abstraction in order to agree in a serializable order for the transactions. For each transaction decided in the consensus, every replica (that has data items accessed by the transaction) sends its vote to the remainder. If every replica receives positive votes the transaction is committed.

Most of the research carried out in this field resorted to the traditional consistency criterion for replicated systems, 1-copy-serializability (1SR) [Ber+87], whereby a data item must appear as one logical copy and the execution of concurrent transactions must be coordinated so that it is equivalent to a serial execution over the logical copy.

In [Ser+07], Serrano et al. identified 1SR as a limitation when designing scalable replicated solutions. Thus, they presented an algorithm with 1-copy-snapshot-isolation (1SI), whereby replicas run under snapshot isolation (Section 2.2.1), as the consistency criterion. Its main advantage is that there is no need to send the read set when certifying transactions, reducing the size of the exchanged messages. Moreover, read-write conflicts are not detected, reducing the abort rate. When a transaction T starts, it gets a start timestamp ($T.ST$) and when it enters its commit phase it gets a commit timestamp ($T.CT$), and sends its write set (in total order) to all the other replicas. Every replica (that holds data items accessed by the transaction) certifies the transaction if the following condition is true:

$$\nexists T' \in commitedTx : T'.ST \leq T.CT \leq T'.CT \wedge T.writeset \cap T'.writeset \neq \emptyset$$

P-store [Sch+10] is a genuine partial replication protocol for wide area networks. In this protocol, replicas are divided into groups and data is partitioned between the groups so as to ensure that, in the same group, all replicas replicate the same data items (data items in different groups does not need to be disjoint). When a transaction enters its commit phase it multicasts (Section 2.3.1.2) a message (with its write and read sets) to every replica that holds a copy of data items accessed by the transaction. When the message is received it is stored in a queue of transactions to be validated. If a transaction is local, i.e., if the data items accessed by the transaction belong to only one group, each replica can individually decide whether to commit or abort; otherwise, it is said that the transaction is global and all the replicas (that hold a copy of data items accessed by the transaction) exchange votes in order to decide the outcome of the transaction.

2.3.3.3 Partial Replication in the TM Context

As stated previously, partial replication is a recently explored technique. To the best of our knowledge, until this moment there are no implementations of DSTM systems exploiting partial replication, although some protocols have already been proposed.

In [Pel+12b], Peluso et al. present a genuine partial replication protocol for transactional systems, named GMU. It uses a highly scalable, distributed multi-versioning scheme. Moreover, read-only transactions never abort and do not need to undergo distributed validation. GMU ensures extended update serializability (EUS) [HP86] as its consistency criterion. It is a particularly attractive criterion, since it is sufficiently strong to ensure correctness for demanding applications, but also weak enough to allow efficient and scalable implementations. EUS serializes all update transactions. However, unlike 1SR, it allows concurrent read-only transactions to observe snapshots generated from different linear extensions of the history of update transactions. To commit update transactions, GMU uses a 2PC protocol (Section 2.3.1.1), involving only the replicas that replicate data items accessed by the transaction (see Figure 2.5). Upon receiving a prepare message, each replica acquires read and write locks on all data items read or written by the transaction. Next, they validate the read set and send back to the coordinator the vote message with the proposal of a new vector clock for the transaction. If all replies are positive, the coordinator builds a commit vector clock and sends back the commit message. This protocol blends into de 2PC messaging pattern a distributed consensus scheme that resembles the one used by Skeen's total order multicast algorithm [Déf+04].

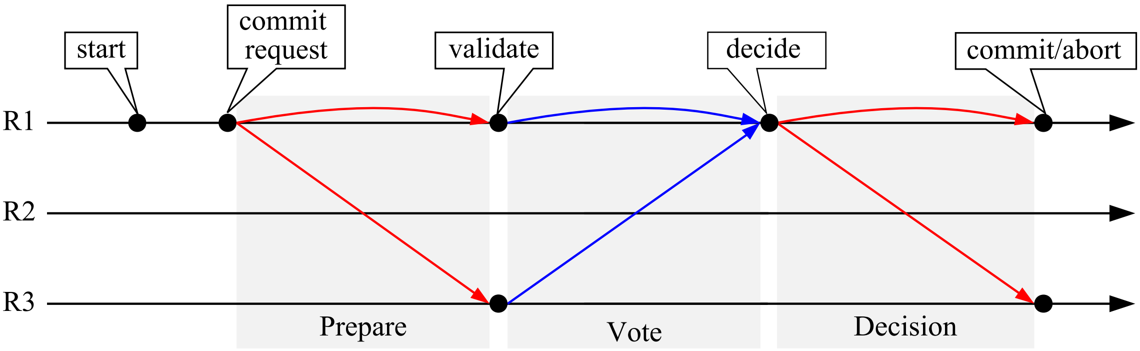


Figure 2.5: Atomic commit protocol used in GMU.

With SCORe [Pel+12a], Peluso et al. go back to the 1SR consistency criterion. They present a genuine partial replication protocol guaranteeing 1SR (at no additional overhead). SCORe uses a multi-version concurrency control scheme, which is coupled with a highly scalable distributed logical clock synchronization scheme that only requires the exchange of a scalar clock value among the nodes involved in the validation of a transaction. It never aborts read-only transactions and spares them from any distributed validation. SCORe relies on a genuine atomic protocol that can be seen as the fusion between the 2PC and the Skeen's total order multicast [Déf+04] (this protocol is quite similar to the one used by GMU (Figure 2.5); the major difference is in the decide phase, where SCORe guarantees that the commit events of all update transactions are totally ordered across all the replicas of a same partition[4]). When a transaction enter its commit phase, the coordinator sends a prepare message to all the involved replicas. Upon receiving the message, each replica verifies whether the transaction can be serialized after every transaction that

[4] The authors abstract over the data placement policy by assuming that data is subdivided across m partitions and that each partition is replicated across r nodes.

has locally committed so far. To this end, it attempts to acquire exclusive and shared locks, respectively for the data items in the transaction's write and read sets, that it locally maintains. Next, it validates the transaction's read set. If any of these operations fail, the transaction will abort, as in classic 2PC. If the transaction passes the validation phase, SCORe explores the vote message of 2PC to overlap a distributed agreement scheme, that aims at establishing the final serialization order for the transaction.

2.3.4 Frameworks

With the advance of DSTM research some memory consistency protocols have been proposed. Naturally, some frameworks have also been developed with the purpose of facilitating the development, testing and evaluation of such protocols. We now overview the state of the art in DSTM frameworks.

2.3.4.1 DiSTM

In [Kot+08] Kotselidis et al. introduced a research platform for easy exploiting of STM on clusters. The authors designed a framework for easy prototyping of TM coherence protocols called DiSTM, which is built on top of DSTM2 [Her+06]. In this system, one of the nodes acts as the *master node* where global data is centralized, while the rest of the nodes act as *worker nodes* and maintain a cached copy of that global data (Figure 2.6). Along with the system, three cache coherence protocols were proposed.

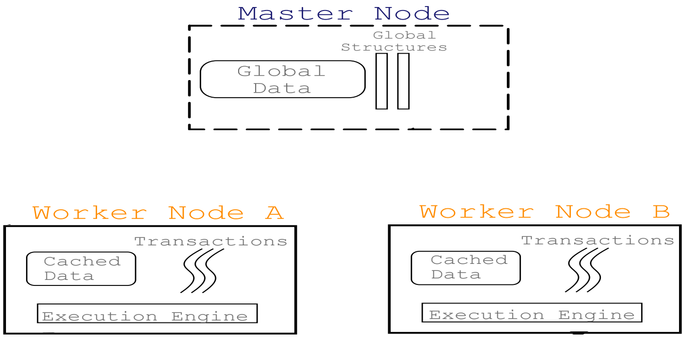


Figure 2.6: DiSTM architecture (adapted from [Kot+08]).

Communication is based on the ProActive framework [Bad+06]. The main idea of this framework is based on *active objects* [LS96], a standard object with its own thread of execution. These objects can be distributed over the network and its activity and localization (local or remote) are completely transparent. Asynchronous requests are sent to an active object and are stored in its request queue before being served according to a service policy.

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

2.3.4.2 D^{2}STM

In [Cou+09], Couceiro et al. leverage from replication to improve performance and also to enhance dependability. Therefore data is replicated across all nodes (i.e., is fully replicated) of the distributed system. Their contribution features a replica coordination protocol based in bloom filters that enables high compression rates on the messages exchanged between nodes at the cost of a tunable increase in the probability of transactions abort due to false positives.

Figure 2.7 shows the components which constitute a D^{2}STM node. The framework is based on the Java versioned software transactional memory (JVSTM) [CRS06], a STM framework and communication is achieved through a GCS implementing a TOB primitive.

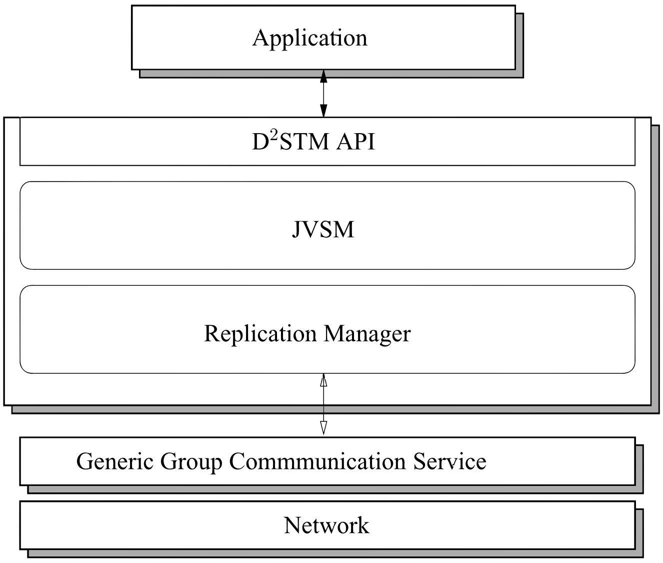


Figure 2.7: Components of a D^{2}STM replica (taken from [Cou+09]).

Between the communication and the STM layers lies the core component of this system. The replication manager implements the distributed coordination protocol required for ensuring replica consistency. It integrates with the STM layer, by having the ability to inspect the internals of transactions execution, explicitly triggering transactions validation and atomically applying the write sets of remotely executed transactions.

2.3.4.3 GenRSTM

Like D^{2}STM, GenRSTM [Car+11a] is a framework for fully replicated STM. From the point of view of the components added to support the replicated setting it is very identical to D^{2}STM, and it seems to be an enhanced version of the latter (Figure 2.8). Specifically, the STM layer can be exchanged as long as it provides an application programming

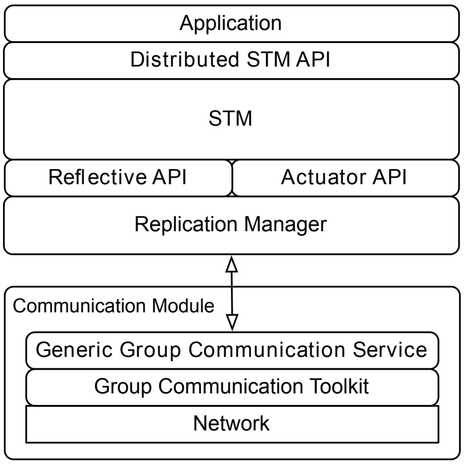


Figure 2.8: GenRSTM node architecture (taken from [Car+11a]).

interface (API) like the one of JVSTM, based on boxes.

In this system, the programmer explicitly marks which objects are replicated by having their classes extend `GenRSTMObject`, a base class supplied by the framework. Transactions also have to be explicitly programmed with the `begin` and `commit` operations.

2.3.4.4 HyFlow

HyFlow [SR11] is a framework for DSTM, with pluggable support for directory lookup protocols, transactions' validation protocols, contention management policies, cache coherence protocols, and network communication protocols.

Figure 2.9 shows the system's architecture. It is composed by five main components:

Transaction Manager Encapsulates the local STM algorithm;

Instrumentation Engine Modifies classes code at load time, e.g., for modifying annotated methods to support transactional behaviour;

Object Access Module Not only provides access to the objects owned by the current node, but is also 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 Enables the network communication between the various nodes of the system.

Objects can be distributed over the network, therefore normal references cannot be used to access them. As such, this framework requires that any distributed class implements the `IDistinguishable` interface (which comprises the method `getId()` that

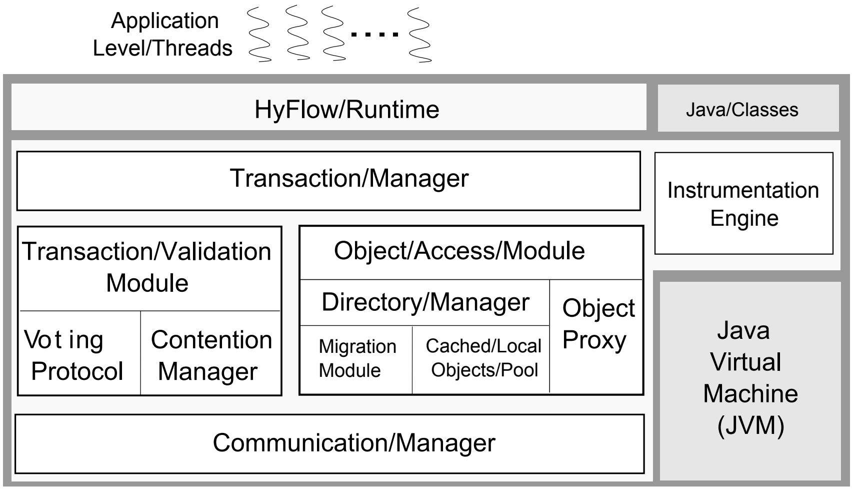


Figure 2.9: HyFlow node architecture (taken from [SR11]).

returns an unique object identifier (UOID)). Now, where before fields held regular references, they must maintain these UOIDs.

Distributed classes can provide remote methods which can be invoked regardless of their objects' location, like Java remote method invocation (RMI). These methods are defined by the programmer annotating them with @Remote.

Since fields maintain UOIDs, in order to obtain a reference to a distributed object the framework has a `Locator` instance from the directory manager component that encapsulates the directory lookup protocol. This component is used to retrieve objects given their UOID. Transactions are defined as methods annotated with @Atomic.

2.3.4.5 TribuDSTM

TribuDSTM [Val12] is an efficient and modular DSTM framework for Java. Its main feature is its non-intrusive public API, only requiring the use of an @Atomic annotation in the methods that should run as transactions. Everything else is hidden by the instrumentation of the Java bytecode.

Figure 2.10 depicts TribuDSTM's architecture. TribuDSTM uses TribuSTM [Dia+12; Dia+13] for local concurrency control, and it implements other two components: a distribution manager and a communication system. The distribution manager implements a distributed/shared memory system, according to the desired distributed memory model, as well as the protocols for distributed transactions validation. The communication system encases the necessary communication primitives for the implementation of the distribution manager, providing an uniform interface, independently from the used GCS.

Due to its modularity, the framework allows various implementations of the different components, allowing other distributed memory models.

From all the existing DSTM frameworks, TribuDSTM stands out as providing an extremely simple and intuitive public API, "hiding" the distribution with Java bytecode

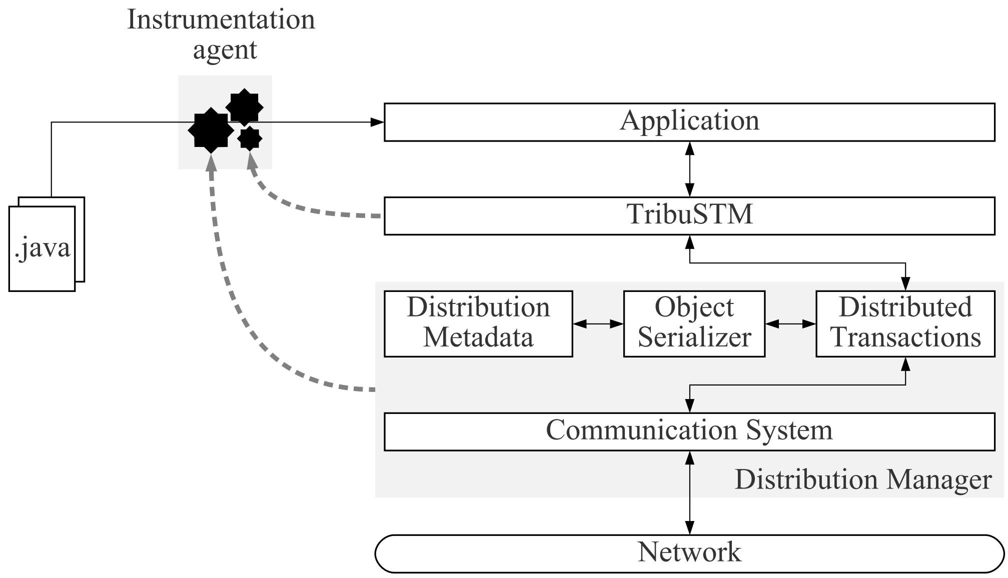


Figure 2.10: TribuDSTM node architecture (taken from [Val12]).

instrumentation. All the other frameworks expose the distribution to the applications requiring classes to extend some framework-provided class or to implement some interface. TribuDSTM's modularity is also advantageous since it allows us to easily extend and implement the different components.

2.4 Summary

Initial work in the DSTM area was focused in clusters and in scalability requirements. Manassiev et al. [Man+06] described a DSTM system targeting clusters of workstations, built on a distributed shared memory system. In turn, Kotselidis et al. [Kot+08] investigated a similar approach, but at the level of objects within a Java virtual machine (JVM) (Section 2.3.4.1). Bocchino et al. [Boc+08] designed a DSTM for use across systems that might scale to thousands of nodes, employing techniques to reduce communication between nodes.

More recent investigation has shifted to full replication and corresponding memory consistency protocols. Couceiro et al. [Cou+09] present a fully replicated STM that uses a certification-based protocol to enforce consistency (Section 2.3.4.2). Other works [Car+10; Car+11b; Cou+11] proposed alternative memory consistency protocols, also in the scope of full replication.

Some DSTM frameworks have also been designed, such as DiSTM [Kot+08], HyFlow [SR11], D^{2}STM [Cou+09], GenRSTM [Car+11a] and, more recently, TribuDSTM [Val12].

When it comes to partial replication, research is not so mature as in full replication. DSTM systems harnessing partial replication have yet not been developed, although some applicable algorithms have been proposed. Both proposals, presented in Section 2.3.3.3, are recent and were only applied to a transactional in-memory key/value

NoSQL data-store and data-grid (RedHat's Infinispan[5]).

As seen in Section 2.3.2, certification-based protocols (targeting full replication) can be easily implemented (Vale [Val12] implemented the non-voting certification protocol in under 100 lines of code (LOC)). But protocols targeting partial data replication are inherently more complex, and the complexity is even higher when we add genuineness to protocols.

Section 2.3.3 presented the state of the art regarding partial data replication and it is visible the variety of approaches followed. The various protocols make use of different communication primitives trying to hide the protocols' complexity. There are some implementations of total order broadcast with good performance, but it removes the genuineness from the protocols since messages are sent to all the system's nodes. Total order multicast enables genuine protocols, but the majority of the existing GCSs still does not support it. Also, data partition across the system is a complex problem. One could see a system with an oracle that knows the entire state of the system's data and could decide the best node where to execute a specific transaction. Or perhaps, a system that uses some heuristic to partition data across the system and distributes the transactions randomly among the nodes, or according some other heuristic respecting the data partitioning strategy. One could easily foresee many different approaches. Could a non-genuine protocols have better performance than a genuine one? All the genuine protocols are variants of the 2PC protocol, but certification-based protocol are known for their simplicity. Is it possible to implement a certification-like protocol targeting partial data replication, simpler than the others proposed?

This chapter presented the foundations for our work. We started by familiarizing the reader with the transactional model, the corresponding transaction concept and its properties (Section 2.1). We then bridged from the databases into STM, bringing transactions into memory, in Section 2.2. We covered its semantics, in Section 2.2.1, and possible implementation alternatives that can be applied in order to achieve such semantics (Section 2.2.2).

Entering the DSTM world, in Section 2.3, we introduced the issues that arise from applying STM into the distributed setting. We then, overview some popular support mechanisms that enable DSTM (Section 2.3.1). Next, we addressed full replication techniques, in Section 2.3.2, some of them taken from the databases literature. Lastly, we presented some of the proposals for partial replication (Section 2.3.3), where the majority comes from the database world, with the exception of two proposals (Section 2.3.3.3) that have been applied to a transactional in-memory distributed storage system, with promising results. Thus, making partial replication a way for developing highly scalable systems. We finished this section with an overview of the state of the art in DSTM frameworks.

[5] http://www.jboss.org/infinispan

3

TribuDSTM

The software architecture developed in the scope of this thesis grows from previous work, namely TribuSTM and its extension to distributed environments, TribuDSTM. As such, we dedicate this chapter to the overviewing of both systems. We start, in Section 3.1, by briefly introducing the Deuce Java STM framework and motivate the need for TribuSTM. We also examine the metadata placement strategy as provided by Deuce, and describe a new strategy featured in TribuSTM. In Section 3.2, we present the extensions to TribuSTM in order to support DSTM. We first provide an overview of the architecture and then detail, in more depth, its specific components. The content of this chapter is heavily based in [Val12].

3.1 TribuSTM

Every STM algorithm associates some kind of information, which we call *transactional metadata*, to each memory location (or object reference) accessed within a transaction. This information is specific to each algorithm and may be constituted by, e.g., locks, timestamps or lists of values. That metadata can be stored either in an external data structure that associates it to the corresponding memory location (*out-place* or *external* strategy), or alongside the memory location (*in-place* strategy).

The external strategy can be implemented using a hash table that matches the memory location to its metadata. For performance reasons the table is pre-allocated (avoiding the overhead of dynamic memory allocation) and cannot be resized (which is known to be a very heavy operation). Moreover, a very fast hashing function is used and there is no collision resolution. This configuration imposes a big limitation: two or more memory locations can be paired with the same metadata. This option is valid for algorithms whose metadata does not have a strong bond with the associated memory locations, e.g., TL2 [Dic+06] whose metadata are locks, since it results in transactions conflicting in practice even though they actually should not, incurring in a performance penalty. However, when metadata depend on the associated memory locations, e.g., the list of values associated with each memory location in a multi-version algorithm like JVSTM, the external strategy is not enough since it would be unacceptable for two memory locations to share the same list of values.

We can conclude that using the external strategy is acceptable when metadata is not strongly tied to its memory location, i.e., the relationship between a location and metadata can be N to one. If it is necessary to have a one to one relationship, then the external strategy is inadequate and the in-place strategy is required.

The in-place strategy, in object-oriented programming, is typically implemented with a variation of the decorator design pattern [Gam+94] by wrapping the targeted object inside a container object which holds the original object and its associated metadata. This approach allows a very direct and efficient access to the metadata, but it is highly intrusive to the application code, which has to be rewritten in order to use the decorator's class in every object that has to be associated with metadata. Moreover, it does not efficiently handle primitive types and arrays. Primitive types have to be replaced by their object counterparts[1]. Arrays are a very strict structure to which we cannot add metadata, being the naive solution to use an array of decorator objects (having an array of objects when we could have an array of a primitive type).

3.1.1 Deuce

Deuce [Kor+10] is an efficient Java STM framework that can be added to existing applications without changing its compilation process or libraries.

In order to accomplish such non-intrusive behaviour, it relies heavily on Java bytecode manipulation using ASM [OW213], a general-purpose Java bytecode manipulation and analysis framework that can be used to modify existing classes or dynamically generate new ones. This instrumentation is performed dynamically as classes are loaded by the JVM using a Java agent [Ora13].

To deal with performance-related issues, Deuce uses `sun.misc.Unsafe` [Sun13], a collection of methods for performing low-level unsafe operations. Using `sun.misc.Unsafe` allows Deuce to directly read and write to specific memory locations. Deuce also allows to plug in custom STM implementations using the external strategy for metadata placement.

The only modification required in the application code is the @Atomic annotation in the methods that have to run as transactions, as all the transformations are performed behind the scene dynamically at class loading. Therefore, programming an application to be used with Deuce is no different from regular Java programming (Figure 3.1), and

[1] At least, in Java.

invoking a transaction is simply to call a method, as seen in Figure 3.1c, provided that the programmer annotates the method with @Atomic (Figure 3.1b).

```java
class Node {
  int value;
  Node next;

  int getValue() {
    return value;
  }
  void setValue(int v) {
    value = v;
  }
  Node getNext() {
    return next;
  }
  void setNext(Node n) {
    next = 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 3.1: Deuce's programming model (taken from [Val12]).

3.1.2 External Strategy in Deuce

For every STM algorithm, Deuce provides an unique identifier for an object field, a pair $\langle O, f^o \rangle$, where O is the object reference and f^o is the logical offset of field f within O. In the framework, since this pair uniquely identifies a field of an object, it can be used as a key to associate object fields with transactional metadata in any map implementation, and for reading and writing directly to the associated memory location using `sun.misc.Unsafe`.

Custom STM algorithms have to implement a `Context` interface, providing methods for transactional read and write operations, and also for transactions' start and commit. The transactional access methods (`{read,write}Tx`) use the pair $\langle O, f^o \rangle$, passed by parameter, to retrieve the associated metadata from an external mapping table.

```
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 3.2: Example of the modifications made for the external strategy (taken from [Val12]).

To sum up, the Deuce framework performs the following instrumentation of the Java bytecode. Let $C = \{f_1, \ldots, f_i, m_1, \ldots, m_j\}$ be a class C containing fields $\{f_x : 1 \leq x \leq i\}$ and methods $\{m_y : 1 \leq y \leq j\}$. The instrumentation adds a new field f_x^o for each field f_x. The new field's value is the logical offset of f_x in the class. For each method m_y, it adds a modified version m_y^t, in which read and write operations are replaced by the corresponding transactional access methods through the runtime. Additionally, if m_y is annotated with @Atomic, its body is replaced by a retry loop that calls m_y^t in the context of a transaction. In the end we have

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

Figure 3.2 provides an example of the modifications made by the framework (the modifications are highlighted). It shows the corresponding f^o field (Lines 3 and 5, Figure 3.2b) for each field in the original class (Lines 2 and 4, Figure 3.2a). For each method in the original class (Lines 16 and 24, Figure 3.2a), we can see the corresponding m^t version on Lines 20 and 28, Figure 3.2b. As `m1` is annotated with @Atomic (Line 16, Figure 3.2a) the code in its body was replaced with the transactional retry loop (Line 17, Figure 3.2b).

3.1.3 In-Place Strategy in TribuSTM

TribuSTM is a extension to Deuce that enables STM algorithms to be implemented with the in-place strategy without changing the API provided to application programmers. In this framework both the external and in-place strategies are supported, making this extension flexible and fully backwards compatible with already existing implementations of STM algorithms.

When using the external strategy, the runtime provides the STM algorithms with the pair $\langle O, f^o \rangle$ for every transactional access to a field f and then, inside the algorithm, the programmer has to obtain the corresponding metadata, f^m, from the external mapping table. In the in-place strategy, there is no table from which to obtain the metadata given the pair $\langle O, f^o \rangle$. Hence, the runtime provides the STM algorithms with f^m itself. To manage this, the instrumentation injects f^m in C, to be used instead of the pair $\langle O, f^o \rangle$. STM algorithms that follow the in-place strategy have to implement a somewhat different API regarding the transactional access methods. When, the algorithms following the external strategy implement the `Context` interface (whose transactional access methods receive the pair $\langle O, f^o \rangle$ as parameter), the algorithms following the in-place strategy implement the `ContextMetadata` interface in which transactional access methods receive f^m as parameter.

Algorithms that adopt the in-place strategy have to follow some rules regarding their metadata. Their transactional metadata class must extend a common super class, `TxMetadata`, that holds the corresponding pair $\langle O, f^o \rangle$ to read and write the associated field, and the algorithm's implementation must also specify its metadata classes through a @InPlaceMetadata annotation, used by the instrumentation process to be aware of which metadata class to inject.

Figure 3.3[2] shows the differences between the interfaces that the STM algorithms have to implement when following the external or the in-place strategies. In Figure 3.3b, showing the in-place strategy interface, the `{read,write}Tx` methods receive the metadata as a parameter, while in Figure 3.3a, the same methods for the external strategy take the pair $\langle O, f^o \rangle$ and require an additional step to fetch the metadata from the external mapping table. Figure 3.3b also shows us that the instrumentation process is aware that it needs to inject f^m fields of type `MyMetadata`, as specified by the @InPlaceMetadata annotation.

[2] In this example we use `T` where there should be a version of each method for every Java primitive data type `int, long, float, double, short, char, byte, boolean` and `Object`.

```
class MyContext implements Context {
  void writeTx(Object obj, long f_o, T val) {
    // obtain the metadata from the table
    // do whatever...
  }
  T readTx(Object obj, long f_o) {
    // obtain the metadata from the table
    // do whatever...
  }
}
```

(a) External metadata interface.

```
@InPlaceMetadata(class=MyMetadata)
class MyContext implements ContextMetadata {
  void writeTx(TxMetadata f_m, T val) {
    // do whatever...
  }
  T readTx(TxMetadata f_m) {
    // do whatever...
  }
}
```

(b) In-place metadata interface.

Figure 3.3: Algorithm implemented with both interfaces (taken from [Val12]).

To sum up, the TribuSTM framework performs the following instrumentation of the Java bytecode. Let $C = \{f_1, \ldots, f_i, m_1, \ldots, m_j\}$ be a class C containing fields $\{f_x : 1 \leq x \leq i\}$ and methods $\{m_y : 1 \leq y \leq j\}$. The instrumentation adds a new field f_x^m for each field f_x (besides f_x^o). The new field references the transactional metadata object associated with f_x. The methods suffer the same transformations as with the external strategy (Section 3.1.2), with the exception of the constructors and the static initializers, where the creation and initialization of the f_x^m metadata fields' objects are injected. The transactional versions m_y^t are adapted to call the transactional access methods using f_x^m instead of the pair $\langle O, f_x^o \rangle$. In the end we have

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

Figure 3.4 provides an example of the modifications made by the framework (using the same example as in Figure 3.2). Besides the f^o fields, we can see the additional f^m fields (Lines 4 and 7, Figure 3.2b) and their initialization (Lines 23 and 24, Figure 3.2b). In Figure 3.2b, Line 31, for example, the `writeTx` method is called taking the metadata object as a parameter. Lines 2, 15-21, 34-35 and 38-40 (Figure 3.2b) have modifications regarding arrays. For more detailed information about the transformations of arrays-related code done by the framework we direct the reader to [Val12] and [Dia+12; Dia+13].

```java
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.

```java
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 3.4: Example of the modifications made for the in-place strategy (taken from [Val12]).

3.2 Putting the D in TribuDSTM

Usually, DSTM infrastructures are built by extending existing STM frameworks. This is the case with TribuDSTM. TribuSTM was extended with support for distributed objects, means to access them transparently and, validating and committing transactions accessing these objects.

TribuDSTM's architecture (Figure 3.5) is comprised by two main components: (1) the local concurrency control; and (2) the distributed commit and memory consistency layer. Local concurrency control is taken care of by TribuSTM (Section 3.1), that supports different algorithms transparently from the application's perspective. To bring TribuSTM to the distributed setting, the framework stack was extended with a distributed commit and memory consistency layer, the distribution manager (DM). It is responsible for establishing the distributed (possibly shared) memory through mechanisms for object distribution and transaction execution on the distributed memory.

TribuDSTM was designed to be an efficient and modular framework, hence it allows different realizations of the two main components. The local concurrency control allows for modularity since it supports different STM algorithms, but the framework is tied to TribuSTM.

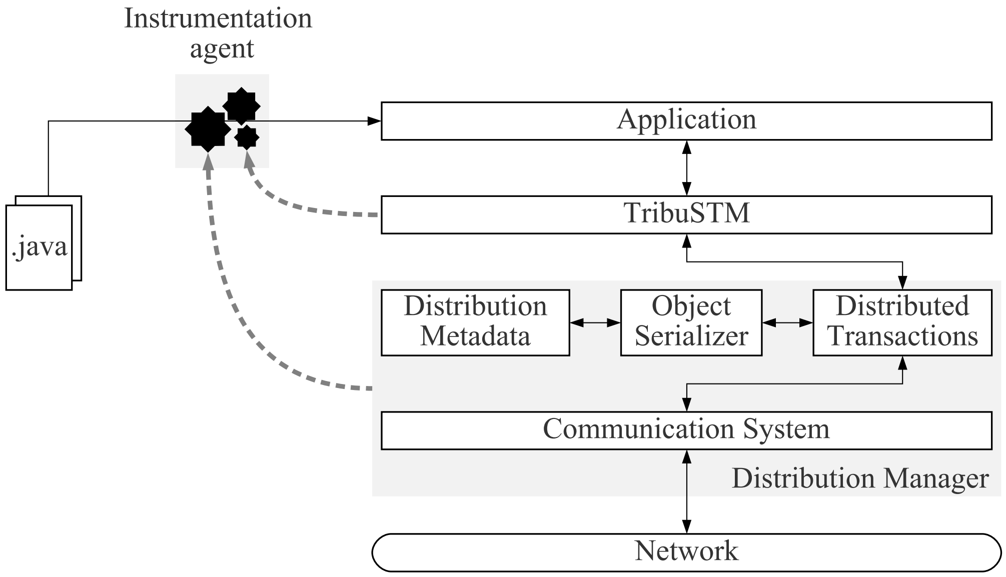


Figure 3.5: TribuDSTM architecture overview (taken from [Val12]).

The interface between the application and TribuSTM remains unchanged, as methods are defined as transactions with the @Atomic annotation, and the instrumentation process redirects read and write accesses to TribuSTM.

To support distributed transactions, the DM should be aware of some events triggered by the application on TribuSTM (e.g., transactional accesses or commit requests). The DM might also need to query or modify the state of the local STM (e.g., to apply updates made by a remote transaction). Distributed objects are achieved with a combination of distribution metadata and the Java serialization support.

As aforementioned, STM algorithms associate metadata to each object's field to manage concurrency. The support for distributed objects also associates metadata with each object to implement a certain distributed memory model. These are two distinct kinds of metadata, thus we will refer to both as *transactional* and *distribution* metadata, respectively, to disambiguate if necessary.

3.2.1 Distributed Transactions

Depending on the used distributed memory model, transaction operations may not be strictly local. A transactional read or write operation (or any other operation) might require communication between the system's participants. In a full replication scenario using a certification-based protocol (Section 2.3.2.3), replicas execute transactions locally and only require synchronization when a transaction attempts to commit. But in a non-replicated scenario, transactional read and write accesses might trigger a request to the owner of the read/written memory location, if such location is not local.

In order to be able to react to certain events from TribuSTM, the DM makes use of the interface presented in Table 3.1, the reflexive API (using the observer design pattern [Gam+94]). The DM registers callbacks for transaction-related events such as transaction start (ONSTART), transactional accesses (ONREAD, ONWRITE), and commit and abort (ONCOMMIT, ONABORT).

Table 3.1: Operations provided by the reflexive API (taken from [Val12]).

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

The DM also needs to retrieve information from the state of the STM, or even to modify it. In a full replication scenario, when the DM initiates the certification protocol it must have (at least) the read and write sets of the transaction, and it updates the STM according to the latter if the validation was successful.

In order to query and/or alter the state of the local STM, the DM makes use of the interface presented in Table 3.2, the actuator API. This interface allows the DM to inspect the state of the local STM and act upon it. The DM can acquire an opaque representation of a transaction's state (CREATESTATE) which can be used to recreate the transaction (RECREATETX). It can also 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).

3.2.2 Distributed Objects

As stated, TribuDSTM achieves distributed objects combining some kind of *distribution metadata* with the memory locations. In a full replication scenario, we want to logically

Table 3.2: Operations provided by the actuator API (taken from [Val12]).

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

identify a memory location across the whole system. If the metadata associated with each memory location includes an unique identifier, a memory location with the same identifier on different replicas represents the same global location. On the other hand, in a fully distributed scenario, metadata might be composed of all the necessary information to execute a remote procedure call (RPC) to the owner of the location. Nevertheless, an unique identifier is always necessary since there has to be a way to distinguish every object.

3.2.2.1 Metadata

The necessary information to be included in the metadata depends on the employed distributed memory model, so the only requirement is that the used metadata class must implement the `DistMetadata` interface.

To associate metadata with objects and to have the means of retrieve it, the framework defines a `DistributedObject` interface comprising a getter and a setter method for the `DistMetadata` object (Table 3.3). Instead of requiring the application programmer to explicitly declare that the application classes implement this interface, the framework's instrumentation agent automatically injects such code in the application.

Note that now, the *transactional metadata* (objects of type `TxMetadata`) also behave as regular distributed objects. Hence, every transactional metadata class extends the

Table 3.3: Operations provided by the distributed object API (taken from [Val12]).

Operation	Description
GETMETADATA(O) : M	Returns the distribution metadata M associated with distributed object O.
SETMETADATA(O, M)	Associates the distribution metadata M with distributed object O.
REPLACEOBJECT(O) : D	Delegates object D to be serialized instead of distributed object O.
RESOLVEOBJECT(O) : D	Delegates distributed object D to be de-serialized instead of object O.

`TxMetadata` class, which in turn implements the `DistributedObject` interface (having an associated distribution metadata).

3.2.2.2 Serialization

To finalize the support for distributed objects, the framework makes use of the Java serialization API, more specifically the `writeReplace` and `readResolve` methods. These methods allow the framework to control how objects are (de)serialized.

The key idea is that distributed objects are serialized in function of its distribution metadata, hence different implementations of the DM might serialize objects differently according to their distributed memory model. The flexibility to (de)serialize objects according to the specific DM implementation is achieved by inserting hooks (REPLACEOBJECT and RESOLVEOBJECT in Table 3.3) in the `writeReplace` and `readResolve` methods, respectively, delegating the process to the DM.

To sum up, the following instrumentation of the Java bytecode is performed by the framework to support distributed objects. Let $C^I = \{\overline{f}, \overline{m}\}$ be a class C implementing the set of interfaces I, and consisting of fields $\overline{f}$ and methods $\overline{m}$. The instrumentation process makes C implement `DistributedObject`, and a new field f^{dm} of type `DistMetadata` is added, along with its getter and setter methods, $getMetadata$ and $setMetadata$. To conclude, the methods $writeReplace$ and $readResolve$ are also added to C. In the end we have

$$C^{I \,\cup\, \texttt{DistributedObject}} = \left\{ \begin{array}{c} \overline{f}, f^{dm}, \overline{m}, \\ getMetadata, setMetadata, writeReplace, readResolve \end{array} \right\}$$

3.2.3 Communication System

Different implementations of the DM can have specific requirements for the communication system (CS). For instance, the non-voting certification protocol (Section 2.3.2.3), running on a fully replicated memory, requires a GCS with support for a `TO-broadcast` primitive. On the other hand, in a purely distributed memory (i.e., every object has one and only one owner) with a pessimistic approach, i.e., lock objects upon access, is only required point-to-point communication.

The framework also provides a clean API between the distributed transactions (DT) component and the CS (Figure 3.5), being easy to plug in different GCSs. Those interfaces are presented in Tables 3.4 and 3.5. The latter one presents to what deliveries the DT can subscribe in order to be notified of incoming messages.

Table 3.4: Interface provided by the GCS to the DT (taken from [Val12]).

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 3.5: Interface provided by the DT to the GCS (taken from [Val12]).

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.

3.2.4 Bootstrapping

The execution model in a distributed environment can execute the same program at all replicas or execute possibly different programs at each replica, being that all of these programs manipulate (some of) the same objects. The only way to distinguish two objects is for both to have some differentiating characteristic. The most straight-forward way of achieving this is for each object to have an unique identifier associated with its distribution metadata (as stated in Section 3.2.2). But those identifiers are assigned randomly. If every object is assigned a different identifier, the managed objects are disjoint. This poses a bootstrap problem.

Consider, for example, the Red-Black Tree microbenchmark. This microbenchmark manipulates a red-black tree, doing three different operations on the tree: (1) insertions; (2) removals; and (3) searches. When the microbenchmark is distributedly executed at each replica, it must handle the same tree in every replica. This means that the tree reference distribution metadata has to have the same identifier in all replicas.

This problem is addressed by TribuDSTM using the @Bootstrap annotation. It is used to denote that a number of fields are semantically the same, hence their value is always the same (as the distribution metadata identifier).

This annotation targets classes' fields and it takes a parameter `id` whose value serves as a seed in the identifier generation. Using the same seed deterministically generates the same identifier. This gives the practical effect of that field 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).

Figure 3.6 shows an example using the annotation. The `tree` field is semantically the same in all replicas and it is initialized in the `createTree` transaction. In all replicas, the `tree` field metadata's identifier is generated using the value given as parameter, hence the identifier is the same in every replica.

```
class Benchmark {
  @Bootstrap(id = 1)
  private static RBTree tree;

  @Atomic
  public void createTree() {
    if (tree == null)
      tree = new RBTree();
  }

  public static void main(String[] args) {
    ...
    createTree();
    // from this point on, tree references the same object in every replica
    ...
  }
}
```

Figure 3.6: Bootstrapping with the @Bootstrap annotation (taken from [Val12]).

3.3 Summary

We presented TribuSTM, an extension to the Deuce Java STM framework. Deuce only supports the implementation of STM algorithms following the external placement strategy for transactional metadata. TribuSTM extends Deuce by supporting also the in-place metadata placement strategy without any changes to the API the framework provides to the applications.

We also described how TribuSTM was extended to support DSTM. In Section 3.2, we presented the framework's architecture, its layers and the defined APIs between them. We also detailed how the local STM layer was extended to support distributed transactions and distributed objects.

4

Supporting Partial Replication with TribuDSTM

In this chapter we present the extensions done to TribuDSTM (Chapter 3) in order to support the partial data replication distributed memory model. In Section 4.2, we describe the extensions added to the programming model of TribuDSTM. We provide an overview of the extended framework and detail the main modifications in Section 4.3. In Section 4.4, we also describe our implementation of a partially replicated STM using the proposed framework.

4.1 Partial Replication Summarized

This section summarizes the concept of partial data replication introduced in Section 2.3.3.

Data distribution maximizes the system's total storage capacity, but provides no fault tolerance support. The failure of one or more nodes means inevitable data loss. On the other end of the spectrum, full data replication maximizes the possibility of data survival while limiting the system's total storage capacity to the capacity of the node with fewer resources.

In this thesis we consider an intermediate solution – partial data replication [Alo97] – where each node only replicates a subset of the system's data. This strategy tries to combine the advantages of both data distribution and full replication while diminishing their disadvantages, thereby allowing the system to make good use of its storage capacity while ensuring some level of fault tolerance.

The key idea is that, in a system with these characteristics, the processing of a transaction only involves a subset of the system's nodes. Given that, since different transactions may access different data items located at distinct nodes, their processing will likely involve different sets of nodes, allowing the latter to process independent parts of the workload in parallel. This approach can improve the scalability of replicated systems since updates only need to be applied to a subset of the system's nodes. This leads to a eventually smaller synchronization cost [Cou+05].

The use of this strategy implies the division of the system nodes into *groups* and the replication of the data among these groups as to ensure that, in the same group, every node replicate the same data items. However, this does not forbid the replication of the same data item in multiple groups nor requires that groups have to be comprised by disjoint sets of nodes.

4.2 Programming Model

Partial replication as a concept originates from the database system's research field [Alo97] where it has been used in the context of large scale systems. Given so, it is natural that they represent a source of inspiration when trying to incorporate this concept in a DSTM system for a general-purpose programming language. However, these contexts have distinct properties. When partial replication is applied to databases, the content of the tables stored in the database is simply partitioned by rows and spread among the system's nodes. More or less the same happens with partial replicated key/value stores. But the concept of partial replication, such as it is used by these, needs to be adapted to this new context of a general-purpose programming language. What data items should be partially replicated? Can any object be partially replicated? These and other issues will be addressed next.

In the JVM, the *memory heap* can be seen a directed graph, where each node represents an object which, in turn, may have references to other objects, as Figure 4.1 depicts. The graph's leaf nodes represent primitive data types and the remainder represent references to other objects or primitive data types. In Figure 4.1, node B can be seen as an object comprising three fields, where two are primitive data types and the other (node E) is a reference to another object which, in turn, has two fields (also two primitive data types). Nodes A and H are referencing the same object (aliasing).

Following the spirit of TribuDSTM [Val12], we decided to keep our framework's public API as less intrusive as possible for programmers. Hence, we added just one new annotation, @Partial. Figure 4.2 illustrates the usage of this annotation for a simple example of a linked list. By adding the @Partial annotation to the field of a class, the programmer is declaring that everything that is downstream of this reference (in the object graph) should be replicated in a single group, i.e., it should be partially replicated. By

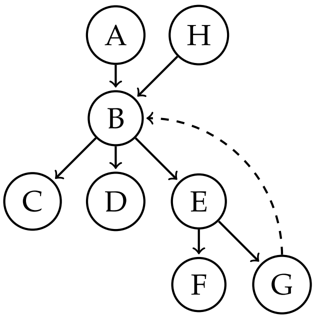


Figure 4.1: Example of a Java heap graph.

default, everything that is not explicitly annotated will be fully replicated. So, we provide a flexible programming model which allows the programmer to adjust the level of distribution/replication to the specificities of each application.

```
class Node<T> {
  private Node next;
  @Partial
  private T value;
  ...
}
```

(a) Class `Node`.

```
class List<T> {
  protected Node<T> head;
  public List() {
    this.head = new Node<T>(null, null);
  }
  ...
}
```

(b) Classe `List`.

Figure 4.2: Example using the @Partial annotation.

However, the use of this annotation requires some caution. Given that data is partially replicated, data access is not uniform, as some may require communication with a remote node. This is the inevitable cost of distribution. In the example of Figure 4.2, if the @Partial was placed in the `next` variable, the iteration over the elements of the list would entail a, possibly remote, read operation for each iterated element, even if we do not inspect its value. This would seriously compromise the application's performance.

The usage of the @Partial annotation is somewhat inspired in distributed databases with partial data replication. In these, data is stored in tables and those tables *exist* in every node. The only thing that is partially replicated is the information stored in the tables. The same happens with key-value data stores, where the structure/"container" that organizes the data can also be found in every node, while the actual information is partially replicated. In a general-purpose programming language, the idea is somewhat the same: full replication of the data structures and partial replication of the hard data they store. In Figure 4.2, the data structure is the linked list, with its fully replicated nodes (`Node` class). The hard data is represented by the `value` variable that is partially replicated. This way, the number of remote read operations resultant from the list's traversal is limited by which values we really want to inspect.

Still in the scope of the linked list example in Figure 4.2, consider a concrete distribution of the data, such as the one depicted in Figure 4.3. We can see two groups. The grey squares can be seen as the nodes of the linked list and the circles as the value stored at each node. The grey circles denote replicated data items and the white ones denote data items that are not replicated. We can observe that the structure of the linked list, i.e., the nodes, is fully replicated and the value in each node is partially replicated, since it is replicated in only one group.

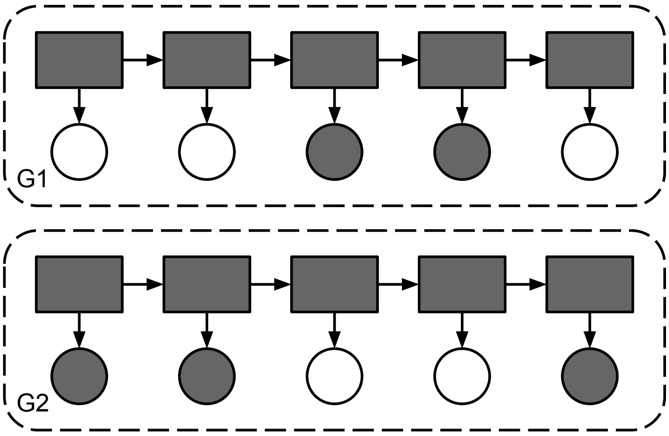


Figure 4.3: Example of a partially replicated linked list.

Following the usage explained above, typically the application of the distribution, i.e., the application of the @Partial annotation, is made in terms of data structures. These, usually, are implemented once, by someone who knows the system, and can later be used transparently by the common user.

4.2.1 Limitations

One limitation of this model is the prohibition of the creation of edges in the heap graph which, in turn, create cycles for nodes upstream of the @Partial annotation. In the example of Figure 4.1, if the field represented by node E was annotated with @Partial, the dashed edge could not exist. By annotating node E, the programmer declared that the objects represented by nodes E, F and G will be partially replicated and that, in contrast, everything that is upstream of node E will be fully replicated. Existing an edge from node G to node B intuitively means that node B should be partial and fully replicated at the same time. On the other hand, if the @Partial annotation was placed in node B, the dashed line could exist, since everything that is downstream of the annotated node would be partially replicated without any problem. This is not a severe limitation.

We verify the violation of this restriction at runtime. The system will throw an exception if the application tries to write an object that is replicated in a group to a memory location associated with a transactional metadata that is replicated in a different group. We will revisit this limitation and explain it in a section ahead, in the light of more detailed information. We will refer to this as the *checkGroup* restriction.

4.3 Runtime System Extensions to TribuDSTM

In Chapter 3, we presented TribuDSTM's architecture and detailed its various components. This section details the extensions performed upon the TribuDSTM framework to support partial replication. Their scope is twofold, targeting both API design and implementation, with a focus essentially directed at the communication layer, and at group and data partitioning.

4.3.1 Communication System

The management of the distributed (possibly shared) memory in TribuDSTM is concentrated on the DM component. Different management strategies, i.e., different DM implementations, may impose different requirements on the lower layers of the software stack, namely the one responsible for the communication primitives. While, in a full replication environment, the majority of the protocols require `TO-broadcast` and `R-broadcast` primitives, in a partial replication scenario, protocols usually need point-to-point communication, or even an `A-multicast` primitive.

To this extent, we extended the APIs presented in Section 3.2.3 (Tables 3.4 and 3.5) to accommodate the new communication primitives usually required by partial replication protocols.

We have added a total of three primitives, as seen in the lower half of Table 4.1. The multicast primitives (AMCAST and RMCAST) send the message passed as a parameter to a group g. This group is conceptually defined by the nodes to which the message should be sent. Parameter g is an opaque representation of a group that conceptually includes the location of the nodes that comprise it. This allows the framework to handle different GCSs transparently. The specific representation of a group used by the framework will be presented in a section ahead.

Table 4.1: Extended 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.
AMCAST(m, g)	A-multicasts message m to group g.
RUCAST(m, dst)	Reliably unicasts message m to node dst.
RMCAST(m, g)	Reliably multicasts message m to group g.

Table 4.2 presents to what deliveries the DT can subscribe, to be notified of incoming messages. We extended the API with the last three operations that allow the DT to be notified of incoming messages sent with the primitives added to the previous table.

It should be noted that from the three GCS implementations considered by Vale [Val12] (JGroups [Ban98], Appia [Mir+01] and Spread [Ami+00]) the only one providing an atomic

Table 4.2: Extended 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.
ONAMDELIVER(m, s)	Notifies of the delivery of message m which has been A-multicasted by sender s.
ONRUDELIVER(m, s)	Notifies of the delivery of message m which has been reliably unicasted by sender s.
ONRMDELIVER(m, s)	Notifies of the delivery of message m which has been reliably multicasted by sender s.

multicast primitive (as presented in Section 2.3.1.2) is JGroups. This primitive was implemented in [Rui11].

4.3.2 Groups

It has been stated multiple times that partial replication requires the grouping of the system's nodes. These groups determine the replication factor for each data item. If a group is comprised by four nodes, then the data items replicated by this group are replicated in those four nodes and are said to have a replication factor of four.

TribuDSTM was compromised with the full replication model and hence did not featured the notion of group. Accordingly, such support has to be added. Table 4.3 presents the implemented group interface, where g denotes a group identifier, a a node identifier and A a list of node identifiers.

Our concrete implementation of a group used by the framework is implemented as a set of addresses. So, when operation GETMEMBERS is invoked it returns a list with the addresses of all the elements that belong to the specified group. The special group ALL denotes the group that represents all the nodes in the system.

In our implementation, we opted for a simple scheme that replicates each data item *only* in one group and groups are comprised by *disjoint* sets of nodes. Although we implemented it this way, it can be easily extended. For a data item to be replicated by multiple groups we just need to replace the association of a single group to the item by a list of groups (in the distribution metadata). For a group to be comprised of non-disjoint sets of nodes, we just need to add the same node address in multiple groups, thus making a node belong to different groups (this is related with the group partitioning).

4.3.2.1 Static Partitioning vs. Dynamic Partitioning

We define static partitioning as being a partitioning process determined in advance and feed to the system at deployment time. Usually this partitioning falls on the user which

Table 4.3: Operations provided by the group API.

Operation	Description
GETMEMBERS(g) : A	Returns a list A with the elements of group g.
CONTAINS(g, a) : *bool*	Returns true if a is an element of group g, false otherwise.
ADD(g, a)	Adds element a to group g.
REMOVE(g, a)	Removes element a from group g.
SETGROUP(g, A)	Sets group g to be comprised by elements A.
SIZE(g) : *int*	Returns the number of elements in group g.
EQUALS(g, g') : *bool*	Returns true if group g is equal to group g', false otherwise.
UNION(g, g') : g''	Returns a new group g'' comprised by the union of group g with group g'.
ISLOCAL(g) : *bool*	Returns true if the local node is an element of group g, false otherwise.
ISALL(g) : *bool*	Returns true if group g is the group `ALL`, false otherwise.

must provide the specific grouping to the system (e.g., using some kind of configuration file). On the other hand, we define dynamic partitioning as being a partitioning process that is performed by the system using some implemented decision process.

Static partitioning greatly restricts extensibility, since it would be impossible to implement a more complex/"smart" and automatic partition strategy. Moreover it burdens the user with writing of a configuration file for every target system/cluster. So, we opted for dynamic partitioning, requiring the implementation of partitioning strategies (and allowing the user to choose from among the implemented strategies, by parametrization, when executing the target program).

An example of what we classify as a smarter strategy is, for example, a strategy where the partitioner measures the network latency between nodes and tries to create groups with the lowest possible latencies (it just has to be a deterministic process).

Note that our decision does not prevent the use of static partitioning. For that purpose, it is only necessary to establish the reading of the static information as the dynamic decision process.

4.3.2.2 Our Approach to Dynamic Partitioning

The group division strategy may have a substantial impact on the overall system's performance. Consider, for instance, a system composed by multiple clusters. A group comprising nodes of different clusters could increase the validation time of a transaction in several orders of magnitude.

Given this impact, the assignment of nodes to the respective group has to follow some logic/strategy. Following the modular design of the TribuDSTM framework, these decision strategies must be encapsulated in some component. To this end, we created a new sub-component that is responsible for this decision, the group partitioner (GP). With the knowledge of the nodes that exist in the system and the number of groups that should exist, the GP clusters the nodes into groups according to the chosen strategy. Both the replication factor and the partitioning strategy are parametrizable. Therefore, the user can configure this component by supplying concrete values to these parameters. No code rewriting is required whatsoever to switch between partitioning strategies.

Given that the decision of how many groups should be deployed falls of the user, if he wants to know the number of groups created by the system, the following calculation can be done:

$$\text{Number of groups} = \frac{\text{Number of nodes}}{\text{Replication factor}}$$

For example, if we have a system with 50 nodes and want our objects to have a replication factor of 2, we need to have $^{50}/_{2} = 25$ groups (each group will comprise 2 nodes).

At the moment, the framework only has implemented two simple strategies:

- Random partitioning; and
- Round robin partitioning.

Both strategies have self-explanatory designations. Note that the random partitioning strategy may (most certainly will) create groups with different number of nodes, thus making data items to have different replication factors.

Besides of embedding the partitioning logic, the GP provides the interface described in Table 4.4. This API allows the DM to have access to group related information.

Table 4.4: Interface provided by the GP to the DM.

Operation	Description
GETGROUPS() : G	Returns a list G with all the existing groups in the system.
GETLOCALGROUP() : g	Returns the group g where the local replica is contained.

In order to reduce the communication overhead, we want for a node to be able to locally infer the composition of any of the system's groups. For that purpose, the grouping process has to be deterministic. Thus, with the same input to the GP, all nodes reach the same outcome.

Our implementation of the random partitioning strategy achieves this by using the same seed for the pseudo-random number generator. The implementation of the round robin strategy resorts to an ordered list of the system's nodes for the partitioning process.

4.3.3 Data Partitioning

The creation of new objects raises another question. Where do the partially replicated objects should be replicated? In our framework, partially replicated objects are replicated within a single group. Accordingly, in which group should a new object be replicated? To handle this decision we designed a new sub-component that encapsulates such decision making algorithm, the data partitioner (DP). Besides of implementing the data partitioning logic, the DP also provides the interface described in Table 4.5.

Table 4.5: Interface provided by the DP to the DM.

Operation	Description
PUBLISHTO(O) : g	Returns the group g where object O should be replicated.

At the moment the framework only has implemented three simple strategies:

- Random partitioning;
- Round robin partitioning; and
- Local partitioning.

The first two can maintain the amount of data balanced between the groups, due to their random/circular nature.

The last one differs from the remainder by establishing that every newly created object must be replicated by the group of the node where they were created at, thus establishing affinity between the node that executed the transaction creating the object in question and the object's final location. This strategy achieves the same load balancing as the first two only if the workload is divided relatively equally by all groups.

More sophisticated strategies could, for instance, try to enforce explicit load-balancing, regarding objects' distribution among the groups. The system could try to keep track of the object count in each group and evenly distribute the objects among them taking that information into account.

4.4 Implementing a Partially Replicated STM

In the context of this thesis we implemented, using the extended framework, a replicated STM which targets an environment where objects may be partially or fully replicated among the participants of the distributed system. Transactions are committed across the system using SCORe [Pel+12a], described in Section 2.3.3.3. This replicated STM implementation is a concrete realization of the DM component of our architecture (Figure 3.5).

4.4.1 Distributed Objects

As stated in Section 3.2.2, TribuDSTM allows the implementation of different distributed memory models by combining distribution metadata with the memory locations (objects, in our case) and also with the help of the Java serialization API.

4.4.1.1 Distribution Metadata

The fully replicated STM by Vale [Val12], used distribution metadata that only included an universally unique identifier (UUID) [Uui] as it was enough to logically identify a memory location across the whole system.

In a partial replication environment, an unique identifier is not sufficient. Since objects are partially replicated, i.e., each object is replicated in just one group, a node may attempt to perform a read operation upon an object for which it does not hold a local copy. In such case, the node must be able to identify the object as being remote and to obtain the necessary information to request a copy from a node that replicates it.

In our implementation, distribution metadata is comprised by the information depicted in Figure 4.4. The inherited UUID, still needed to identify objects across the system, is complemented with additional information, namely the group where the object is replicated and a flag that indicates if the object has already been published.

```
class PartialRepMetadata implements DistMetadata {
  UUID id;
  Group group;
  Group partialGroup;
  boolean isPublished;
  ...
}
```

Figure 4.4: Content of the partial replication distribution metadata.

The remaining `partialGroup` variable is necessary to handle the particular case of metadata objects associated with variables annotated with @Partial, i.e., when crossing the boundary from full to partial replication.

Recall that, in Section 3.1.3, with the in-place metadata placement strategy, the runtime provides the STM algorithms with a transactional metadata upon any transactional access. Hence, in a transactional read (write) operation the algorithm reads (writes) from (to) a memory location associated with the provided transactional metadata. In the specific case of a transactional read, the desired object might not be replicated locally. Under such circumstances, the STM algorithm requests the DM to obtain a copy of the aforementioned object from some node that replicates it. This case is detected by inspecting the value of the `partialGroup` variable. Since the STM algorithm only has access to the distribution metadata of the provided transactional metadata, the `partialGroup` variable informs in which group the object associated with the transactional metadata is replicated. With this information, the DM is able to request the desired object from the nodes in that group. Note that the `partialGroup` variable only matters in this case, as with all other (regular) distributed objects `partialGroup` = `group`. In fact, `TxMetadata` objects only have `partialGroup` ≠ `group` when they are associated with a field that is annotated with @Partial, as seen in Figure 4.5.

This figure depicts an example of a simple object graph with three objects, O_0, O_1 and O_2. Each object has a field (f_0, f_1 and f_2, respectively) and the field's corresponding transactional metadata (*tm*-f_0, *tm*-f_1 and *tm*-f_2, respectively). Field f_0 is a reference to object O_1 and f_1 is a reference to object O_2. Field f_1 is also annotated with @Partial, i.e., everything downstream of this reference is replicated in only one group. The dashed boxes represent distribution metadata (the three in the top are the distribution metadata

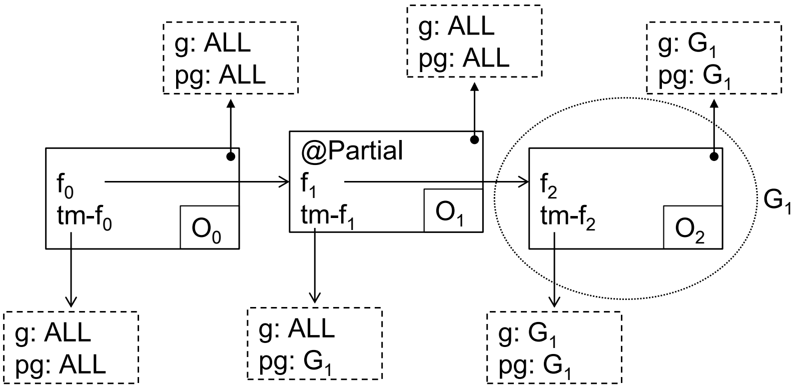


Figure 4.5: Distribution metadata with the `partialGroup` and `group` variables.

of the three regular distributed objects and the three in the bottom are the distribution metadata of the fields' transactional metadata). In the distribution metadata of *tm*-f_0 we see that `partialGroup` = `group` (ALL = ALL) as it is a fully replicated field. We encounter the same with *tm*-f_2, but in a partial replication scope this is somewhat different ($\mathtt{G}_1 = \mathtt{G}_1$) since it belongs to a partially replicated object (O_2). With *tm*-f_1, since it is annotated with @Partial, we see that `partialGroup` ≠ `group`, as `group` informs of which group replicates *tm*-f_1 itself, and `partialGroup` informs of which group replicates the object referenced by the field associated with *tm*-f_1. In the distribution metadata of the regular distributed objects, we can just look at the value in `group`, since both variables always hold the same value.

The specific interface provided by the distribution metadata used for partial replication is described in Table 4.6. It provides getters and setters for both "groups" (GET{GROUP, PARTIALGROUP} and SET{GROUP, PARTIALGROUP}) and operations for publishing the object.

Table 4.6: Operations provided by the partial replication distribution medatata API.

Operation	Description
GETGROUP(M) : g	Returns the group g associated with distribution metadata M.
SETGROUP(M, g)	Assigns group g to distribution metadata M.
GETPARTIALGROUP(M) : g	Returns the *partial* group g associated with distribution metadata M.
SETPARTIALGROUP(M, g)	Assigns the *partial* group g to distribution metadata M.
EQUALS(M, M') : *bool*	Returns true if distribution metadata M is equal to distribution metadata M', false otherwise.
ISPUBLISHED(M) : *bool*	Returns true if the distribution metadata M was already published to the network.
PUBLISH(M)	Sets the distribution metadata M as being published to the network.

4.4.1.2 Serialization

Before the transaction state is sent through the network, its contents are serialized. The devised partial replication serialization strategy guarantees that each replica manipulates its own representatives of the replicated objects.

The serialization algorithm is presented in Figure 4.6 and works as follows. When an object is being serialized, it invokes the function OBJECTREPLACE. As metadata is assigned during the object's creation, the algorithm resorts to the ISPUBLISHED operation to know if the object as already been sent through the network or not. If ISPUBLISHED(oid) returns true then the object is said to be *published*, otherwise it is deemed as *private*.

Partial replication inserts the need for the serialization algorithm to be aware if its invocation results from a remote read. Accordingly, it must resort to GETSERIALIZATION-CONTEX (Line 4) to retrieve the serialization context provided by the DT. The invocation of GETSERIALIZATIONCONTEXT(O) returns true if the algorithm is in the presence of a read context for object O, false otherwise.

When in the presence of a read context (Line 5), the original object has always to be serialized (Line 6), since if we are responding to a read request it is because the requester does not replicate the object locally.

If the object is already published (Line 7), its UUID oid is nominated to be serialized in its place (Line 8). Type consistency is maintained because the deserialization of oid when delivered on a replica will return the local representative on that replica, i.e., the object O such that GETMETADATA(O) $= oid$ (Line 29)[1]. This is an optimization, since if the replicas (that compose the group where the object is replicated) already possess a local representative of the object being serialized it would be a waste of resources to send the whole object graph through the network.

If the object is private (Line 9), it is set to be published (Line 11) and the original object is designated to be serialized (Line 14). If the local replica belongs to the group where the object should be replicated (Line 12), the object is saved in a map (named memory in Figure 4.6) associating distribution metadata to (published) objects (Line 13).

When an object is being deserialized, it invokes OBJECTRESOLVE. The deserialization process is different if the local replica belongs to the group where the object should be replicated (Line 18). In this case, we check if the object's metadata is associated with something in the map (Line 19), and if there is an object in the map for that metadata we return it (Lines 20 and 21), as it is the local representative on the local replica. If not, it means that it is the first time the local replica receives this object, so we save it in the map and return the original object itself (Lines 22-24).

If the object should not be replicated in the local replica, i.e., the local replica does not belong to the group where the object is replicated, the original object is simply returned. (Lines 25-27). In Line 26, although the local replica does not belong to the group

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

where the object should be replicated, we still save the object in the map. This together with some more nuances are part of an optimization that we will refer to as the *readOpt* optimization. We will explain and clarify this optimization in the next section.

```
 1: memory ← $\emptyset$

 2: function OBJECTREPLACE($O$)
 3:     oid ← GETMETADATA($O$)
 4:     isRead ← GETSERIALIZATIONCONTEXT($O$)

 5:     if isRead then
 6:         return $O$
 7:     else if ISPUBLISHED(oid) then
 8:         return oid
 9:     else                          ▷ not ISPUBLISHED(oid)
10:         group ← GETGROUP(oid)
11:         PUBLISH(oid)
12:         if ISLOCAL(group) then
13:             memory ← memory[oid ↦ $O$]
14:         return $O$

15: function OBJECTRESOLVE($O$)
16:     oid ← GETMETADATA($O$)
17:     group ← GETGROUP(oid)
18:     if ISLOCAL(group) then
19:         obj ← memory(oid)
20:         if $\exists$ obj then
21:             return obj
22:         else                                  ▷ $\nexists$ obj
23:             memory ← memory[oid ↦ $O$]
24:             return $O$
25:     else                     ▷ not ISLOCAL(group)
26:         memory ← memory[oid ↦ $O$]
27:         return $O$
```

(a) Object Serializer component.

```
28: function READRESOLVE(oid)
29:     return memory(oid)
```

(b) Distributed Metadata component.

Figure 4.6: Pseudo-code of the replicated objects' serialization algorithm.

After a careful scrutiny of this algorithm one question may arise. What are the implications of using the map in Line 1? By keeping references to both the distribution metadata and the objects themselves, it prevents them from being garbage-collected when they become unreachable by the application, thus causing memory leaks.

The implemented solution, taken from Vale [Val12], is twofold. It 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 distribution 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 becomes unreachable by the application it can be garbage-collected from the map, which in turn makes the associated key unreachable, thus also being garbage-collected.

4.4.1.3 Programming Model Limitations Revisited

In Section 4.2.1, we explained some of the limitations of the provided programming model. We introduced a restriction, that we called the *checkGroup* restriction, which could not be explained adequately without more information about how we implemented distributed objects in the framework.

Recalling, the framework will throw an exception at runtime if the application tries to write an object that is replicated in a group to a memory location associated with a transactional metadata that is replicated in a different group. One could think of two simple ways to surpass this: (1) we could merge the two groups, creating a new group comprised by the nodes in both groups; and (2) we could change the group of one of the objects so that both objects are replicated in the same group. Both solutions would require the system to notify every node of the intervening groups to modify the corresponding objects' metadata accordingly. We think that is would impose an unsatisfying overhead (and in case (1) the partially replicated objects could degenerate in fully replicated).

Figure 4.7 shows the verification that the STM algorithm makes in every (transactional) write operation done to an object[2], i.e., when an application makes a write access inside a transaction.

```
procedure CheckGroupRestrictions($O$, $m$)
    txOid ← GetMetadata($m$)
    objOid ← GetMetadata($O$)
    txPartialGroup ← GetPartialGroup(txOid)
    objGroup ← GetGroup(objOid)
    if isPublished(txOid) and isPublished(objOid) then
        if not equals(txPartialGroup, objGroup) then
            throw Exception
    else if isPublished(txOid) and not isPublished(objOid) then
        objPartialGroup ← GetPartialGroup(objOid)
        txPartialGroupMembers ← GetMembers(txPartialGroup)
        setGroup(objGroup, txPartialGroupMembers)
        setGroup(objPartialGroup, txPartialGroupMembers)
    else if not isPublished(txOid) and isPublished(objOid) then
        objGroupMembers ← GetMembers(objGroup)
        setGroup(txPartialGroup, objGroupMembers)
    else                                        ▷ not isPublished(txOid) and not isPublished(objOid)
        objPartialGroup ← GetPartialGroup(objOid)
        txPartialGroupMembers ← GetMembers(txPartialGroup)
        setGroup(objGroup, txPartialGroupMembers)
        setGroup(objPartialGroup, txPartialGroupMembers)
```

Figure 4.7: Pseudo-code for checking group restrictions in every (transactional) write operation.

[2] This excludes all the primitive data types.

Very briefly, the procedure receives two parameters: the written object O and the transactional metadata m associated with the written memory location (recall that transactional metadata objects are also distributed objects). If both objects are already published (Line 6) and the groups where they are replicated are not the same (Line 7), we throw a runtime exception since we do not allow this case. Otherwise, we have three possible cases: (1) the transactional metadata is published and the object is not (Line 9); (2) the object is published and the transactional metadata is not (Line 14); and (3) both objects are not published (Line 17). In all these three cases, we can modify the distribution metadata of one of the objects since at least one of them is not published. In case (1), since the transactional metadata is already published, we have to modify the group of the written object to conform to the group of the transactional metadata. In case (2) we find the opposite, where is the written object that is already published. In this case we modify the group of the transational metadata to conform to the group of the written object. In case (3), both the transactional metadata and the written object are not published, so we decided to take the same approach as in case (1).

4.4.2 Distributed Transactions

In full replication environments, certification-based protocols show up as an interesting option by no requiring synchronization between nodes during the execution of transactions. These protocols are simple and easy to implement, and above all, they require few communication steps.

However, it is not easy to support partial replication from full replication protocols. Certification-based protocols rely in the TOB communication primitive to ensure that all nodes receive transactions in the same order, functioning as a finite automaton. In the case of partial replication this solution does not work. For example, consider a system comprised by three nodes ($N1$, $N2$ and $N3$). Node $N1$ replicates objects B and C, $N2$ replicates A and B, and $N3$ replicates A and C. Consider further that there are two transactions $T1$ and $T2$ being concurrently executed in $N1$ and $N2$, respectively. Transaction $T1$ modifies B and $T2$ modifies A and B. In the end, $N1$ and $N2$ certify both transactions, but $N3$ (since it only replicates A and C) only certifies $T2$. If the total delivering order of the transactions is $T1$ and $T2$, $N1$ and $N2$ validate $T1$ and abort $T2$, but $N3$ validates $T2$, modifying the value of A, thus making the data inconsistent between nodes.

Thus, nodes cannot validate transactions alone, because they do not have all the necessary information. They have to agree among them which transactions are validated and which are aborted. There must be a vote. Following this thought, several protocols for partial replication have been proposed, based in 2PC. First, by distributed databases [Sch+10; Ser+07; Sou+01] and more recently in the context of TM [Pel+12a; Pel+12b].

In Section 2.3.3.3, we presented the two proposed protocols for partial replication in the context of TM. Of these, we decided to implement SCORe as it seems to scale better (with its logical clocks vs. the vector clocks of GMU) and to be easier to implement.

4.4.2.1 SCORe

SCORe [Pel+12a] is a scalable one-copy serializable partial replication protocol. It is a genuine protocol, thus it ensures that only the nodes that maintain data accessed by a transaction are involved in its processing. It also guarantees that read operations always access consistent snapshots, thanks to a one-copy serializable multi-version scheme, which never aborts read-only transactions and spares them from any (distributed) validation. The protocol implements a distributed multi-version scheme coupled with a distributed logical clock synchronization scheme that only requires the exchange of a scalar clock value.

To commit transactions, SCORe relies on a genuine atomic commit protocol that can be seen as the fusion between the 2PC algorithm and the Skeen's total order multicast [Déf+04]. 2PC is used to validate update transactions and the Skeen's total order multicast is used to serialize transactions.

SCORe maintains two scalar timestamps per node, namely $commitId$ and $nextId$. The former maintains the timestamp that was attributed to the last update transaction to have committed on that node. The latter keeps track of the next timestamp that the node will propose when it will receive a commit request for a transaction that accessed some of the data that it maintains. Snapshot visibility for transactions is achieved by associating to each one a scalar timestamp called snapshot identifier (sid), that is established upon a transaction's first read operation.

The pseudo-code of the SCORe protocol is reported in Figures 4.8, 4.9 and 4.10, with regard to the already defined APIs. Figure 4.8 describes the invocation of the commit operation in TribuSTM, Figure 4.9 describes the distributed commit phase of the protocol, and Figure 4.10 describes the read operations in the protocol. The protocol works as follows.

The DT maintains, for local transactions, a mapping between the transaction's identifier and the transaction itself (Line 8), and for remote transactions, a mapping between the transaction's identifier and the transaction's state (Line 9). It also maintains two priority queues with tuples $\langle$transaction's identifier $\times$ transaction's sid$\rangle$, ordered by transaction's sid, and a set of transactions' states (Line 12).

Commit. When a node N_i requests to commit a transaction T, COMMIT is invoked on TribuSTM (Line 2), which delegates to the DT by issuing ONCOMMIT.

When ONCOMMIT is issued, on the DT, we first add the transaction to the map of local transactions (Line 14). To guarantee genuineness, SCORe involves in the commit phase of T only the nodes that maintain a copy of the data items that T accessed, so we proceed by obtaining the group of nodes that are involved in this commit (i.e., the nodes

```
committed ← false

function COMMIT($T$)
    ONCOMMIT($T$)
    return committed

procedure COMMITPROCESSED($T$, $c$)
    committed ← $c$
    $T$ was processed
```

Figure 4.8: Pseudo-code for commit invocation in TribuSTM.

that comprise the groups where the data items in the transaction's write and read sets are replicated, or simply the nodes belonging to $Nodes(T.\text{rs} \cup T.\text{ws})$), and also the state of the transaction which is subsequently reliably multicast to that group (Lines 15-17). This represents the sending of the *prepare* message of 2PC. At this point, this thread of execution (the application thread) waits for the transaction processing to finish.

Prepare Message. Upon the delivery of the *prepare* message (reliably multicast), the ONRMDELIVER callback is invoked by a thread from the GCS (Line 19). Depending on whether the received transaction state is local or remote, the DT either obtains the local transaction (Line 21) or recreates the remote transaction (Line 24). If the transaction is a remote one, we also add it to the map of remote transactions (Line 23). Next, we verify whether the transaction can be serialized after every transaction that has locally committed so far. To this end, we validate the transaction (Line 25). In this validation we attempt to acquire exclusive and shared locks for the data items in the transaction's write and read sets, respectively, that are locally maintained. We also validate the read set, verifying that none of the read data items has been overwritten by a more recently committed transaction.

Vote Message. If the transaction passes the validation (Line 27), the protocol exploits the *vote* message of 2PC to overlap a distributed agreement scheme that aims to establish the final serialization order for the transaction. We increment the $nextId$ timestamp (Line 28), insert the pair $\langle T.\text{id}, nextId \rangle$ in `pendQ`, a queue of pending committing transactions (Line 29) and send back to the transaction coordinator the value of $nextId$ in piggyback to the *vote* message (Line 30).

The coordinator gathers the *vote* messages reliably unicasted by the participants (Line 32), keeping the maximum of the proposed timestamps (Lines 37 and 38), and multicasts back a *decide* message with the transaction's final commit timestamp (Line 60).

Decide Message. Upon the reception of the *decide* message (Line 41), we have two possibilities: (1) the outcome is positive, meaning this transaction may be committed; and (2) the outcome is negative, meaning this transaction must be aborted. In case (1), with a positive outcome, we buffer the transaction in a queue of stable transactions, namely `stableQ`, waiting to be committed (Line 44). This is required because since SCORe wants to ensure 1SR without requiring the validation of read-only transactions, it guarantees that the commit events of all update transactions (even non-conflicting ones) are totally ordered across all the nodes of a same group. Thus, a transaction T with final timestamp fsn is immediately committed only if there are no other transactions in both `pendQ` and `stableQ` with timestamp less that fsn. Otherwise, the transaction is buffered in `stableQ` till it can be ensured that no other pending transaction will ever receive a final commit timestamp less then fsn. In case (2), with a negative outcome, we have to release the locks acquired at the time of validation (Line 55) and to generate the appropriate notification (Line 56). The transaction is now processed (with a negative outcome), thus we generate a notification to the application thread. On the node where the transaction is local, this will signal the application thread (Lines 7 and 18) which returns the result of the commit request (Line 4). In either cases the transaction is removed from `pendQ` (Line 45).

In Lines 31 and 57, we process any pending transactions, since these are the only occasions where both `pendQ` and `stableQ` are possibly modified. It works as follows. A transaction T is committed if the following condition is true:

$$\exists\langle T, fsn\rangle : \langle T, fsn\rangle = \text{stableQ.head} \wedge \nexists\langle T', sn\rangle : \langle T', sn\rangle = \text{pendQ.head} \wedge sn \leq fsn$$

This condition is verified in Line 71, and if it is true, we apply the write set and release the acquired locks (Lines 86 and 87).

Since multiple transactions can end up with the same sid during the vote phase, we can only process these transactions when all of them have been applied. To this end we insert the transaction's state in a set of transactions waiting to be processed (Lines 89-93). When it is safe to make the snapshot visible (Lines 65-66, 71-72 and 82), we update the *commitId* timestamp of the node and invoke RELEASETXS, where all the transactions waiting to be processed are processed. The transactions that are local to the node signal the application thread.

Note that, differently from the original SCORe protocol, for simplicity sake we do not implement timeouts when the coordinator is gathering the vote messages nor when trying to acquire the locks. The original protocol also suggests distributed garbage collecting mechanisms to deal with obsolete data versions. Instead, we use a multi-version scheme that has a fixed number of versions (that can be parametrized by the user). This implies that, unlike the original protocol, our implementation may abort read-only transactions.

Since we are in a partially replicated environment, some data items may be not replicated locally. So, some read operations may have to access remote data replicated in other nodes. The pseudo-code of the read operations in SCORe is described in Figure 4.10, and works as follows.

Write operations are buffered in a private write set, which is only made visible upon transaction's commit, as in regular STM algorithms.

Read Operations. When a read operation is triggered in TribuSTM, the STM algorithm needs to return the value stored in the desired memory location, but that data may not be replicated locally, so it requests the distributed protocol to check for the data locality.

But first, the STM algorithm checks if the accessed data item has already been updated by the transaction, returning in this case the value present in the transaction's write set (it also inserts the accessed data item in the transaction's read set). If the accessed data item is not in the transaction's write set, the STM algorithm passes the control of the read operation to the distributed protocol, issuing ONREAD (with two parameters: transaction T and transactional metadata object m).

Remember that, when using the in-place metadata placement strategy (Section 3.1.3), the runtime provides the STM algorithm with the transactional metadata object associated with the accessed memory location.

Firstly, the protocol establishes which of the versions of the accessed data item is visible to the transaction. In SCORe, transactions establish the sid that they use to determine version's visibility upon their first read operation. After that, the protocol checks for the data locality in Line 113. Here, the protocol also checks for the locality of the object graph being accessed. This is what we called the *readOpt* optimization, in Section 4.4.1.2.

***ReadOpt* Optimization.** In Section 4.4.1.2, when explaining the serialization algorithm, even if an object should not be replicated in the local node we still save it in the memory map (Line 26, Figure 4.6). We use weak references in this map, so we do not prevent garbage collection. Here, when we request a remote object from the nodes that replicate it and we receive the object, the object is saved with weak references by the memory map, but with a strong reference by the transaction's read set. This is advantageous when we read a remote object that has references to other objects. This way we can do some sort of "caching", reducing the number of remote reads needed to access objects downstream of the already read (remote) object.

Recall Figure 4.5. The distribution metadata of the accessed transactional metadata has `partialGroup` $\neq$ `group` only when the accessed memory location is annotated with @Partial, the "border" of a partially replicated memory location. The distribution metadata has `partialGroup` $=$ `group` only when the accessed memory location is totally replicated (outside the "border" of a partially replicated memory location) or when the accessed memory location is referenced by a memory location annotated with @Partial (inside the "border" of a partially replicated memory location).

This is verified in Line 112, where if `partialGroup` $=$ `group` means that we already had to do a remote read for the "border" memory location and that we have the rest of the object graph cached. So, we just need to do a local read.

The logic associated with the execution of a read operation, in a node N_i, triggered by a (either local or remote) transaction T is encapsulated in the DOREAD function. First, if the transaction's sid is higher than the node's $nextId$, the latter is set equal to T.sid (Line 137), to ensure that update transactions that subsequently issue a commit request on this node are serialized after T. Next, the version visible by the transaction is determined by selecting the most recent version having a commit timestamp less than

```
8:  localTxs $\leftarrow \emptyset$
9:  remoteTxs $\leftarrow \emptyset$
10: pendQ $\leftarrow$ $nil$: priority queue of $\langle$id, sid$\rangle$
11: stableQ $\leftarrow$ $nil$: priority queue of $\langle$id, sid$\rangle$
12: toProcess $\leftarrow \emptyset$

13: procedure ONCOMMIT($T$)
14:     localTxs $\leftarrow$ localTxs[$T$.id $\mapsto$ $T$]
15:     group $\leftarrow$ $T$.involvedNodes
16:     $S \leftarrow$ CREATESTATE($T$)
17:     RMCAST([$S$], group)
18:     wait until $T$ is processed

19: procedure ONRMDELIVER([$S$], $s$)
20:     if SELF($s$) then
21:         $T \leftarrow$ localTxs($S$.id)
22:     else
23:         remoteTxs $\leftarrow$ remoteTxs[$S$.id $\mapsto$ $S$]
24:         $T \leftarrow$ RECREATETX($S$)
25:     outcome $\leftarrow$ VALIDATE($T$)
26:     next $\leftarrow -1$
27:     if outcome then
28:         next $\leftarrow$ ($N_i$.nextId $\leftarrow$ $N_i$.nextId +1)
29:         pendQ $\leftarrow$ pendQ $\cup$ $\langle T$.id, next$\rangle$
30:     RUCAST([$T$.id, outcome, next], $s$)
31:     PROCESSTX

32: procedure ONRUDELIVER([id, outcome, sid], $s$)
33:     $T \leftarrow$ localTxs(id)
34:     if not outcome then
35:         FINALIZEVOTEPHASE($T$, false)
36:     else
37:         if sid > $T$.maxVote then
38:             $T$.maxVote $\leftarrow$ sid
39:         if is last vote then
40:             FINALIZEVOTEPHASE($T$, true)

41: procedure ONRMDELIVER([id, outcome, sid], $s$)
42:     if outcome then
43:         $N_i$.nextId $\leftarrow$ MAX($N_i$.nextId, sid)
44:         stableQ $\leftarrow$ stableQ $\cup$ $\langle$id, sid$\rangle$
45:     pendQ $\leftarrow$ pendQ $\setminus\langle$id, –$\rangle$
46:     ADVANCECOMMITID
47:     if not outcome then
48:         if SELF($s$) then
49:             $T \leftarrow$ localTxs(id)
50:             localTxs $\leftarrow$ localTxs $\setminus$ $T$.id
51:         else
52:             $S \leftarrow$ remoteTxs(id)
53:             $T \leftarrow$ RECREATETX($S$)
54:             remoteTxs $\leftarrow$ remoteTxs $\setminus$ $T$.id
55:         UNLOCK($T$)
56:         COMMITPROCESSED($T$, false)
57:     PROCESSTX

58: procedure FINALIZEVOTEPHASE($T$, outcome)
59:     group $\leftarrow$ $T$.involvedNodes
60:     RMCAST([$T$.id, outcome, $T$.maxVote], group)

61: procedure PROCESSTX
62:     ADVANCECOMMITID
63:     snId $\leftarrow -1$
64:     loop
65:         if stableQ is empty then
66:             if snId ! $= -1$ then
67:                 $N_i$.commitId $\leftarrow$ snId
68:                 RELEASETXS
69:             return
70:         $\langle$id, sid$\rangle$ $\leftarrow$ stableQ.head
71:         if $\exists\langle$id$'$, sid$'\rangle$ $\in$ pendQ : sid$'$ $\leq$ sid then
72:             if snId ! $= -1$ then
73:                 $N_i$.commitId $\leftarrow$ snId
74:                 RELEASETXS
75:             return
76:         if $\exists$ remoteTxs(id) then
77:             $S \leftarrow$ remoteTxs(id)
78:             $T \leftarrow$ RECREATETX($S$)
79:         else
80:             $T \leftarrow$ localTxs(id)
81:         $T$.sid $\leftarrow$ sid
82:         if snId ! $= -1$ and sid > snId then
83:             $N_i$.commitId $\leftarrow$ snId
84:             RELEASETXS
85:         snId $\leftarrow$ $T$.sid
86:         APPLYWS($T$)
87:         UNLOCK($T$)
88:         stableQ $\leftarrow$ stableQ $\setminus\langle$id, sid$\rangle$
89:         if $\exists$ remoteTxs(id) then
90:             toProcess $\leftarrow$ toProcess $\cup$ $S$
91:             remoteTxs $\leftarrow$ remoteTxs $\setminus$ id
92:         else
93:             toProcess $\leftarrow$ toProcess $\cup$ CREATESTATE($T$)

94: procedure RELEASETXS
95:     for all tx $\in$ toProcess do
96:         if $\exists$ localTxs(tx.id) then
97:             $T \leftarrow$ localTxs(tx.id)
98:             localTxs $\leftarrow$ localTxs $\setminus$ tx.id
99:         else
100:            $T \leftarrow$ RECREATETX(tx)
101:        COMMITPROCESSED($T$, true)
102:        toProcess $\leftarrow$ toProcess $\setminus$ tx
```

Figure 4.9: Pseudo-code for the commit phase of the SCORe protocol.

T.sid (Line 140-142). But before that, the transaction must wait for the completion of the commit of any transaction that is updating the same data item and is currently in its commit phase (Line 139).

Remote Read Operation. The logic for handling remote read operations is defined by the ONRMDELIVER (Line 124) and ONRUDELIVER (Line 131) procedures. Note that, even though transactions establish their own sid only upon their first read operation, a node attempts to advance their local timestamps $commitId$ and $nextId$ whenever it receives a message related with a remote read from another node in the system, informing that snapshots with higher timestamps have already been committed. This aims at maximizing the freshness of the snapshots visible by transactions (Line 144).

In Lines 121-122 we can see a simple optimization done by SCORe. It aborts update transactions which, based on their sid, are forced to observe, upon a read operation, overwritten data by more recently committed transactions.

For more detailed and generic information about the SCORe protocol we direct the reader to the original article [Pel+12a].

4.5 Summary

In this chapter we described how we extended the software from Chapter 3 to support partial replication as a possible distributed memory model. In Section 4.2 we presented the framework's new programming model and its limitations/restrictions, and in Section 4.3 we detailed the specific extensions done to the framework. In Section 4.4 we provide a detailed description of the implementation, in the framework, of a partially replicated STM using SCORe to maintain consistency when committing transactions.

This chapter's contribution was featured in the article "Replicação Parcial com Memória Transacional Distribuída", Proceedings of the Simpósio de Informática (INForum), 2013 [Sil+13].

```
function onRead($T$, $m$)
    firstRead ← $T$.firstRead
    if firstRead then
        $T$.sid ← $N_i$.commitId
        $T$.firstRead ← false
    $M$ ← getMetadata($m$)
    group ← getGroup($M$)
    partialGroup ← getPartialGroup($M$)
    localRead ← contains(partialGroup, $N_i$)
    localGraph ← equals(partialGroup, group)
    if localRead and localGraph then
        ⟨value, lastCommitted, mostRecent⟩ ← doRead($T$.sid, $M$)
    else
        RMcast([$T$.id, $M$, firstRead, $T$.sid], partialGroup)
        wait until remote read is processed
        ⟨value, lastCommitted, mostRecent⟩ ← $T$.readRes
    if firstRead then
        $T$.sid ← max($T$.sid, lastCommitted)
    if $T$.isUpdate and not mostRecent then
        abort($T$)
    return value

procedure onRMDeliver([id, metadata, firstRead, sid], $s$)
    readSid ← sid
    if firstRead and $N_i$.commitId $>$ readSid then
        readSid ← $N_i$.commitId
    ⟨value, lastCommitted, mostRecent⟩ ← doRead(readSid, metadata)
    RUcast([id, value, lastCommitted, mostRecent], $s$)
    updateNodeTimestamps(readSid)

procedure onRUDeliver([id, value, lastCommitted, mostRecent], $s$)
    $T$ ← localTxs(id)
    $T$.readRes ← ⟨value, lastCommitted, mostRecent⟩
    updateNodeTimestamps(lastCommitted)
    remote read is processed

function doRead(sid, metadata)
    $N_i$.nextId ← max($N_i$.nextId, sid)
    $m$ ← memory(metadata)
    wait until ($N_i$.commitId $\geq$ sid or isExclusiveUnlocked($m$))
    version ← getLastVersion($m$)
    while version.verNum $>$ sid do
        version ← version.previous
    return ⟨version.value, $N_i$.commitId, isLastVersion($m$, version)⟩

procedure updateNodeTimestamps(lastCommitted)
    $N_i$.nextId ← max($N_i$.nextId, lastCommitted)
    $N_i$.maxSeenId ← max($N_i$.maxSeenId, lastCommitted)
    advanceCommitId

procedure advanceCommitId
    if $N_i$.maxSeenId $>$ $N_i$.commitId and pendQ is empty and stableQ is empty then
        $N_i$.commitId ← $N_i$.maxSeenId
```

Figure 4.10: Pseudo-code for the read event of the SCORe protocol.

5

Evaluation

This chapter reports the results of an experimental study aimed at evaluating the system's characteristics. We use some known benchmarks from the literature.

We begin by presenting the experimental settings used in these experiments, in Section 5.1. Next, in Section 5.2, we explain all the used benchmarks and its characteristics. Then, we proceed by presenting and analysing the obtained results in Section 5.3. We evaluate the system's memory consumption, the impact of data partitioning in the system's overall performance, the system's behaviour under different workloads, and so forth.

5.1 Experimental Settings

The experiments presented in this chapter were performed in two different clusters.

The first cluster, henceforth called Cluster@DI, is a heterogeneous cluster comprised by 8 nodes. The first five nodes are equipped with 2× Quad-Core AMD Opteron 2376 at 2.3 Ghz, 4 × 512 KB cache L2, and 16 GB of RAM. The last three nodes are equipped with 1× Quad-Core Intel Xeon X3450 at 2.66 Ghz (with hyper-threading) and 8 GB of RAM. The operating system is the Debian 5.0.10 Linux distribution with the Linux 2.6.26-2-amd64 kernel, and the nodes are interconnected via private gigabit ethernet. The max send buffer was set to 640 KB and the max receive buffer to 25 MB, as needed by the JGroups configuration. The installed Java platform is version 6, specifically OpenJDK Runtime Environment (IcedTea6 1.8.10, package 6b18-1.8.10-0 lenny2).

The second cluster, henceforth called Supernova, is also a heterogeneous cluster with a total of 570 nodes, with the characteristics presented in Table 5.1. At all times there are around 300 active nodes in the cluster. The operating system is Scientific Linux 5.9 with the Linux 2.6.18-348.3.1.el5 kernel, and the nodes are interconnected via Infiniband. The max send and receive buffers were set to the same values as in Cluster@DI, but due to cluster configurations nodes only allow a max receive buffer of 16.78 MB. The installed Java platform is version 6, specifically Java SE Runtime Environment (build 1.6.0_31-b04) and Java HotSpot 64-Bit Server VM (build 20.6-b01, mixed mode). To run computations the cluster uses PBSPro[1], a batch system, so the user has no control over when the jobs run and the nodes are shared among jobs.

Table 5.1: Specifications of the Supernova cluster nodes.

Specifications	Generation II (126)	Generation III (40)	Generation IV (404)
Processor	Intel Xeon E5345 2.33 Ghz	Intel Xeon L5420 2.5 Ghz	Intel Xeon X5650 2.67 Ghz
Num. Cores	8 cores (2× quad-core)	8 cores (2× quad-core)	12 cores (2× six-core)
L1/L2/L3 Cache	128 KB/8 MB/–	256 KB/12 MB/–	64 KB/1536 KB/12 MB
RAM	16 GB	16 GB	24 GB

Cluster@DI can be seen as a representative of small private clouds or data center envorinments, with dedicated servers and a fairly large amount of available (computational and memory) resources per node. In turn, Supernova can be seen as a representative of high performance computing (HPC) infrastrutures, which are typically characterized by somewhat powerful nodes but with more competitive resource sharing.

In Cluster@DI, the results were obtained from five runs of each experiment configuration, dropping the highest and the lowest results, and averaging the remaining three. In Supernova, we used only nodes of generation IV (to have control in what kind of nodes the experiments were executed), and since we could not have exclusive use of the nodes, the results were obtained from ten runs and we also present the corresponding standard deviations.

5.1.1 System Configurations

In order to allow a comparison between full and partial replication strategies, we used two configurations in the framework. For the full replication configuration we used the implementation done by Vale [Val12], and for the partial replication configuration we used our implementation (Section 4.4).

Partial Replication Configuration. In the partial replication configuration, the local STM layer uses a multi-version algorithm as described in [Pel+12a]. The DM implementation is the SCORe protocol described in Section 4.4.2.1.

Full Replication Configuration. In the full replication configuration, the local STM layer uses the TL2 algorithm [Dic+06]. The DM implementation is the non-voting certification protocol described in Section 2.3.2.3.

[1] http://www.pbsworks.com

Group Communication System. With regard to the underlying GCS providing the necessary communcation primitives (TOB for full replication and point-to-point communication for partial replication), we considered only one implementation: JGroups [Ban98] (but switching between GCS is done by parameterization when executing the target program, hence no code rewriting is needed). Concretely, we used JGroups version 3.3.0 final.

JGroups is a well known toolkit used in several projects (e.g., JBoss [Red13]). 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 (for the full replication configuration).

JGroups provides non-uniform total order using a fixed sequencer. Informally, as depicted in Figure 5.1, 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, as the sequencer should have the upper hand, since it does not need to unicast its messages to itself. The relaxation of the uniformity property is likely to allow higher throughput, and even more unfairness, because it does not require all nodes to send back acknowledgements to the sequencer.

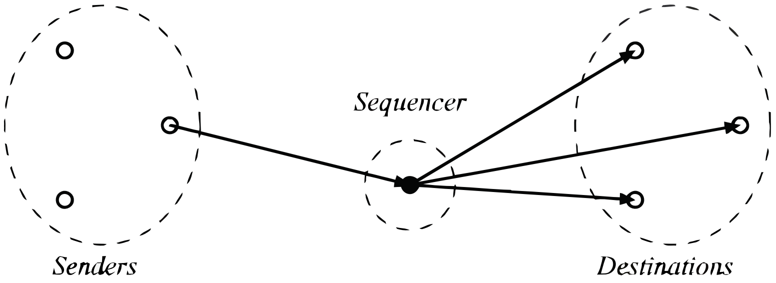


Figure 5.1: Fixed sequencer.

5.2 Benchmarks

5.2.1 Red-Black Tree Microbenchmark

To evaluate our implementation, we start by considering a common microbenchmark in the literature (taken from the Deuce STM benchmarks), the Red-Black Tree. It is composed of three types 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.

[2] `https://github.com/belaban/JGroups`

The microbenchmark was parametrized according to Table 5.2[3]. The tree was populated with 32 768 pseudo-randomly generated values, ranging from 0 to 131 072, thus having a height of 15. Each thread executed 10% of write transactions.

Table 5.2: Parametrization of the Red-Black Tree microbenchmark.

Initial size (-i)	**Value range (-r)**	**Write transactions % (-w)**
32768(2^{15})	131072(4× initial size)	10

This workload is characterized by very small and fast transactions that perform little work, and contention is very low.

In order to use the partial replication configuration we had to add the corresponding annotation @Partial. Following the idea described in Section 4.2, we applied the annotation to the value stored by the tree nodes, as depicted in Figure 5.2.

```
public class Node {
  int key;
  @Partial
  int value;
  Node p; // parent
  Node l; // left child
  Node r; // right child
  int c; // color

  public Node() {}
}
```

Figure 5.2: Annotation @Partial applied in tree node.

5.2.2 Adapted Vacation

This benchmark is from the STAMP suite [Min+08], a more complex benchmark that allows us to test the system with workloads substantially different from the previous microbenchmark.

The Vacation benchmark emulates a travel reservation system. This application implements an online transaction processing system that 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/transactions: reservations, cancellations and updates [Min+08]. All these transactions are update transactions. So, we adapted the benchmark in order to also allow the execution of read-only transactions. Our adaptation consists of implementing a new read-only session/transaction that consults reservations.

[3] Unless stated otherwise, for all the benchmarks, we used the parametrizations presented in this section's tables.

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

The benchmark was parametrized according to Table 5.3, which is the low contention configuration presented in [Min+08], but user transactions now consist of reservations and consultations. The database has 16 384 records of each reservation item, and clients perform 4096 sessions. Of these sessions, 98% are user sessions, and 90% of them are consultations (the rest being reservations or cancellations), the remainder create or destroy items. Sessions operate on up to 2 items and are performed on 90% of the total records.

Table 5.3: Parametrization of the Vacation benchmark.

Operated items (-n)	Accessible records (-q)	User transactions (-u)	Records (-r)	Sessions (-t)
2	90	98	16384	4096

The original benchmark used only one node to populate the data structures. We modified it so that each group has one node (the "group master") that populates a part of the data structures, enabling us to use the local data partitioner.

Similarly to the previous microbenchmark, in this benchmark we applied the @Partial annotation to the values (in the nodes) of the trees that work as the system's database.

5.2.3 TPC-W

TPC-W [Tra13] models an online bookstore. Servers handle 14 different user requests such as browsing, searching, adding products to a shopping cart, or placing an order. The user requests (operations) are much more demanding in terms of processing power when compared to the previous benchmarks, allowing us to test the system with more challenging workloads. Additionally, this benchmark is representative of client-server architectures. The initial data set is generated with 1000 items. For our experiment we ran the browsing mix, which consists of 95% of operations related with browsing and 5% related with purchases. This benchmark was taken from the freely available repository[4].

As in the other two benchmarks, we applied the @Partial annotation to the values stored by the used data structures (namely red-black trees and hash maps)[5].

5.3 Results

Now, we present and analyse the obtained results.

[4] https://github.com/pedrogomes/tpcw-benchmark

[5] For detailed information about the used data structures and the corresponding application of the @Partial annotation, the reader can go to https://github.com/jaasilva/TPCw-benchmark/tree/partial_random

5.3.1 Memory Consumption

One of the obvious characteristics of partial data replication is that as the number of groups grows, the amount of data that each group replicates decreases. This means that, in a concrete system, as the number of groups grows each node consumes less memory.

To assess if our system embeds this characteristic, we used the Red-Black Tree microbenchmark. This microbenchmark has a population phase where *one* of the nodes populates the entire data structure, hence to ensure an even data partitioning we used the round robin data partitioner. We also parametrized the microbenchmark with 0% of write transactions to verify the memory used once the population of the data structure is done, without removing or inserting nodes.[6]

We compared the partial replication configuration *versus* the full replication configuration, and we obtained the result shown in Figure 5.3.

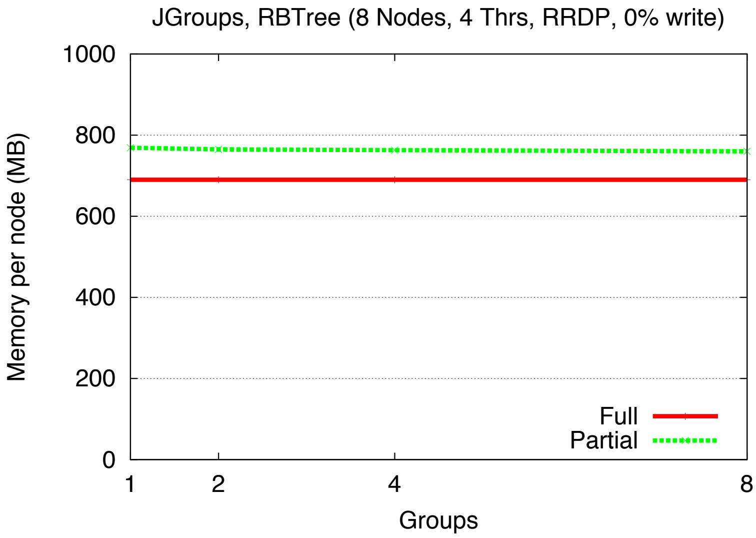


Figure 5.3: Memory consumption on the Red-Black Tree microbenchmark (Cluster@DI).

This was an expected result. As Figure 5.2 shows, the tree's nodes store little information. Each node stores a value of type integer that is the same as the key, and it is that value that is annotated with the @Partial annotation. This means that the integer value is partially replicated. The result of this, in Java, is that if group A does not replicate integer value B, all the nodes in group A will still have the integer value but with the value 0 (zero), and will have to request the partially replicated value from another group. So, in this microbenchmark we will never save memory. In fact, Figure 5.2 shows that the partial replication configuration uses more memory. This happens because both the transactional and distribution metadata, in the partial replication configuration, store substantially more information that the ones used by the full replication configuration.

[6] We also used the JVM flag `-Xincgc`. It allows applications to consume less memory since it does an incremental garbage collection, and the consumed memory has less fluctuations.

The partial replication configuration will only save memory when the partially replicated fields are objects with a considerable amount of data, because in this case, the groups that does not replicate the field will have a pointer to a null value instead of the object.

To verify our intuition, we modified the Red-Black Tree microbenchmark in order to store in each node an object with 3 MB of data. We named it Adapted Red-Black Tree version 1 microbenchmark. We parametrized the microbenchmark according to Table 5.4.

Table 5.4: Parametrization of the Adapted Red-Black Tree version 1 microbenchmark.

Initial size (-i)	**Value range (-r)**	**Write transactions % (-w)**
$1024(2^{10})$	$4096(4\times$ initial size)	0

The result is shown in Figure 5.4. With this experiment we can see that, in fact, as the number of groups grows, the memory used by each node decreases.

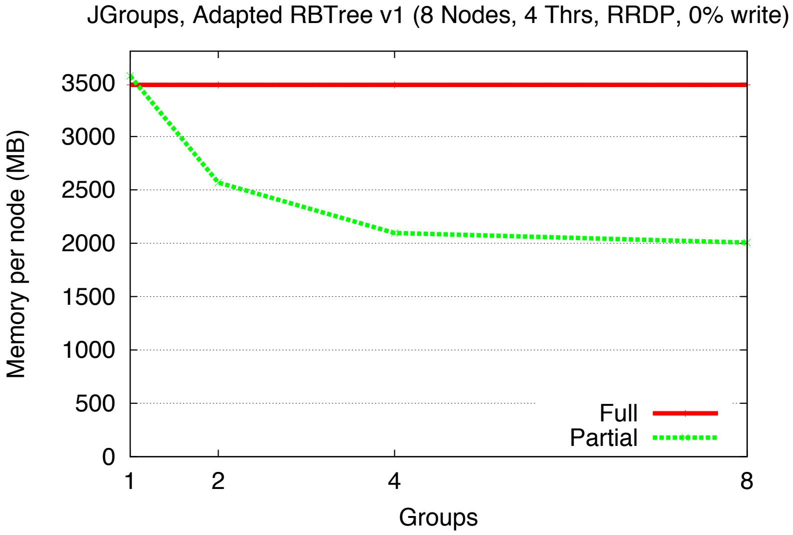


Figure 5.4: Memory consumption on the Adapted Red-Black Tree version 1 microbenchmark (Cluster@DI).

Note that with one group partial replication is mimicking full replication, since there is only one group comprised by all the nodes in the system and consequently all nodes replicate the entire system's data set. We can see the expected overhead verified in the previous test. With one group, partial replication consumes more memory that the full replication configuration.

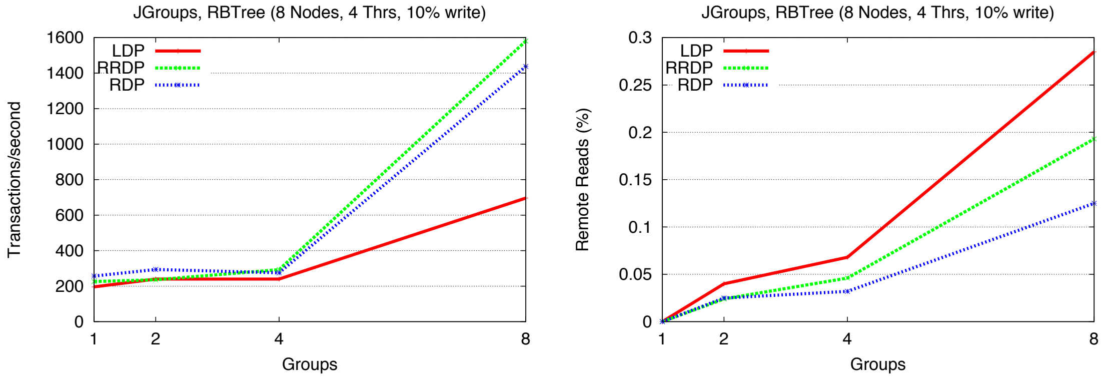
In the other end of the spectrum, with eight groups, partial replication is mimicking pure data distribution, since there are eight groups and each group is comprised by only one node. This means that partially replicated data items are replicated only by one node.

5.3.2 Impact of Data Partitioners

In Section 4.3.3, we presented the data partitioning strategies implemented in our framework. Since these strategies decide where newly created data is located, they have impact on the application's behaviour.

To study this impact we used the Red-Black Tree microbenchmark, and we executed the microbenchmark using the three data partitioning strategies. Recall that we have not modified the microbenchmark, so the population phase is performed by a single node, and the amount of remote read operations is randomly decided.

The result is depicted in Figure 5.5. Since the data partitioning strategies influence the amount of remote read operations performed by the system we also measured its percentage.



(a) Throughput.

(b) Remote reads percentage.

Figure 5.5: Throughput and remote reads percentage on the Red-Black Tree microbechmark (Cluster@DI).

The local data partitioner has the weakest performance (Figure 5.5a) because of its nature. Since objects are replicated by the group of the node where they were created at (Section 4.3.3), during the population phase of the microbenchmark all the created objects are replicated in only one group, which causes some imbalance, forcing the other groups to perform a lot more remote read operations (Figure 5.5b). This partitioning strategy is also dependent on workload distribution (aspect that the partial replication configuration can control, as presented in Section 5.3.4.2) performed by the system, and on the number of objects created by transaction. The round robin and the random data partitioners behave very similar because of their identical nature.

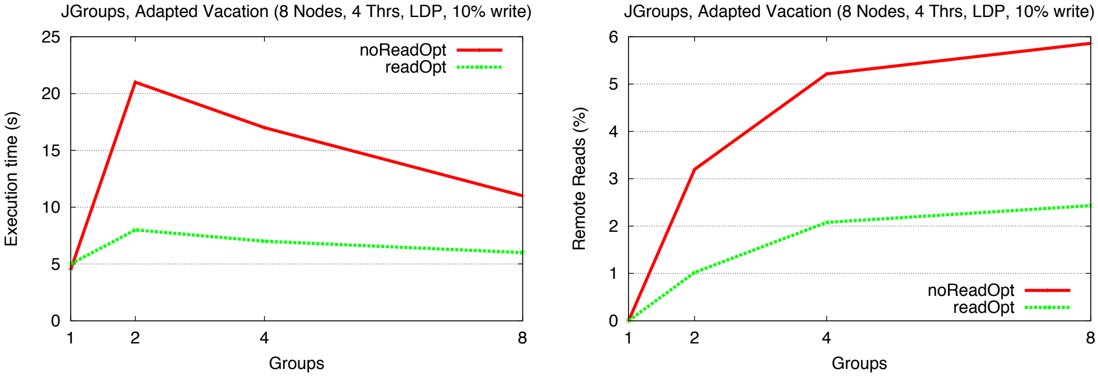
This results claim for a data partitioning strategy that takes into account both the load balancing factor and the affinity between data.

5.3.3 ReadOpt Optimization

In Section 4.4.2.1 we discussed an optimization that we called *readOpt*. This optimization enables the system to perform some kind of caching when reading remote objects, reducing the number of remote read operations.

To prove our intuition, we used the Vacation benchmark and executed it with two implementations of the SCORe protocol - one with the *readOpt* optimization and one without it. Figure 5.6 shows the result. Take into account that this benchmark measures the execution time (lower is better).

We did not use the Red-Black Tree for this experiment because, as already stated, the only partially replicated fields are the tree nodes' integer values, and this optimization will only be advantageous in cases where the partially replicated fields are objects that have references to other objects.



(a) Execution time.

(b) Remote reads percentage.

Figure 5.6: Execution time and remote reads percentage on the Adapted Vacation benchmark (Cluster@DI).

As Figure 5.6b presents, this optimization indeed reduces the amount of remote read operations. Naturally, as the number of groups grows the percentage of remote read operations also grows, but without this optimization the percentage of remote read operations grows 3 times more than with the optimization. Figure 5.6a also shows the impact of this optimization in the benchmark execution time, being that with the optimization, the benchmark consistently performs better.

5.3.4 Partial Replication *versus* Full Replication

We think that it is important to compare the partial replication configuration with the full replication configuration. Given that, next we present some experiments that compare both configurations.

5.3.4.1 Read Transactions

Both configurations allow read-only transactions to commit without the need for a distributed validation. Hence it is interesting to see how they perform in a scenario with just read-only transactions.

To this extent, we applied both configurations to the Red-Black Tree microbenchmark parametrized with 0% write transactions. The result is shown in Figure 5.7.

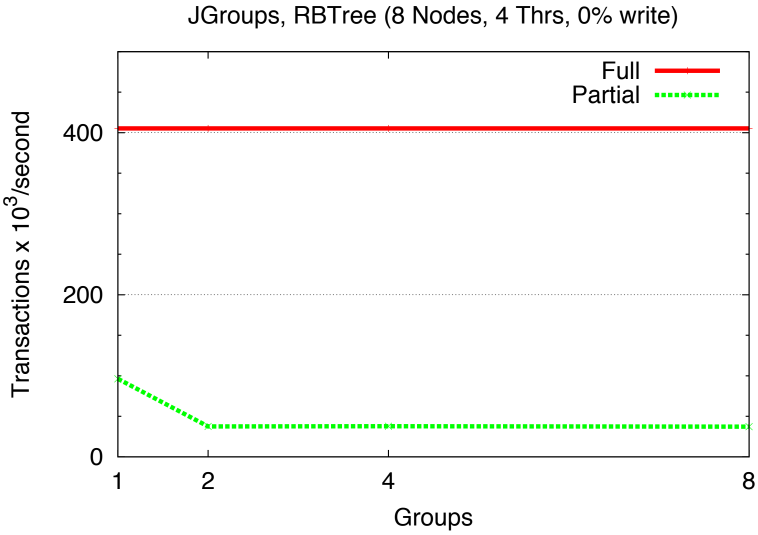


Figure 5.7: System's throughput with 100% read-only transactions on the Red-Black Tree microbenchmark (Cluster@DI).

The partial replication configuration, when using one group, mimics full replication, thus there are no remote read operations. Given that, we can see the overhead imposed by the partial replication configuration when executing local read operations. the chart shows that the throughput of the partial replication configuration is 4 times less when compared with the throughput of the full replication configuration.

When the number of groups grows, the partial replication configuration requires remote read operations (due to the distribution of the data) cutting the throughput in half.

5.3.4.2 Write Transactions

Typical scenarios also have write transactions, so we experimented both configurations in such scenarios. First, we used the Red-Black Tree microbenchmark parametrized with 10% write transactions. For the partial replication configuration we used the round robin data partitioner. The result is depicted in Figure 5.8.

We can observe that the partial replication configuration's throughput is very low regarding its full replication counterpart. This can be justified with the simple reason that all the write transactions modify the structure of the tree. Remember that the write transactions in the Red-Black Tree microbenchmark insert or remove nodes from the tree and those nodes are fully replicated. Given that, all the transactions that need distributed validation (i.e., the write transactions) have to be validated by all the nodes in the system. Thus, in this case, the use of partial replication reduces the overall performance of the system.

Trying to observe the behaviour of the partial replication configuration with different study cases, we also experimented with two other benchmarks, namely the Vacation benchmark and TPC-W. These benchmarks' transactions have a mixed behaviour that

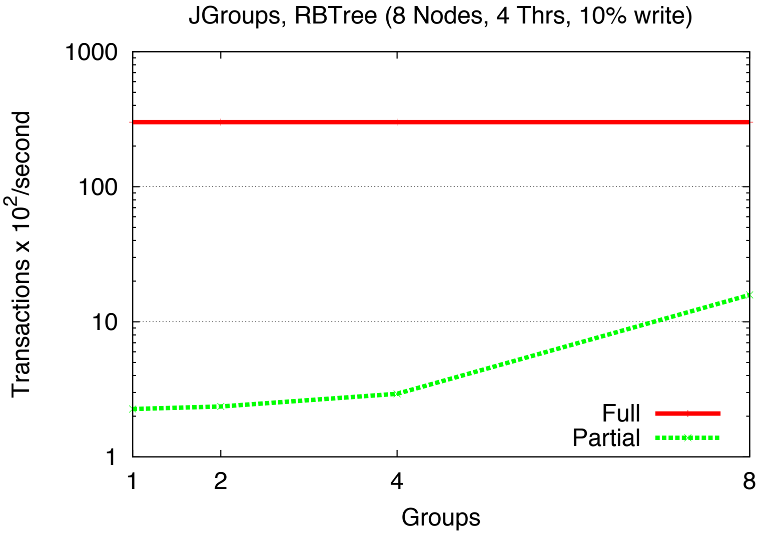
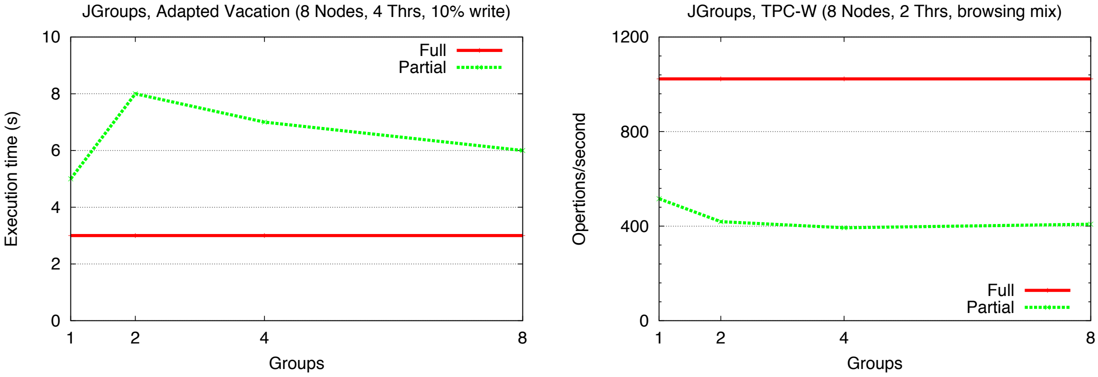


Figure 5.8: System's throughput with 10% write transactions on the Red-Black Tree microbenchmark (Cluster@DI).

involve both modifications to the data structures and to the partially replicated fields. Figure 5.9 shows the result.



(a) Vacation.

(b) TPC-W.

Figure 5.9: System's throughput using mixed behaviour benchmarks (Cluster@DI).

With this mixed behaviour, we can observe that the partial replication configuration still reduces the overall system's performance, but with a smaller gap between them. With the Vacation benchmark, in Figure 5.9a, the throughput of the partial replication configuration is around half of the full replication configuration's throughput. And with TPC-W, in Figure 5.9b, the throughput of the partial replication configuration is around 2.6 times lower than the full replication configuration's throughput. Thought it still cannot outperform the full replication configuration in neither cases since during these experiments, with the Vacation benchmark 90% of the transactions had to be validated by all the nodes, and with TPC-W it dropped to around 85%, which still is too much.

To check if the partial replication configuration could take advantage of its distributed and replicated nature, we modified the Red-Black Tree microbenchmark in order to have write transactions that do not modify the structure of the tree and thus do not have to be validated by all the system's nodes. We named it Adapted Red-Black Tree version 2 microbenchmark, and write transactions just modify the values stored by the tree's nodes. The result is shown in Figure 5.10.

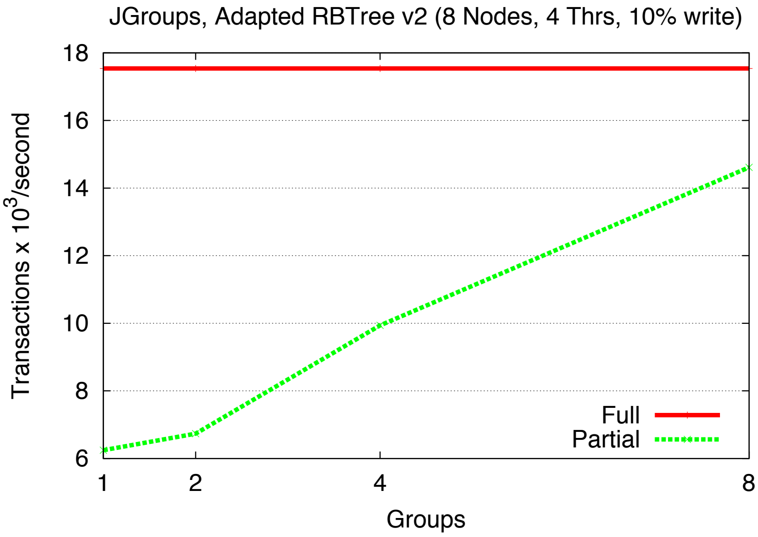


Figure 5.10: System's throughput with 10% write transactions on the Adapted Red-Black Tree version 2 microbenchmark (Cluster@DI).

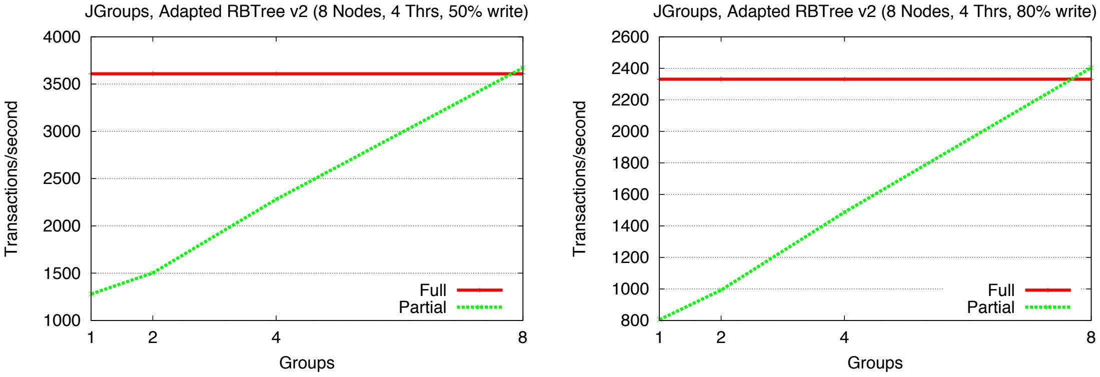
We can see that, in fact, with this modified benchmark, the partial replication configuration takes advantage of its distributed validation protocol. As the number of groups grows, transactions have to be validated by less nodes allowing the system to scale. Nevertheless, it still does not outperform the full replication configuration. We think this will only be possible in a system with a reasonable amount of nodes, or in a scenario with a higher percentage of write transactions (modifying only the partially replicated fields).

To put a part of this intuition to the test, we ran the same experiment as before with higher percentages of write transactions (namely 50% and 80%), and the result is depicted in Figure 5.11.

We can observe that as the number of groups grows, the system with the partial replication configuration scales and, in fact, it surpasses the full replication configuration, even if just for a short difference. This result demonstrates that when the number of groups grows, write transactions have to be validated by less nodes improving the system's overall performance. On the other hand, the full replication configuration, using the TOB primitive, is somewhat limited by the latency when totally ordering messages.

To put the other part of out intuition to the test, we ran the same experiment as in Figure 5.10 but with a larger number of nodes, namely 20 nodes. Figure 5.12 depicts the result.

As we can see, in fact, in a system with a larger number of nodes, as the number



(a) 50% write transactions.

(b) 80% write transactions.

Figure 5.11: System's throughput with higher percentages of write transactions on the Adapted Red-Black Tree version 2 microbenchmark (Cluster@DI).

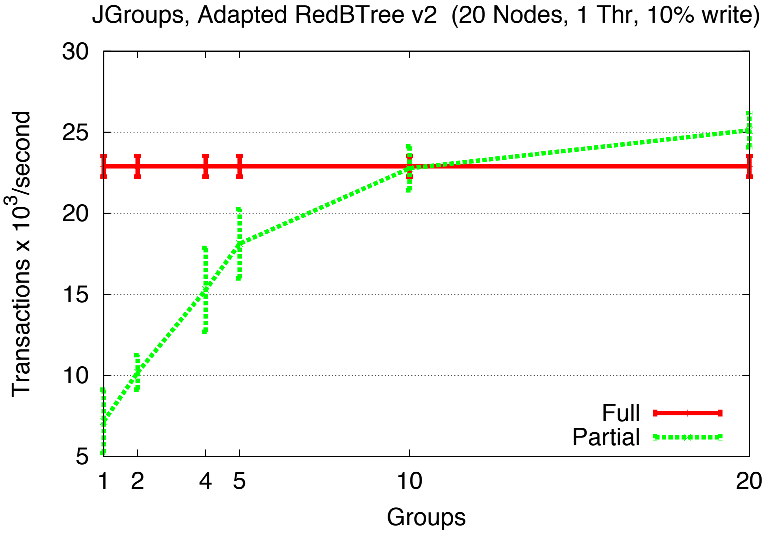
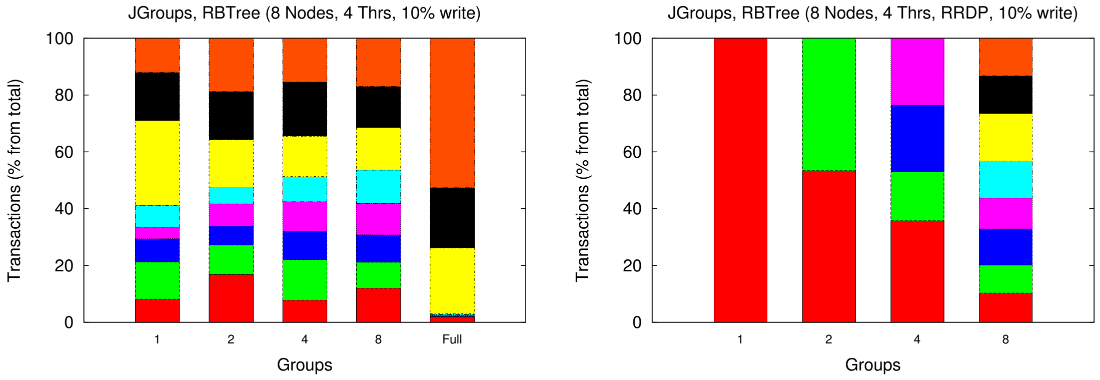


Figure 5.12: System's throughput with 10% write transactions on the Adapted Red-Black Tree version 2 microbenchmark (Supernova).

of groups grows, the partial replication configuration starts to surpass the full replication configuration's throughput (for a large number of groups). We can justify this, once again, with the TOB primitive used by the full replication configuration. In the partial replication configuration, as the number of groups grows each write transaction needs to be validated by a decreasing number of nodes, having lower latencies in the distributed voting compared with the full replication configuration's total ordering of messages.

Workload distribution. During these experiments we also measured the work done by each node and by each group in an attempt to verify if all the nodes/groups contribute with the same amount of work to the system's overall throughput. The result can be seen in Figure 5.13. These measures were taken during the execution of the Red-Black Tree microbenchmark (as seen in Figure 5.8), but for all the other benchmarks the results are similar. In Figure 5.13a each color represents one replica, and in Figure 5.13b each color represent one group.



(a) Per node.

(b) Per group.

Figure 5.13: Execution breakdown on the Red-Black Tree microbenchmark (Cluster@DI).

As we can see every node (Figure5.13a) and every group (Figure 5.13b) contribute roughly with the same amount of work for the system's overall throughput, thus SCORe ordering mechanism achieves fairness. The same cannot be said when using the full replication configuration that uses JGroups' implementation of the TOB communication primitive. JGroups provides non-uniform total order using a fixed sequencer (Section 5.1.1) known for its unfairness regarding message ordering, where the sequencer has the upper hand having its messages consistently ordered before everyone else's, hence dominating the system.

Thus, the partial replication configuration achieves better workload distribution. On the other hand, our intuition says that, in the full replication configuration, the sequencer will probably become the main bottleneck of the system.

5.3.4.3 Time Breakdown

As seen it the previous experiments, the partial replication configuration presents a great overhead regarding the full replication configuration. In order to verify its source, we display the main differences between them in terms of the execution time of some parts of a transaction's execution and validation.

To that extent, we compare the execution of the Red-Black Tree microbenchmark using both configurations. The used partial replication configuration for this experiment was parametrized with the round robin data partitioner and with a replication factor of 2 (4 groups, in this case). The result is shown in Figure 5.14.

Right away, we can see that the partial replication configuration has longer execution times for the different procedures. The labels in the x-axis have the following meaning:

Val Is the time spent validating transactions (i.e., acquiring locks and validating the read set);

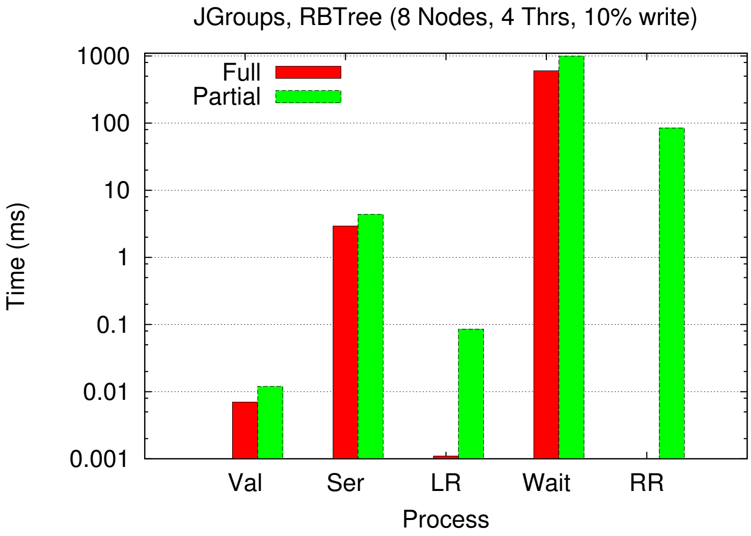


Figure 5.14: Execution times breakdown on the Red-Black Tree microbenchmark (Cluster@DI).

Ser Is the time spent serializing messages to be sent through the network;

LR Is the time spent executing a local read operation;

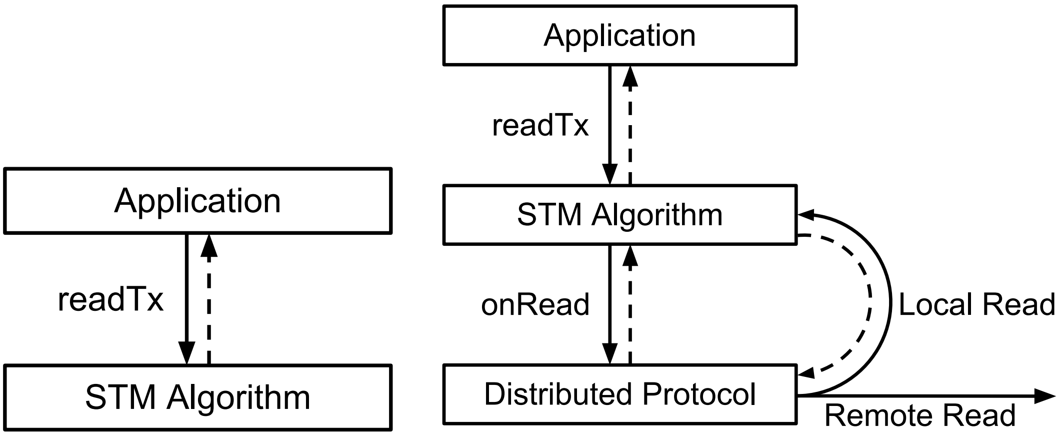
Wait has different meanings for the two configurations. For the full replication configuration, it means the time waiting between the sending and the reception of a totally ordered message. For the partial configuration configuration, it means the time waiting between the sending of a prepare message and the reception of the last vote from the participants; and

RR Is the time spent executing a remote read operation.

Validation. The validation phase in the partial replication configuration takes slightly longer that in the full replication configuration, since it has more steps. The TL2 validation locks the write set and checks the validity of the read set. The SCORe validation acquires exclusive and shared locks for both the write and read sets, respectively, and it checks the validity of the read set.

Serialization. The serialization of messages also takes slightly longer that in the full replication configuration. This happens because, besides the objects, both the transactional and distribution metadata, in the partial replication configuration, store substantially more information that the ones used by the full replication configuration.

Local Read. The local read operations executed by the partial replication configuration take much longer that the ones executed by the full replication configuration. When, with the full replication configuration, the local read operations take around one microsecond, with the partial replication configuration take around 80 microseconds. This has to do with the increased complexity of the read operations done by SCORe. The TL2 read operation just checks a logical clock and reads a memory location. On the other hand, the SCORe read operation checks the locality of the data, checks a logical clock, may have to wait on a condition, and has to access the versions list. More over, the framework's architecture imposes an indirection that is not observed in the TL2 read operations. This indirection is depicted in Figure 5.15.



(a) TL2.

(b) SCORe.

Figure 5.15: Read operation done by TL2 and by SCORe.

When using the TL2 algorithm (Figure 5.15a), applications execute a transactional read operation directly to the STM algorithm and the algorithm responds back, directly. But when using SCORe (Figure 5.15b), since it is the distributed protocol that has the power to check the locality of the data, the algorithm is forced to request the distributed protocol to execute the read operation.

Waiting. The full replication configuration uses a TOB communication primitive to totally order messages. This imposes some latency in the messages delivery. The partial replication configuration uses point-to-point communication, but it has to wait for the votes of all the participants. Usually, this imposes a larger latency compared with the TOB primitive (when waiting for the votes of many nodes).

Remote Read. Obviously, the full replication configuration does not execute remote read operations. But this is inherent to the partial replication configuration and it imposes a high latency in those read operations (around 80 milliseconds).

5.4 Final Remarks

Regarding the system's memory consumption using the partial replication configuration, we observed that it behaves as expected, consuming less memory per node than the full replication configuration. This only happens if the partially replicated data is of considerable size.

We demonstrated that the different data partitioners have a considerable impact on the system's overall performance, and the appropriate use of them can bring a good overall improvement. Thus, the proper data partitioner depends on the applications' behaviour.

The *readOpt* optimization works as expected, reducing the number of remote read operations performed by the system, when using the partial replication configuration, thus improving the system's overall performance.

When comparing both configuration, we have to take into account that those configurations have considerably different types of protocols and STM algorithms. The multi version algorithm used by SCORe is much more complex than TL2 and the non-voting certification protocol is much simpler (requiring only one communication step) than the distributed protocol used by SCORe (based in 2PC). Those differences are evident in Section 5.3.4.3. The partial replication configuration can only take advantage of its protocol when the applications have write transactions that modify the partially replicated data. When transactions modify fully replicated data, the partial replication configuration will always perform poorly, since it requires three communication steps to confirm such transactions. We also observed that the partial replication configuration distributed the workload among the system's nodes in a fair manner, unlike the full replication configuration where the sequencer dominates the system.

A more general remark related to the advantage of using partial replication. It is a given that partial replication can be advantageous in scenarios where there is a small amount of transactions that modify fully replicated data (usually the data structures). Given that, when using partial replication, partially replicate the data structures is a bad idea, we opt by fully replicating them. In this case, if the relative amount of operations that modify the data structures is too large, the overhead of the voting protocol, like SCORe, will be too much.

These results call for some sort of hybrid protocol capable of seamlessly combining both full and partial data replication in the same system.

5.5 Summary

In this chapter, we presented the results of an experimental evaluation of the implementation of a partially replicated STM described in Section 4.4.

We started by describing the test environment used for this evaluation in Section 5.1. Next, in Section 5.2, we characterized the used benchmarks with the corresponding parametrizations. In Section 5.3, we present and analyse the obtained results. We finalize this chapter, with Section 5.4, making the final remarks resulting from the results' analysis.

6 Conclusion

We now present our main conclusions and introduce interesting directions for future work.

6.1 Concluding Remarks

Despite of initially being studied in the context of chip-level multiprocessing, the benefits of STM over traditional concurrency control methods make it an interesting model for distributed concurrency control. However, the existing DSTM frameworks are tied to a specific distributed memory model and provide an intrusive interface to applications.

This thesis addresses the problem of implementing a modular DSTM system for a general-purpose programming language providing partial data replication as a possible distributed memory model, and we claim that it is possible to combine both full and partial data replication in a DSTM system.

We propose an extension to the TribuDSTM framework to support partial data replication. This extension keeps the framework backwards compatible, since it can still be used with the previous implemented full replication configuration, by parametrization of the system when executing the target application. It still allows the implementation of different distributed memory models and the associated protocols to support distributed transactions. We also maintain the programming model as less intrusive as possible. This paves the way of for the study of the impact of different distributed memory models in different contexts.

The provided programming model requires only small modifications to the applications, presenting sufficient expressiveness to develop all kinds of applications. At least the benchmarks used in the evaluation were relatively easy to adapt, and following the recommended usage presented previously, the application of the @Partial annotation was confined to the data structures.

Using our proposed framework, we provide an implementation of a partially replicated STM which resorts to the SCORe protocol to commit distributed transactions. We evaluated the prototype implementation under different workloads using well known benchmarks.

From our evaluation, we are able to draw some initial conclusions on the applicability of partial replication in the specific context of TM for general-purpose programming languages. First of all, we conclude that it must not be used to replicate data structures. The overhead of remote read operations when traversing a partially replicated data structure would impose a great performance penalty. Accordingly, we restricted its use to the contents of such data structure and, even so, the system only reduces memory consumption when the partially replicated data is of a considerable size (e.g., the @Partial annotation applied to an integer field would have little effect).

In this context, both partial and full replication coexist. However, some constraints must be applied for the system to deliver good performance, given that the cost of confirming transactions that operate on fully replicated data, in a partial replication configuration, is substantially higher then in a full replication configuration. This is based on the fact that the protocol used in the partial replication configuration to confirm transactions requires three communication steps (since it is based in 2PC), while the protocol used in the full replication configuration requires only one step (it uses a TOB communication primitive). Our particular implementation of a partially replicated STM takes advantage of the partitioning of the nodes into groups to minimize this impact and reduce the latency when confirming transactions. In fact, as the system scales, we can reduce the number of nodes that confirm transactions limiting the data's replication factor.

Our evaluation also gave us some indicators that partial replication can be advantageous in certain contexts, e.g., in applications where there is a small amount of transactions that modify fully replicated data. This is based on the fact that the majority of fully replicated STMs make use of TOB communication primitives to totally order messages, being these their main bottleneck.

In conclusion, we were able to sustain our claim and, as such, the implemented prototype effectively combines partial and full replication in a DSTM system. The presented DSTM framework allows the easy implementation of replicated STMs using both full and partial data replication, and it provides a non-intrusive interface to applications.

6.2 Future Work

Interesting directions for future work include:

- Implementation of the state of the art algorithms of partially replicated STM distributed commit and memory consistency (presented in Section 2.3.3) on our framework, and subsequent evaluation and comparison;

- Implementation of group hierarchies, trying to take advantage of data locality, and possibly trying to apply this STM model to cloud environments;

- Under a partially replicated STM, support data and thread/transaction migration considering the affinity between data items and threads/transactions;

- The design and implementation of algorithms allowing the seamless combination of partial and full data replication strategies presents a challenging and technical problem that can be investigated;

- Support different replication factors for different data items, giving the possibility of creating "data hotspots", where the most requested data items dynamically raise their replication factor;

- A solution with partially replicated arrays, i.e., where arrays would be partitioned among various groups, would present a challenging and interesting problem that could be investigated; and

- The study of the impact of different serialization libraries (like kryo [Kry] or fst [Fst]) in the serialization time and in the communication.

Bibliography

[Uui] *A Universally Unique IDentifier (UUID) URN Namespace*. http://www.ietf.org/rfc/rfc4122.txt. May 2013.

[Aba+08] M. Abadi, A. Birrell, T. Harris, and M. Isard. "Semantics of transactional memory and automatic mutual exclusion". In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '08. San Francisco, California, USA: ACM, 2008, pp. 63–74. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328449. URL: http://doi.acm.org/10.1145/1328438.1328449.

[Agr+97] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. "Exploiting Atomic Broadcast in Replicated Databases (Extended Abstract)". In: *Proceedings of the Third International Euro-Par Conference on Parallel Processing*. Euro-Par '97. London, UK, UK: Springer-Verlag, 1997, pp. 496–503. ISBN: 3-540-63440-1. URL: http://dl.acm.org/citation.cfm?id=646662.699395.

[Alo97] G. Alonso. "Partial Database Replication and Group Communication Primitives (Extended Abstract)". In: *in Proceedings of the 2 nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*. 1997, pp. 171–176.

[AD76] P. A. Alsberg and J. D. Day. "A principle for resilient sharing of distributed resources". In: *Proceedings of the 2nd international conference on Software engineering*. ICSE '76. San Francisco, California, United States: IEEE Computer Society Press, 1976, pp. 562–570. URL: http://dl.acm.org/citation.cfm?id=800253.807732.

[Ami+00] Y. Amir, C. Danilov, and S. Stanton. "A low latency, loss tolerant architecture and protocol for wide area group communication". In: *Proceedings of the International Conference on Dependable Systems and Networks*. 2000, pp. 327–336. DOI: 10.1109/ICDSN.2000.857557.

[Bad+06] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. "Grid Computing: Software Environments and Tools". In: Springer-Verlag,

2006. Chap. Programming, Deploying, Composing, for the Grid. URL: http://www-sop.inria.fr/oasis/proactive/userfiles/file/papers/ProgrammingComposingDeploying.pdf.

[Ban98] B. Ban. "Design and implementation of a reliable group communication toolkit for java". In: *Cornell University* (1998).

[Ber+87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987. ISBN: 0-201-10715-5.

[Blo70] B. H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: http://doi.acm.org/10.1145/362686.362692.

[Blu+06] C. Blundell, E. Lewis, and M. Martin. "Subtleties of Transactional Memory Atomicity Semantics". In: *IEEE Comput. Archit. Lett.* 5.2 (July 2006), pp. 17–17. ISSN: 1556-6056. DOI: 10.1109/L-CA.2006.18. URL: http://dx.doi.org/10.1109/L-CA.2006.18.

[Boc+08] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. "Software transactional memory for large scale clusters". In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. PPoPP '08. Salt Lake City, UT, USA: ACM, 2008, pp. 247–258. ISBN: 978-1-59593-795-7. DOI: 10.1145/1345206.1345242. URL: http://doi.acm.org/10.1145/1345206.1345242.

[BM03] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey". In: *Internet Mathematics*. 2003, pp. 636–646.

[Bud+93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. "Distributed systems (2nd Ed.)" In: ed. by S. Mullender. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993. Chap. The primary-backup approach, pp. 199–216. ISBN: 0-201-62427-3. URL: http://dl.acm.org/citation.cfm?id=302430.302438.

[CRS06] J. Cachopo and A. Rito-Silva. "Versioned boxes as the basis for memory transactions". In: *Science of Computer Programming* 63.2 (2006), pp. 172 –185. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2006.05.009. URL: http://www.sciencedirect.com/science/article/pii/S0167642306001171.

[Car+10] N. Carvalho, P. Romano, and L. Rodrigues. "Asynchronous lease-based replication of software transactional memory". In: *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*. Middleware '10. Bangalore, India: Springer-Verlag, 2010, pp. 376–396. ISBN: 978-3-642-16954-0. URL: http://dl.acm.org/citation.cfm?id=2023718.2023744.

[Car+11a] N. Carvalho, P. Romano, and L. Rodrigues. "A Generic Framework for Replicated Software Transactional Memories". In: *Proceedings of the 2011 IEEE 10th International Symposium on Network Computing and Applications*. NCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 271–274. ISBN: 978-0-7695-4489-2. DOI: 10.1109/NCA.2011.45. URL: http://dx.doi.org/10.1109/NCA.2011.45.

[Car+11b] N. Carvalho, P. Romano, and L. Rodrigues. "SCert: Speculative certification in replicated software transactional memories". In: *Proceedings of the 4th Annual International Conference on Systems and Storage*. SYSTOR '11. Haifa, Israel: ACM, 2011, 10:1–10:13. ISBN: 978-1-4503-0773-4. DOI: 10.1145/1987816.1987830. URL: http://doi.acm.org/10.1145/1987816.1987830.

[CT96] T. D. Chandra and S. Toueg. "Unreliable failure detectors for reliable distributed systems". In: *J. ACM* 43.2 (Mar. 1996), pp. 225–267. ISSN: 0004-5411. DOI: 10.1145/226643.226647. URL: http://doi.acm.org/10.1145/226643.226647.

[Cho+01] G. V. Chockler, I. Keidar, and R. Vitenberg. "Group communication specifications: a comprehensive study". In: *ACM Comput. Surv.* 33.4 (Dec. 2001), pp. 427–469. ISSN: 0360-0300. DOI: 10.1145/503112.503113. URL: http://doi.acm.org/10.1145/503112.503113.

[Cou+09] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. "D2STM: Dependable Distributed Software Transactional Memory". In: *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. PRDC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 307–313. ISBN: 978-0-7695-3849-5. DOI: 10.1109/PRDC.2009.55. URL: http://dx.doi.org/10.1109/PRDC.2009.55.

[Cou+11] M. Couceiro, P. Romano, and L. Rodrigues. "PolyCert: polymorphic self-optimizing replication for in-memory transactional grids". In: *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*. Middleware'11. Lisbon, Portugal: Springer-Verlag, 2011, pp. 309–328. ISBN: 978-3-642-25820-6. DOI: 10.1007/978-3-642-25821-3_16. URL: http://dx.doi.org/10.1007/978-3-642-25821-3_16.

[Cou+05] C. Coulon, E. Pacitti, and P. Valduriez. "Consistency Management for Partial Replication in a High Performance Database Cluster". In: *Proceedings of the 11th International Conference on Parallel and Distributed Systems - Volume 01*. ICPADS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 809–815. ISBN: 0-7695-2281-5-01. DOI: 10.1109/ICPADS.2005.114. URL: http://dx.doi.org/10.1109/ICPADS.2005.114.

[Déf+04] X. Défago, A. Schiper, and P. Urbán. "Total order broadcast and multicast algorithms: Taxonomy and survey". In: *ACM Comput. Surv.* 36.4 (Dec. 2004), pp. 372–421. ISSN: 0360-0300. DOI: 10.1145/1041680.1041682. URL: http://doi.acm.org/10.1145/1041680.1041682.

[Dia+11] R. J. Dias, J. M. Lourenço, and N. Preguiça. "Efficient and Correct Transactional Memory Programs Combining Snapshot Isolation and Static Analysis". In: *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism (HotPar'11)*. HotPar'11. Usenix Association. Usenix Association, May 2011.

[Dia+12] R. J. Dias, T. M. Vale, and J. M. Lourenço. "Efficient support for in-place metadata in transactional memory". In: *Proceedings of the 18th international conference on Parallel Processing*. Euro-Par'12. Rhodes Island, Greece: Springer-Verlag, 2012, pp. 589–600. ISBN: 978-3-642-32819-0. DOI: 10.1007/978-3-642-32820-6_59. URL: http://dx.doi.org/10.1007/978-3-642-32820-6_59.

[Dia+13] R. J. Dias, T. M. Vale, and J. M. Lourenço. "Efficient support for in-place metadata in Java software transactional memory". In: *Concurrency and Computation: Practice and Experience* 25.17 (Dec. 2013), pp. 2394–2411. ISSN: 1532-0634. DOI: 10.1002/cpe.3098. URL: http://dx.doi.org/10.1002/cpe.3098.

[Dic+06] D. Dice, O. Shalev, and N. Shavit. "Transactional locking II". In: *international conference on Distributed Computing*. 2006, pp. 194–208. ISBN: 3-540-44624-9, 978-3-540-44624-8. DOI: 10.1007/11864219_14. URL: http://dx.doi.org/10.1007/11864219_14.

[Esw+76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. "The notions of consistency and predicate locks in a database system". In: *Commun. ACM* 19.11 (Nov. 1976), pp. 624–633. ISSN: 0001-0782. DOI: 10.1145/360363.360369. URL: http://doi.acm.org/10.1145/360363.360369.

[Fst] *fst - fast serialization for Java*. http://code.google.com/p/fast-serialization/. 2013.

[Gam+94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994, p. 416. ISBN: 0201633612.

[Gra78] J. Gray. "Notes on Data Base Operating Systems". In: *Operating Systems, An Advanced Course*. London, UK, UK: Springer-Verlag, 1978, pp. 393–481. ISBN: 3-540-08755-9. URL: http://dl.acm.org/citation.cfm?id=647433.723863.

[GR92] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902.

[GK08] R. Guerraoui and M. Kapalka. “On the correctness of transactional memory”. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. PPoPP ’08. Salt Lake City, UT, USA: ACM, 2008, pp. 175–184. ISBN: 978-1-59593-795-7. DOI: 10.1145/1345206.1345233. URL: http://doi.acm.org/10.1145/1345206.1345233.

[HR83] T. Haerder and A. Reuter. “Principles of transaction-oriented database recovery”. In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: http://doi.acm.org/10.1145/289.291.

[HP86] R. Hansdah and L. Patnaik. “Update serializability in locking”. In: *ICDT ’86*. Ed. by G. Ausiello and P. Atzeni. Vol. 243. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1986, pp. 171–185. ISBN: 978-3-540-17187-4. DOI: 10.1007/3-540-17187-8_36. URL: http://dx.doi.org/10.1007/3-540-17187-8_36.

[Har+10] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. 2nd. Morgan and Claypool Publishers, 2010. ISBN: 1608452352, 9781608452354.

[HM93] M. Herlihy and J. E. B. Moss. “Transactional memory: architectural support for lock-free data structures”. In: *SIGARCH Comput. Archit. News* 21.2 (May 1993), pp. 289–300. ISSN: 0163-5964. DOI: 10.1145/173682.165164. URL: http://doi.acm.org/10.1145/173682.165164.

[Her+06] M. Herlihy, V. Luchangco, and M. Moir. “A flexible framework for implementing software transactional memory”. In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 253–262. ISBN: 1-59593-348-4. DOI: 10.1145/1167473.1167495. URL: http://doi.acm.org/10.1145/1167473.1167495.

[HW90] M. P. Herlihy and J. M. Wing. “Linearizability: a correctness condition for concurrent objects”. In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: http://doi.acm.org/10.1145/78969.78972.

[KA98] B. Kemme and G. Alonso. “A suite of database replication protocols based on group communication primitives”. In: *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*. May 1998, pp. 156–163. DOI: 10.1109/ICDCS.1998.679498.

[Kor+10] G. Korland, N. Shavit, and P. Felber. “Deuce: Noninvasive software transactional memory in Java”. In: *Transactions on HiPEAC* 5.2 (2010).

[Kot+08] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. "DiSTM: A Software Transactional Memory Framework for Clusters". In: *Proceedings of the 2008 37th International Conference on Parallel Processing*. ICPP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 51–58. ISBN: 978-0-7695-3374-2. DOI: 10.1109/ICPP.2008.59. URL: http://dx.doi.org/10.1109/ICPP.2008.59.

[Kry] *kryo - Fast, efficient Java serialization and cloning*. http://code.google.com/p/kryo/. 2013.

[LS96] R. G. Lavender and D. C. Schmidt. "Pattern languages of program design 2". In: ed. by J. M. Vlissides, J. O. Coplien, and N. L. Kerth. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. Chap. Active object: an object behavioral pattern for concurrent programming, pp. 483–499. ISBN: 0-201-895277. URL: http://dl.acm.org/citation.cfm?id=231958.232967.

[Lom77] D. B. Lomet. "Process structuring, synchronization, and recovery using atomic actions". In: *SIGSOFT Softw. Eng. Notes* 2.2 (Mar. 1977), pp. 128–137. ISSN: 0163-5948. DOI: 10.1145/390019.808319. URL: http://doi.acm.org/10.1145/390019.808319.

[Man+06] K. Manassiev, M. Mihailescu, and C. Amza. "Exploiting distributed version concurrency in a transactional memory cluster". In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '06. New York, New York, USA: ACM, 2006, pp. 198–208. ISBN: 1-59593-189-9. DOI: 10.1145/1122971.1123002. URL: http://doi.acm.org/10.1145/1122971.1123002.

[Min+08] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. "STAMP: Stanford Transactional Applications for MultiProcessing". In: *IEEE International Symposium on Workload Characterization*. 2008, pp. 35–46. DOI: 10.1109/IISWC.2008.4636089.

[Mir+01] H. Miranda, A. Pinto, and L. Rodrigues. "Appia, a flexible protocol kernel supporting multiple coordinated channels". In: *Proceeding of the 21st International Conference on Distributed Computing Systems*. 2001, pp. 707–710. DOI: 10.1109/ICDSC.2001.919005.

[MG08] K. F. Moore and D. Grossman. "High-level small-step operational semantics for transactions". In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '08. San Francisco, California, USA: ACM, 2008, pp. 51–62. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328448. URL: http://doi.acm.org/10.1145/1328438.1328448.

[Ora13] Oracle. *The* `java.lang.instrument` *package*. http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html. May 2013.

[OW213] OW2 Consortium. *ASM*. http://asm.ow2.org. May 2013.

[Pal+10] R. Palmieri, F. Quaglia, and P. Romano. "AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing". In: *Proceedings of the 2010 Ninth IEEE International Symposium on Network Computing and Applications*. NCA '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 20–27. ISBN: 978-0-7695-4118-1. DOI: 10.1109/NCA.2010.10. URL: http://dx.doi.org/10.1109/NCA.2010.10.

[PS98] F. Pedone and A. Schiper. "Optimistic Atomic Broadcast". In: *Distributed Computing*. Ed. by S. Kutten. Vol. 1499. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 318–332. ISBN: 978-3-540-65066-9. DOI: 10.1007/BFb0056492. URL: http://dx.doi.org/10.1007/BFb0056492.

[Pel+12a] S. Peluso, P. Romano, and F. Quaglia. "SCORe: A Scalable One-Copy Serializable Partial Replication Protocol". In: *Middleware 2012*. Ed. by P. Narasimhan and P. Triantafillou. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 456–475. ISBN: 978-3-642-35169-3. DOI: 10.1007/978-3-642-35170-9_23. URL: http://dx.doi.org/10.1007/978-3-642-35170-9_23.

[Pel+12b] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. "When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication". In: *Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems*. ICDCS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 455–465. ISBN: 978-0-7695-4685-8. DOI: 10.1109/ICDCS.2012.55. URL: http://dx.doi.org/10.1109/ICDCS.2012.55.

[Red13] Red Hat. *JBoss Application Server*. http://www.jboss.org/jbossas. 2013.

[Rie+06] T. Riegel, C. Fetzer, and P. Felber. "Snapshot isolation for software transactional memory". In: *Proceedings of the first ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. TRANSACT '06. Ottowa, Ontario, Canada: ACM, 2006.

[Rod+06] L. Rodrigues, J. Mocito, and N. Carvalho. "From spontaneous total order to uniform total order: different degrees of optimistic delivery". In: *Proceedings of the 2006 ACM symposium on Applied computing*. SAC '06. Dijon, France: ACM, 2006, pp. 723–727. ISBN: 1-59593-108-2. DOI: 10.1145/1141277.1141441. URL: http://doi.acm.org/10.1145/1141277.1141441.

[Rom+08] P. Romano, N. Carvalho, and L. Rodrigues. "Towards distributed software transactional memory systems". In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. LADIS '08. Yorktown Heights, New York: ACM, 2008, 4:1–4:4. ISBN: 978-1-60558-296-2. DOI: 10.1145/1529974.1529980. URL: http://doi.acm.org/10.1145/1529974.1529980.

[Rui11] P. Ruivo. "Replicação Parcial para Sistemas de Memória Transaccional por Software". MA thesis. Instituto Superior Técnico, Universidade Técnica de Lisboa, 2011.

[SR11] M. M. Saad and B. Ravindran. "HyFlow: a high performance distributed software transactional memory framework". In: *Proceedings of the 20th international symposium on High performance distributed computing*. HPDC '11. San Jose, California, USA: ACM, 2011, pp. 265–266. ISBN: 978-1-4503-0552-5. DOI: 10.1145/1996130.1996167. URL: http://doi.acm.org/10.1145/1996130.1996167.

[Sch+06] N. Schiper, R. Schmidt, and F. Pedone. "Optimistic Algorithms for Partial Database Replication". In: *Principles of Distributed Systems*. Vol. 4305. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 81–93. ISBN: 978-3-540-49990-9. DOI: 10.1007/11945529_7. URL: http://dx.doi.org/10.1007/11945529_7.

[Sch+10] N. Schiper, P. Sutra, and F. Pedone. "P-Store: Genuine Partial Replication in Wide Area Networks". In: *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*. SRDS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 214–224. ISBN: 978-0-7695-4250-8. DOI: 10.1109/SRDS.2010.32. URL: http://dx.doi.org/10.1109/SRDS.2010.32.

[Sch90] F. B. Schneider. "Implementing fault-tolerant services using the state machine approach: a tutorial". In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167. URL: http://doi.acm.org/10.1145/98163.98167.

[Ser+07] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme. "Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation". In: *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*. PRDC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 290–297. ISBN: 0-7695-3054-0. DOI: 10.1109/PRDC.2007.39. URL: http://dx.doi.org/10.1109/PRDC.2007.39.

[ST95] N. Shavit and D. Touitou. "Software transactional memory". In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. PODC '95. Ottowa, Ontario, Canada: ACM, 1995, pp. 204–213. ISBN: 0-89791-710-3. DOI: 10.1145/224964.224987. URL: http://doi.acm.org/10.1145/224964.224987.

[Sil+13] J. A. Silva, T. M. Vale, J. M. Lourenço, and H. Paulino. "Replicação Parcial com Memória Transacional Distribuída". In: *INForum 2013: Proceedings of INForum Simpósio de Informática*. Universidade de Évora, 2013.

[Ske82] D. Skeen. "A Quorum-Based Commit Protocol". In: *Berkeley Workshop*. 1982, pp. 69–80.

[Sou+01] A. Sousa, R. Oliveira, F. Moura, and F. Pedone. "Partial Replication in the Database State Machine". In: *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*. NCA '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 298–. ISBN: 0-7695-1432-4. URL: http://dl.acm.org/citation.cfm?id=580585.883102.

[Sun13] Sun Microsystems, Inc. *sun.misc.Unsafe*. http://www.docjar.com/docs/api/sun/misc/Unsafe.html. May 2013.

[Tra13] Transaction Processing Performance Counsil. *TPC Benchmark W*. http://www.tpc.org/tpcw. May 2013.

[Val12] T. Vale. "A Modular Distributed Transactional Memory Framework". MA thesis. Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2012.