INSTITUTO
SUPERIOR
TÉCNICO

UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

A Generic and Distributed Dependable Software Transactional Memory

Nuno Miguel Rei Carvalho

Supervisor: Professor Luís Eduardo Teixeira Rodrigues

Thesis approved in public session to obtain the PhD degree in
Information Systems and Computer Engineering

Jury final classification: Pass with Merit

Jury

Chairperson: Chairman of the IST Scientific Board
Members of the Committee:
Doctor Pascal Amédée Bernard Felber
Doctor Luís Eduardo Teixeira Rodrigues
Doctor José Carlos Alves Pereira Monteiro
Doctor João Manuel dos Santos Lourenço

2011

INSTITUTO
SUPERIOR
TÉCNICO

UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

A Generic and Distributed Dependable Software Transactional Memory

Nuno Miguel Rei Carvalho

Supervisor: Professor Luís Eduardo Teixeira Rodrigues

Thesis approved in public session to obtain the PhD degree in
Information Systems and Computer Engineering

Jury final classification: Pass with Merit

Jury

Chairperson: Chairman of the IST Scientific Board
Members of the Committee:
Doctor Pascal Amédée Bernard Felber, Full Professor, Université de Neuchâtel
Doctor Luís Eduardo Teixeira Rodrigues, Full Professor, Universidade Técnica de Lisboa
Doctor José Carlos Pereira Monteiro, Associate Professor, Universidade Técnica de Lisboa
Doctor João Manuel dos Santos Lourenço, Assistant Professor, Universidade Nova de Lisboa

Funding Institutions

Fundação para a Ciência e Tecnologia

2011

à Ana

ao João Pedro

Resumo

A Memória Transaccional por Software (STM) é uma abstracção que pretende facilitar o desenvolvimento de programas concorrentes. Ao usar uma STM o programador não é obrigado a especificar explicitamente o controlo de concorrência através de mecanismos de sincronização elementares, tais como os trincos. Pelo contrário, o programador apenas necessita de identificar quais as sequências de instruções que necessitam de executar com isolamento, recorrendo ao conceito de transacção. O controlo de concorrência é então realizado de forma automática.

Esta dissertação aborda o problema de desenvolver STMs distribuídas e tolerantes a faltas. Uma STM distribuída suporta a abstracção de um espaço de endereçamento global que pode ser acedido por fios de execução em máquinas diferentes. Uma STM tolerante a faltas mantém diversas cópias dos dados, mutuamente coerentes, de forma a que não se perca informação quando falha uma máquina. Estas características são necessárias para satisfazer os requisitos impostos por sistemas STMs em produção (tal como o sistema FénixEDU).

Um dos problemas chave na gestão de dados replicados é a redução do custo inerente à coordenação entre réplicas. Considerando que o custo de executar transacções em memória é significativamente mais baixo do que noutros contextos (como por exemplo em sistemas de gestão de bases de dados), existe o risco destes custos de coordenação dominarem o desempenho do sistema. Assim, esta dissertação propõe vários novos protocolos de gestão da replicação adequados ao desenvolvimento de STMs distribuídas tolerantes a faltas.

Os resultados aqui relatados indiciam que, com grande probabilidade, não é possível desenvolver um protocolo de replicação que ofereça um desempenho óptimo para a grande diversidade de padrões de carga que caracterizam os ambientes baseados em STM. Desta forma, a dissertação propõe também uma arquitectura genérica que permite suportar na mesma STM vários protocolos de replicação. Esta arquitectura estabelece a base necessária para o desenvolvimento de STMs autonómicas com a capacidade de oferecer um desempenho adequado numa vasta gama de cenários.

Abstract

A Software Transactional Memory (STM) is an abstraction that aims at simplifying the development of concurrent programs. When using STMs, the programmers are not required to manage explicitly concurrency control, for instance, by using low-level synchronization mechanisms such as locks. Instead, programmers only need to identify the sequences of operations that need to be executed in isolation, using the concept of a transaction. Concurrency control is then performed by the runtime support, in a transparent manner to the programmer.

This thesis addresses the problem of implementing distributed dependable STMs. A distributed STM provides the abstraction of a global address space, that can be accessed from threads in different nodes as a local STM. Furthermore, a dependable STM ensures that multiple copies of the data are maintained, and kept consistent, such that data is not lost if a node fails. These features need to be added to STMs to address the high availability and scalability requirements posed by realistic production environments (e.g. the FénixEDU system).

One of the most challenging problems in the management of replicated data is to reduce the cost of preserving replica consistency, a task that requires coordination among replicas. Given that the costs of executing an in-memory transaction are much smaller than executing transactions in other settings (such as in database systems), there is the risk that coordination costs become prohibitively expensive. Therefore, the thesis proposes several novel replication protocols suitable for building dependable distributed STMs.

The results reported in the thesis, show that it is unlikely that a single replication protocol can outperform all the other protocols, for all workloads that characterize STMs environments. Therefore, the thesis also proposes a generic architecture that allows multiple replication protocols to coexist in a seamless manner in the same STM. This architecture opens the door to build adaptive solutions that can dynamically and automatically select the best replication protocol for a given deployment and workload, thus paving the way to the implementation of autonomic distributed dependable STMs that offer good performance in a wide range of scenarios.

Palavras Chave

Sistemas Distribuídos

Memória Transaccional em Software

Tolerância a Faltas

Replicação de Dados

Coerência de Dados

Computação Paralela

Comunicação em Grupo

Controlo de Concorrência

Sistemas Configuráveis

Middleware

Keywords

Distributed Systems

Software Transactional Memory

Fault Tolerance

Data Replication

Data Consistency

Parallel Computing

Group Communication

Concurrency Control

Configurable Systems

Middleware

Agradecimentos

Em primeiro lugar, os meus agradecimentos são dirigidos ao meu professor e orientador Luís Eduardo Teixeira Rodrigues. A sua dedicação e profissionalismo foram imprescindíveis para o sucesso deste trabalho. Em segundo lugar, quero dirigir os meus agradecimentos ao Paolo Romano, pelo seu empenho, pela sua ajuda e pelas discussões que geraram muitas ideias descritas neste trabalho.

Quero agradecer também aos restantes colegas do Grupo de Sistemas Distribuídos do INESC-ID, João Leitão, Liliana Rosa, José Mocito, Diogo Mónica, João Paiva e Maria Couceiro. Pelo excelente trabalho de equipa, pela amizade, pelo companheirismo e pela disponibilidade que sempre tiveram para comigo.

Agradeço ainda ao David Matos, pelo template de LaTeX que, felizmente, teima em perdurar no tempo.

Agora quero dirigir algumas palavras à pessoa que mais me tem apoiado no último ano e meio. Quero aqui expressar os meus mais sinceros agradecimentos à Ana Inácio. A mulher que amo. A mulher com quem vivo e quero viver o resto da minha vida. A pessoa que me apoia e ajuda sempre, incondicionalmente, em todas as decisões que tomo, em cada cruzamento, em cada obstáculo. Obrigado.

Quero agora redirecionar os meus agradecimentos a todos os meus amigos, aqueles que sempre estiveram lá para me apoiar. Aqueles que me dirigem uma palavra amiga nos momentos difíceis. À Lara Santos, ao José Adonis, ao Miguel Martinho, ao João Abreu, à Alexandra Oliveira, ao Hugo Ortolá, e aos muitos outros que, mesmo não estando citados neste parágrafo, estão gravados na minha memória e no meu coração.

Não quero terminar estes agradecimentos sem dedicar algumas palavras à minha família. A compreensão e paciência que sempre tiveram para comigo, assim como o apoio que sempre me deram, não tem qualquer preço. Obrigado, João Pedro, pelos 7 anos de sorrisos e alegria que deliciam qualquer ser humano.

Finalmente, quero agradecer ao Instituto Superior Técnico, ao INESC-ID e ao Grupo de Sistemas Distribuídos pelas instalações e acolhimento, que me permitiram ter as condições necessárias para desenvolver este trabalho. Este trabalho foi parcialmente suportado pela Fundação para a Ciência e Tecnologia, através da bolsa de Doutoramento (SFRH / BD / 35880 / 2007) e pelo projecto Aristos (PTDC/EIA- EIA/102496/2008). Este trabalho foi ainda parcialmente suportado pela Comissão Europeia, através do projecto STReP Cloud-TM (contracto número 257784, do sétimo programa quadro).

Lisboa, Julho de 2011

Nuno Miguel Rei Carvalho

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The current state of technology in computer architecture and processor design has lead to the pervasive adoption of multi-core CPUs. In turn, this has increased the need for tools that simplify the development of parallel applications, that can make full use of the availability of multiple cores.

A Software Transactional Memory (STM) (Adl-Tabatabai, Kozyrakis, & Saha 2006; Cachopo & Rito-Silva 2006; Korland, Shavit, & Felber 2009) is an abstraction that aims at simplifying the development of concurrent programs. When using STMs, the programmers are not required to manage explicitly concurrency control, for instance, by using low-level synchronization mechanisms such as spin-locks, locks or semaphores. Instead, programmers only need to identify the sequences of operations that are required to be executed in isolation, using the concept of a transaction. This concept is being used with large success for decades in database management systems. Concurrency control is then performed by the runtime support, in a transparent manner to the programmer, ensuring that transactions that commit do not violate the consistency constraints specified for the system, avoiding problems such as deadlocks and priority inversions.

The use of STM based systems has been maturing quickly and there are today notable examples of systems based in this technology that have been deployed and are used in production. A relevant example is the FénixEDU system, an infrastructure to support web applications that relies on a STM-based solution in order to ensure the consistency of an in memory middle-tier object cache. The current version of the FénixEDU system is facing scalability and dependability challenges, as it has to process between 1,000,000 and 4,500,000 transactions per day for a population of 12000 students, 900 faculty and 800 administrative members of the Instituto Superior Técnico (IST[1]).

To face the demanding scalability and dependability requirements imposed by these en-

[1]`http://www.ist.utl.pt`

terprise systems, STMs need to be distributed and augmented with fault-tolerance mechanisms (Carvalho, Cachopo, Rodrigues, & Silva 2008). A distributed STM provides the abstraction of a global address space, that can be accessed from threads in different nodes as a local STM. By allowing the data to be maintained and accessed in different nodes, one may address the scalability requirements of modern STMs. Furthermore, a fault-tolerant STM ensures that multiple copies of the data are maintained, and kept consistent, such that data is not lost if a node fails. This addresses the dependability requirements, as higher availability and reliability can be achieved.

This thesis addresses the problems of designing and implementing *distributed dependable Software Transactional Memories.*

1.1 Problem Statement

One of the most challenging problems in the management of replicated data is to reduce the cost of preserving replica consistency, a task that requires coordination among replicas. This problem is exacerbated in STM systems. In fact the costs of executing in-memory transactions are much smaller than executing transactions in other settings (such as in database systems). As a result, coordination costs in STMs may easily become prohibitively expensive in relative terms.

Also, experience with data replication in different contexts has shown that the performance of replication protocols is highly dependent of the workload characterization and the characteristics of deployment scenarios (Pedone, Guerraoui, & Schiper 2003; Patiño Martínez, Jiménez-Peris, Bettina, & Alonso 2000; Cecchet, Marguerite, & Zwaenepole 2004; Agrawal, Alonso, Abbadi, & Stanoi 1997). This may also prove to be a challenge in STM systems that are faced with highly heterogeneous workloads (Romano, Carvalho, & Rodrigues 2008).

Therefore, the key problem that this thesis addresses is the following:

> *Is it possible to implement distributed and dependable STMs, that can provide acceptable performance under different workloads and deployment scenarios?*

To answer this problem, this thesis proposes solutions that leverage on the experience gained in the following related areas:

- Distributed Shared Memory systems;
- Software Transactional Memory systems;
- Replication algorithms for Database Management systems.

1.2 Contributions

As a result of the quest for replication protocols suitable for STMs environments, this thesis proposes two novel replication protocols that provide good performance for different workloads and system configurations. The results obtained with these protocols, and with protocols developed by other members of the Distributed Systems Group at INESC-ID, show that it is unlikely that a single replication protocol can outperform all the other protocols, for all workloads that characterize STMs environments. Therefore, the thesis also proposes a generic architecture that allows multiple replication protocols to coexist in a seamless manner in the same STM. In summary, the contributions of the thesis are as follows:

- A lease based replication protocol that increases the throughput of the system in face of workloads with data locality;
- A speculative replication protocol that increases the performance of the system in scenarios with stable network latencies;
- A generic and configurable architecture that can be tuned to optimize the performance of an STM system for different scenarios and workloads.

1.3 Results

Given the contributions listed above, the results of this thesis are twofold:

- An implementation of the architecture and of the replication protocols;
- An experimental evaluation of each instance of the architecture with heterogeneous workloads.

These results open the door to build adaptive solutions that will be able to dynamically and automatically select the best replication protocol for a given deployment and workload, thus paving the way to the implementation of autonomic distributed dependable STMs that offer good performance in a wide range of scenarios.

1.4 Research History

This work leverages on my research background on distributed systems and consistency protocols. My first work on distributed systems addressed group communication and protocol composition frameworks. On 2004 I integrated the research team of the GORDA[2] – Open Replication of Databases – european project. In this project, I have contributed to an architecture where DBMSs expose their processing stages to external components to allow efficient database replication.

After the GORDA project (in 2008), I have joined the Distributed Systems Group at INESC-ID and gained contact with the FénixEDU system, maybe one of the first production systems based on a STM (more precisely, JVSTM). I have started the current work by studying how the scalability problems of the FénixEDU system could be solved using the replication solutions already used in DBMSs. This study has been published by Carvalho, Cachopo, Rodrigues, & Silva (2008) and motivated the implementation of an initial prototype of a replicated STM based on JVSTM and standard certification based replication protocols. In cooperation with other team members, I was able to make a comparative study that highlights the differences between database and STM transactions, in (Romano, Carvalho, & Rodrigues 2008). This study gave us some hints on how the replication protocols could be improved to cope with the needs of STM transactions.

The study above has shown that the amount of time that a replicated STM spends on the commit procedure can be reduced by reducing the group communication message sizes. This issue was addressed in the scope of the Master Thesis of Maria Couceiro, using Bloom Filters to reduce the message sizes at the cost of a configurable abort rate, using a new certification scheme called Bloom Filter Certification (BFC). BFC was implemented in an improved version of the first prototype, leading to the Dependable and Distributed STM (D^2STM) system, published by

[2]`http://gorda.di.uminho.pt`

Couceiro, Romano, Carvalho, & Rodrigues (2009). I have contributed to that system with the architectural components that supported the execution of the BFC protocol. With this work we have identified the need to build a composable architecture that could be used to test different protocols, not only with JVSTM, but also with other STM systems. I also got insights on what a STM needs to export (using reflexive interfaces) in order to allow efficient replication of any STM system. This allowed to generate an improved version of the architecture (Carvalho 2010; Carvalho, Romano, & Rodrigues 2011a).

With this new architecture we started to explore new certification based replication protocols. One of the main contributions of this thesis is the Asynchronous Lease-based Certification protocol (Carvalho, Romano, & Rodrigues 2010) that uses data locality to improve the performance of the system by avoiding the use of heavy communication procedures, such as Atomic Broadcast. The second contribution of this thesis with regard to replication protocols is the Speculative Certification (Carvalho, Romano, & Rodrigues 2011b). This protocol leverages on the optimistic deliveries of Atomic Broadcast protocols to create an overlap between transaction executions and communication, and also to create a chain of speculative transactions. This thesis shows that both protocols are able to achieve good performance in different workloads.

1.5 Structure of the Document

This document is organized as follows. Chapter 2 describes the systems that served as inspiration to this work, with emphasis on the memory consistency models. Chapter 3 describes the related work on distributed Software Transactional Memory (STM) systems and discusses the challenges of building fully replicated STMs. Chapters 4 and 5 describe the proposed replication protocols tailored for STMs. Chapter 6 provides a comparison study of different protocols and workloads, and describes the generic architecture used to test and compose the proposed protocols with different workloads. Finally, Chapter 7 concludes this document.

2 Background

The work presented in this dissertation is inspired by three complementary lines of work as enumerated below:

- Software Transactional Memory (STM) systems that, with few exceptions, have been designed and implemented for non-distributed systems. These works have introduced the STM paradigm, identified the relevant consistency criteria in this context, and propose different concurrency control mechanisms. However, at the start of this work, replicated STMs have not been addressed in the literature.
- Distributed Shared Memory (DSM) systems, that aim at providing the abstraction of a global shared address space in a distributed system. Most of these systems were not integrated with transactional mechanisms and did not address concurrency control or considered explicit locking mechanisms.
- Database Management Systems (DBMS) that support both transactions, distribution, and replication, but in a controlled setting where applications access the data via a dedicated query language and in a context where durability is a primary concern.

The following sections summarize the key concepts and results from these related works.

2.1 Distributed Shared Memory

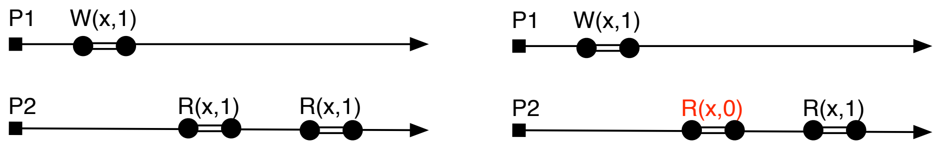
In a Distributed Shared Memory (DSM) system, the address space of a process is not limited to the memory available in the local node, it aggregates memory shared by different nodes. A DSM system can be modeled by a set of (single-threaded) processes that execute on different machines and access the global address space by reading and writing memory positions. Each node has a local memory, and the access to those local memories needs to be coordinated to provide a consistent view of the global address space. It is worth noting that this coordination problem is not restricted to distributed systems, as it also appears in multi-core processor that access a shared memory, since data may reside in the cache of each individual core (typically, in those systems coordination is ensured by the hardware).

The expected behavior of the memory system is defined by a *memory consistency model.* The most important consistency models are summarized next. For each model, examples of valid and invalid executions are included, to better illustrate what a programmer can expect when using a certain consistency model. In these examples, $W(x, v)$ denotes a *write* operation of the value v to variable x and $R(x, v)$ denotes a *read* operation of the variable x, returning the value v. Without loss of generality, the examples assume that the initial value of the variables is always 0 (zero).

2.1.1 Memory Consistency Models

A memory consistency model is a contract between processes and the middleware that provides the DSM. It specifies the rules that processes need to obey when accessing the data, and the consistency guarantees provided by the middleware. The middleware exports operations to read and write data and, sometimes, also synchronization operations. Depending on the memory consistency model, the set of possible outcomes for the same (partially ordered) set of operations may be different.

Probably the most intuitive memory model is the one that matches the operation of an idealized single memory copy, where each position is accessed in an atomic manner. The models that approximate this behavior are often named strong consistency models. In opposition, models where the programmer becomes aware that the memory internal organization departs from the idealized single copy are named relaxed consistency models. Naturally, the implementations of strict consistency models require more message exchanges and synchronization points, which are often the cause of performance degradation. Weak models aim at improving the performance at the cost of exhibiting a less intuitive behavior.



(a) Valid execution. (b) Invalid execution.

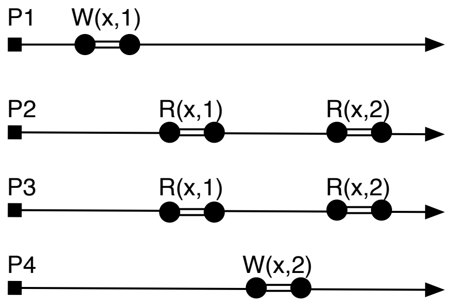
Figure 2.1: Strict consistency example.

Strong Consistency Models

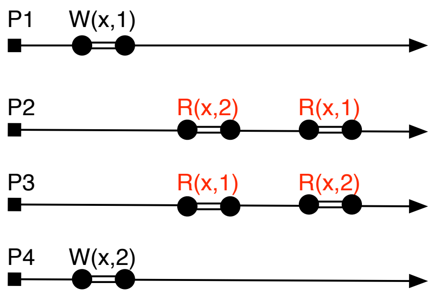
Strict consistency The strict consistency model is defined by the following properties: (*i*) any read to a memory location X returns the value stored by the most recent (completed) write operation to X; (*ii*) the execution of concurrent write operations is equivalent to a serial execution of those operations. The Figure 2.1 depicts two executions of read and write operations. The execution shown in the Figure 2.1(a) is valid under strict consistency, since subsequent *read* operations always return the latest written value. The execution depicted in the Figure 2.1(b) is not valid because there is a *read* operation issued after the latest *write* operation that does not return the latest written value.

This is the most intuitive model to use, because the programmer can invoke a read operation at any time and always get the most recently written value. However, the implementation of this model requires that any pair of read and write operations must coordinate, to ensure that the latest written value is returned by the reader, which is very inefficient in a distributed system.

Linearizability Linearizability (Herlihy & Wing 1990) requires that all operations appear to have been executed in some sequential order that is consistent with the global ordering of non-overlapping operations. Furthermore, each operation must appear to take place instantaneously at some time between its invocation and response. Linearizability is a compositional (or local) consistency model: a system with several objects is Linearizable if and only if it is Linearizable with respect to each object. This is important because it allows concurrent applications to be designed and verified in a modular way. Linearizability is stronger and more difficult to achieve than other consistency models that will be introduced later in the text, but the composability property makes it more attractive than strict consistency.



(a) Valid serial execution: $W_1(1)$, $R_2(1)$, $R_3(1)$, $W_4(2)$, $R_2(2)$, $R_3(2)$.



(b) Invalid execution: no serial equivalence.

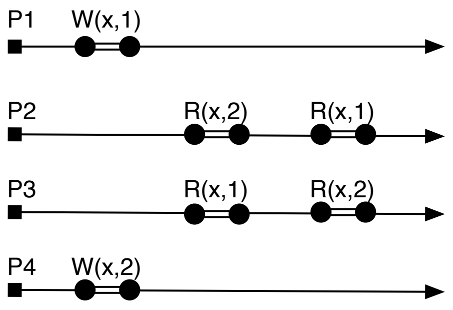
Figure 2.2: Sequential consistency example.

Sequential consistency The sequential consistency model was introduced by Lamport (1979) and it is defined as follows: a system is sequentially consistent if the result of any execution is the same as if the operations of all the processes were executed in a serial order, and the operations of each individual process appear in this sequence in the order specified by its program. This means that the system provides sequential consistency if every node of the system reads and writes on each memory location in the same order. The Figure 2.2 shows two executions of *read* and *write* operations. The execution depicted in the Figure 2.2(a) is valid, since it is equivalent to the serial execution $W_1(1)$, $R_2(1)$, $R_3(1)$, $W_4(2)$, $R_2(2)$, $R_3(2)$. The Figure 2.2(b) shows an execution that is not valid, since there is no similar execution with similar results.

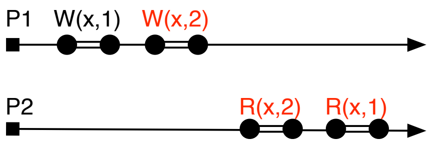
The sequential consistency is weaker than strict consistency, but also a very intuitive consistency model. This simplicity is one more time achieved at cost of efficiency, since all the write operations must be totally ordered, for instance, by being propagated using a total order primitive, such as Atomic Broadcast (Guerraoui & Rodrigues 2006).

Relaxing Consistency

The previously described memory consistency models are easy to understand by the programmers, but are also very costly to implement. Because of this, several consistency models were defined that allows to improve the system performance at the cost of relaxing consistency. Some of these models are described in the next paragraphs.



(a) Valid execution.



(b) Invalid execution.

Figure 2.3: Processor consistency example.

Processor consistency Goodman (1989) relaxes some ordering constraints imposed by the sequential consistency based on the order that processors access memory in a scenario where no caches are present. A *von Neumann* processor has an implied order in which it access memory with no caches. On a single processor, the *program order* implies that a read operation returns the value most recently written (to the same memory location). For a multi-processor, the order in which memory operations occur may be observed by other processors to achieve implicit synchronization. Based on this, Goodman defined the processor consistency model: write operations done by a single processor are received by all other processors in the order in which they were issued, but write operations from different processors may be seen in a different order by different processors. The Figure 2.3 shows two execution examples of read and write operations. The example depicted in the Figure 2.3(a) is the same example shown in the Figure 2.2(b) that was invalid in the sequential consistency model. In the case of processor consistency, concurrent updates can be seen by other processes in a different order. The example shown in the Figure 2.3(b) is invalid because $P2$ sees the updates of $P1$ in a different order than the one they were issued.

The basic idea of processor consistency is to better accommodate networks in which the latency between different nodes may be different. It relaxes the sequential consistency model but improves the system performance, since processes do not have to coordinate to achieve a global order of all write operations that occur in the system. Instead, it is possible to implement the model using a FIFO reliable broadcast (Guerraoui & Rodrigues 2006) for each write operation. The programmers must have an extra care when programming on top of this consistency model,

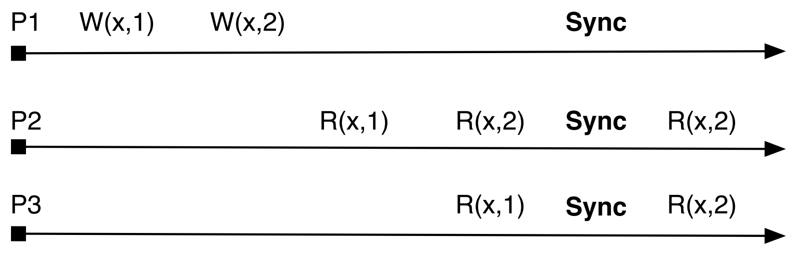


Figure 2.4: Weak consistency example.

since the last memory update that has been received may not be the last value written.

Causal Consistency This model was introduced by Ahamad, Neiger, Burns, Kohli, & Hutto (1995) and it is based on the causal order of read/write operations. The operations that are causally related must be seen in causal order by all processes. When a process performs a read operation followed later by a write operation, even on a different variable, the first operation is said to be causally ordered before the second, because the value stored by the write operation may have been dependent upon the result of the read operation. Two write operations are concurrent if they are not causally ordered. Concurrent operations can be seen in different orders by different processes. This solution allows for more efficient implementations than the previous models, even if the implementations must keep track of the causal relations among operations.

Weak consistency This model was introduced by Dubois, Scheurich, & Briggs (1998) and relaxes the sequential and processor consistency models by introducing the notion of synchronization points in the program. The programmer cannot make any assumptions about changes to data between two synchronization points. This model improves the system performance because the memory only needs to be consistent at the moment of synchronization. The synchronization operations must be ordered sequentially. This weak consistency model introduces the notion of explicit synchronization: the programmer must invoke synchronization primitives. This is shown in the Figure 2.4, where *Sync* denotes the operation of synchronization issued by the programmer to retrieve the latest updates issued by other processes.

Release consistency Introduced by Gharachorloo, Lenoski, Laudon, Gibbons, Gupta, & Hennessy (1990), release consistency (RC) is similar to weak consistency. Synchronization operations are named: *acquire* and *release*. Performing an acquire ensures that previous write operations to protected variables are made visible locally, and performing a release ensures that local write operations to protected variables are exported, and will be made visible to other processes. There are two variants of the release consistency model, that differ on when (after the release operation) the updated data becomes available to other processes. These two variants are named *Eager* and *Lazy*. Eager protocols (e.g. (Lenoski, Laudon, Gharachorloo, Gupta, & Hennessy 1990)), propagate updates to protected variables immediately on the release operation, or (in the case of invalidation protocols) invalidate other copies immediately. In Lazy protocols (Keleher, Cox, & Zwaenepoel 1992), the consistency-related operations are postponed to the next acquire operation. The authors of this algorithm show that it reduces both the number of messages and the amount of data transferred between processors. This causes also a reduction of false sharing (Torrellas, Lam, & Hennessy 1994), improving the system performance.

Both Lazy and Eager RC provide the same properties as the ones provided by RC. To ensure these properties, the consistency-related operations maintain an order of operations based on the *happened-before* partial order (Adve 1993).

Eventual consistency Eventual Consistency (Gustavsson & Andler 2002) is defined as a specific form of weak consistency: the system ensures that if no new updates are made to the object for a period of time, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays and the load on the system. A widely used system that implements eventual consistency is DNS (Domain Name System). Updates to a name are distributed according to a configured pattern and, eventually, all clients will see the update.

2.1.2 Examples of DSM Systems

There are several systems that implement the DSM abstraction. Some representative systems are described in this section.

IVY One of the first DSM runtime systems was IVY (Li 1988). It was implemented at the Yale University and divides the memory in two classes: private and shared. IVY was implemented using a combination of a user-level library and Operating System (OS) modifications. IVY provides a mechanism for consistency maintenance of shared memory using an invalidation approach on 1 Kbyte pages. Three page management implementations were integrated into IVY: centralized manager scheme, fixed distributed manager scheme and dynamic distributed manager scheme. IVY provides Sequential Consistency (see Section 2.1.1) and uses the write invalidate update protocol (Goodman 1998).

TreadMarks TreadMarks (Keleher, Cox, Dwarkadas, & Zwaenepoel 1994) supports parallel computing on networks of workstations by providing the application with a shared memory abstraction, facilitating the transition from sequential to parallel programs. After identifying possible sources of parallelism in the code, most of the data structures can be retained without change and only synchronization needs to be added to achieve a correct shared memory parallel program.

TreadMarks was implemented using the C programming language, provides shared memory as a linear array of bytes and uses the Release Consistency Model (Gharachorloo, Lenoski, Laudon, Gibbons, Gupta, & Hennessy 1990) (see Section 2.1.1). The implementation uses the virtual memory hardware to detect accesses but it uses a multiple writer protocol (Carter, Bennett, & Zwaenepoel 1995) (that allows multiple nodes to concurrently modify the same page) to alleviate the problems resulting from mismatch between the page size and the application's granularity of sharing. This DSM system is used by the Cluster OpenMP (Intel 2008) system.

Terracotta Terracotta (Terracotta 2009) is an open source clustering software for Java. It delivers clustering as a runtime infrastructure service, which simplifies the task of clustering a Java application, by clustering the Java Virtual Machine (JVM) underneath the application, instead of clustering the application itself. Terracotta's JVM-level clustering can turn single-node, multi-threaded applications into distributed, multi-node applications with minor code changes. It uses standard byte-code manipulation techniques to plug into the Java Memory Model in order to maintain the semantics of Java (Java Language Specification), such as pass-by-reference, thread coordination, and garbage collection across the cluster. Terracotta manages data using the notion of Network-Attached Memory (NAM). NAM enables Terracotta to cluster JVMs directly underneath applications, providing to Java applications both high availability and scalability. In Terracotta, clients connect to a server (or set of servers), where the data is maintained in a persistent store.

2.1.3 Fault-Tolerance

Usually, DSM systems such as (Katsinis & Hecht 2004), rely on Backward Error Recovery (Vounckx, Deconinck, Vounckx, Lauwereins, & Peperstraete 1993) to implement fault tolerance, allowing an application that encounters an error to restart its execution from an earlier, error-free state. This is achieved through the periodic saving of system information (checkpoint), which is restored when an error is detected. For increased fault tolerance, checkpoints can be replicated in the memory of other nodes (Kermarrec, Morin, & Banâtre 1998). This provides the advantages of higher speed and tolerance to multiple node failures when some copies of the checkpoint do not reside on the failed nodes. Recovery algorithms compute the latest global state to which the system can recover on failure. This is called the recovery line. There are several checkpointing algorithms (Morin & Puaut 1997), that can be classified by the following classes:

Coordinated checkpointing Processes coordinate with each other so that, at any time, the set of latest checkpoints for all tasks forms a recovery line. The advantage of this algorithm is that recovery is simple and it keeps only the latest checkpoint of each process. The disadvantage is the fact that coordination is needed to establish such a checkpoint.

Uncoordinated checkpointing Processes take checkpoints independently. The advantage is that checkpointing involves no coordination and hence is simpler. The disadvantage is that recovery is complex. Usually recovery requires maintaining a number of checkpoints for each process as well as a history of the interactions.

Communication Induced Checkpointing Processes take independent checkpoints, but in addition to these they also take additional checkpoints referred to as forced checkpoints. Thus one may view this as introducing some amount of coordination into an uncoordinated checkpointing protocol. The forced checkpoints help making the recovery line progress. Hence it may not be needed to maintain as much history information as in the case of uncoordinated checkpointing.

Network partitioning must also be handled in DSM systems. For instance, in the work of Schöttner, Frenz, Göckelmann & Schulthess (2004) processes must acquire a token to memory pages. The system only allows one token in the system, ensuring that progress is only done in one of the partitions.

2.2 Database Management Systems

A Database Management System (DBMS) controls the organization, storage, retrieval, and integrity of data in a database. It accepts requests from the application and instructs the operating system to transfer the appropriate data between volatile and persistent storage. In relational databases, information is organized in tables, where some records must provide a common field, such as account number, to allow for matching. The requests are made using the Structured Query Language (SQL). For each request, the DBMS must parse it, build an execution plan, optimize this plan, and finally execute it, producing results that can be returned to the client application.

The DBMSs use a programming construct that is now widely known and understood by application programmers: *Transactions.* Transactions in DBMSs provide the following properties: Atomicity, Consistency, Isolation and Durability (ACID).

Atomicity Atomicity refers to the ability of the DBMS to guarantee that either all of the operations of a transaction are performed or none of them are. For example, the transfer of funds from one account to another can be completed or it can fail for different reasons, but atomicity guarantees that one account will not be debited if the other is not credited. Atomicity states that database modifications must follow an *all or nothing* rule. Each transaction is said to be atomic if when one part of the transaction fails, the entire transaction fails. It is critical that the database management system maintains the atomic nature of transactions in spite of DBMS, operating system or hardware failures.

Consistency The consistency property ensures that if the database is in a consistent state before the start of a transaction, it remains consistent when the transaction is over (whether successful or not). The consistency property states that only valid data will be written to the database. If, for some reason, a transaction is executed that violates the database consistency constraints, the entire transaction will be rolled back and the database will be restored to a state consistent with those constraints. On the other hand, if a transaction successfully executes, it will take the database from one state that is consistent with the constraints to another state that is also consistent with the constraints.

Isolation Isolation refers to the requirement that other transactions cannot access or see the data in an intermediate state during a transaction. This constraint is required to maintain the consistency among transactions in a DBMS. Thus, each transaction is unaware of other transactions executing concurrently in the system.

Durability Durability refers to the guarantee that once the user has been notified of success, the committed transaction will persist, and will not be undone. This means it will survive system failures. Many databases implement durability by writing all transactions into a persistent transaction log that can be played back to recreate the system state right before a failure. A transaction can only be considered committed after it is safely stored in the log. Durability does not imply a permanent state of the database. Another transaction may overwrite any changes made by the current transaction without hindering durability.

2.2.1 Consistency Models

Several consistency models for transactional systems may be obtained by relaxing the Isolation property. As in DSM systems, there is an overhead in maintaining the most restrictive level of isolation. For instance, a DBMS that uses a lock-based implementation of concurrency control implementation must acquire locks on data, which may result in performance degradation and loss of concurrency.

Most DBMSs offer a number of transaction isolation levels which differ on the degree of isolation that is enforced when accessing data. For many database applications, the majority of database transactions can be programmed in such a way as to not require high isolation levels, thus reducing the overhead in the system. Conversely, at higher isolation levels, one may be faced with concurrency loss, a problem that also requires careful analysis and careful programming. Transaction isolation levels are a measure of the extent to which transaction isolation succeeds. In particular, transaction isolation levels are defined by the presence or absence of the following phenomena:

Dirty Reads A dirty read occurs when a transaction reads data that has not yet been committed. For example, suppose transaction 1 updates a row. Transaction 2 reads the updated row before transaction 1 commits the update. If transaction 1 rollback the change, transaction 2 will have read data that is considered to have never existed.

Non-repeatable Reads A non-repeatable read occurs when a transaction reads the same row twice but gets different data each time. For example, suppose transaction 1 reads a row.

Transaction 2 updates that row and commits the update. If transaction 1 re-reads the row, it retrieves different row values. The same problem occurs if Transaction 2 deletes the row instead of updating it.

Phantoms A phantom is a row that matches the search criteria but is not initially seen. For example, suppose transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 generates a new row (through either an update or an insert) that matches the search criteria for transaction 1. If transaction 1 re-executes the statement that reads the rows, it gets a different set of rows.

The next paragraphs describe the several existing transaction isolation levels in terms of the previously described phenomena, either if they can occur or not.

Serializable This isolation level specifies that all transactions are completely isolated: all transactions in the system execute in a way that is equivalent to a serial order, i.e. as if they have been executed one after the other and *none* of the previously described phenomena can occur. The DBMS may execute two or more transactions at the same time only if the equivalence to a serial execution can be maintained. With a lock-based concurrency control DBMS implementation, serializability requires that range locks are acquired in certain ranged queries. When using non-lock concurrency control, no lock is acquired, but if the system detects a concurrent transaction in progress which would violate the serializability, it must force that transaction to rollback, and the application will have to restart the transaction.

Snapshot Isolation Many databases (e.g. PostgreSQL) implement a form of isolation, called *snapshot isolation*. Some databases (e.g. MS SQL Server) support both serializable and snapshot isolation modes. A transaction executing under snapshot isolation appears to operate on a personal snapshot of the database, taken at the start of the transaction. When the transaction concludes, it will successfully commit only if the values of the items updated by the transaction have not been updated by other committed transactions since the snapshot was taken. Such a write-write conflict causes the transaction to abort. In this isolation level, *write skew* anomalies can occur, which happens when two transactions concurrently read an overlapping data set, concurrently make *disjoint* updates, and finally concurrently commit, neither having seen the update performed by the other. If the system was serializable, such an anomaly would be impossible, as either one of the transactions would have to occur *first*, and be visible to the other. In contrast, snapshot isolation permits write skew anomalies.

Repeatable Read A transaction using the Repeatable Read isolation level can retrieve and manipulate the same rows of data as many times as needed until it completes. However, no other transaction can insert, update, or delete a row of data that would affect the read operations issued by the first transaction until it is either committed or aborted. Although data read is unchanged, *Phantoms* can occur.

Read Committed In this isolation level, data records retrieved by a query are not prevented from modification by some other transaction. This means that *Phantoms* and *Non-repeatable reads* may occur.

Read Uncommitted The Read Uncommitted isolation level allows a transaction to access uncommitted changes that have been made by other transactions. A transaction using the Read Uncommitted isolation level cannot prevent other transactions from accessing the row of data that it is reading. Therefore, transactions are not isolated from each other, meaning that *Phantoms*, *Non-repeatable reads* and *Dirty reads* may occur.

2.2.2 Concurrency Control

Concurrency control in a DBMS ensures that database transactions are performed concurrently without violating the consistency model selected by the programmer. The following paragraphs consider the case where, by default, the DBMS guarantees that only serializable and recoverable schedules are generated. It also guarantees that no effect of committed transactions is lost, and no effect of aborted transactions remains in the database. There are two main concurrency control mechanisms: *Pessimistic* and *Optimistic*. These mechanisms will be described in the next paragraphs.

Pessimistic Concurrency Control Pessimistic concurrency control prevents consistency violations by acquiring locks on data. The most used form of pessimistic concurrency control is Two-Phase Locking (2PL). According to the 2PL protocol, a transaction handles its locks in two distinct, consecutive phases during the transaction's execution: (*i*) *expanding phase*: locks are acquired and no locks are released and (*ii*) *shrinking phase*: locks are released and no locks are acquired. The serializability property is guaranteed for a schedule with transactions that obey the protocol. The 2PL schedule class is defined as the class of all the schedules comprising transactions with data access orders that could be generated by the 2PL protocol. 2PL is a super-class of Strong Strict Two-Phase Locking (SS2PL), which is widely utilized for concurrency control in database systems. Using locks that block processes, 2PL may be subject to deadlocks that result from the mutual blocking of two or more transactions.

Optimistic Concurrency Control Optimistic concurrency control (OCC) is a concurrency control method that assumes that most transactions complete without affecting each other. Therefore, it allows transactions to proceed without locking the data resources. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction aborts. If conflicts happen often, the cost of repeatedly restarting transactions hurts performance significantly. Pessimistic concurrency control methods have better performance under these conditions.

More specifically, OCC transactions involve the following phases: (*i*) *begin*, where a timestamp is saved, marking the transaction's beginning; (*ii*) *modify*, where read and write operations are issued; (*iii*) *validate* that checks whether other transactions have modified data that this transaction has modified. The validation requires to always check transactions that completed after this transaction's start time. Optionally, check also transactions that are still active at validation time; finally, (*iv*) *Commit/Rollback* that makes all changes part of the official state of the database, if there are no conflicts. If there is a conflict, this is typically resolved by aborting the transaction. OCC is generally used in environments with low data contention. When conflicts are rare, transactions can complete without the expense of managing locks and without having transactions wait for other transactions' locks to clear, leading to higher throughput than other concurrency control methods.

There is also an optimistic concurrency control method that provides each user connected to the database with a *snapshot* of the database, which is called Multi-Version Concurrency Control (MVCC). MVCC uses timestamps or increasing transaction IDs to ensure the required consistency model. MVCC ensures that transactions never have to wait for a database object, by maintaining several versions of an object. Each data version has a write timestamp. A transaction is allowed to read the most recent version of an object which precedes the transaction timestamp. The obvious drawback to this method is the cost of storing multiple versions of objects in the database. On the other hand reads are never blocked, which can be important for workloads mostly involving reading values from the database. MVCC is particularly apt for implementing snapshot isolation.

2.2.3 Distribution

Databases can be distributed among several nodes in a network. In this case, data is spread through several DBMSs and distributed commit protocols are needed to maintain the atomicity of distributed transactions. A distributed transaction can be seen as a database transaction that must provide the ACID properties among multiple participating databases which are distributed among different physical locations. The Atomicity and Isolation properties pose a special challenge for distributed database transactions. Atomicity must be ensured in all participating nodes, where all the nodes must reach to the same transaction outcome (commit or rollback). Isolation is also a challenge, since the global isolation level (e.g. Serializability) may be violated, even if each database provides it.

There are mainly two protocols that are used in distributed transactions to ensure Atomicity, each briefly described in the next paragraphs.

2 Phase Commit Two Phase Commit (2PC) is a distributed algorithm that coordinates all the processes that participate in a distributed atomic transaction to decide on whether to commit or rollback the transaction. The two phases of the algorithm are the *commit-request* phase, in which a coordinator process attempts to prepare all the processes involved in the transaction to take the necessary steps for either committing or aborting the transaction; a process responds to the prepare command with a COMMIT vote if it is ready to commit the transaction, or with an ABORT abort vote otherwise. Subsequently, is executed a *commit* phase, in which, based on the votes (COMMIT or ABORT) of the participants, the coordinator decides whether to commit (only if all vote COMMIT) or abort the transaction (otherwise), and notifies the result to the participants. The participants then execute the required actions to commit or abort the local outcome of the transaction.

The protocol is efficient and thus often adopted. However, it is not resilient to all possible failure scenarios and may block if the coordinator crashes after sending the *commit-request* and before disseminating the decision. In these rare cases user intervention is needed to remedy the outcome. The 3 Phase Commit protocol solves most of these issues.

3 Phase Commit Like 2PC, the three-phase commit protocol (3PC) is a distributed algorithm which allows nodes in a distributed system to agree whether to commit or abort a transaction. Unlike 2PC however, 3PC is non-blocking. Specifically, 3PC places an upper bound on the amount of time required before a transaction either commits or aborts. This property ensures that if a given transaction is attempting to commit via 3PC and holds some resource locks, it will release the locks after the timeout.

The 3PC algorithm works as follows. The coordinator receives a transaction request. If there is a failure at this point, the coordinator aborts the transaction (i.e. upon recovery, it will consider the transaction aborted). Otherwise, the coordinator sends a canCommit? message to the participants and moves to the "waiting" state. If there is a failure, timeout, or if the coordinator receives a No message in the "waiting" state, the coordinator aborts the transaction and sends an abort message to all participants. Otherwise the coordinator will wait until it receives Yes messages from all participants within the time window, and then it sends preCommit messages to all participants and moves to the "prepared" state. If the coordinator fails in the "prepared" state and later recovers, it will move to the "commit" state. However if the coordinator times out while waiting for an acknowledgement from a participant, it will abort the transaction. In the case where all acknowledgements are received, the coordinator moves to the "commit" state as well. The participants receive a canCommit? message from the coordinator. If the participant agrees it sends a Yes message to the coordinator and moves to the "prepared" state. Otherwise it sends a No message and aborts. If there is a failure, it moves to the "abort" state. In the "prepared" state, if the participant receives an abort message from the coordinator, fails, or times out waiting for a commit, it aborts. If the participant receives a preCommit message, it sends an acknowledgement message back and commits. The main disadvantage of this algorithm is that it cannot recover in the event the network is partitioned. The original 3PC algorithm assumes a fail-stop model, where processes fail by crashing and crashes can be accurately detected, and does not work with network partitions or asynchronous communication.

2.2.4 Replication

Many classical approaches to database replication are based on a primary/backup model where one replica has unilateral control over one or more other replicas. For example, the primary might execute a transaction and send the log of updates to a backup replica, which can then take over if the primary fails. This approach is the most common one for replicating databases, despite the risk that if a portion of the log is lost during a failure, the backup might not be in a state identical to the one the primary was in, and transactions could then be lost. Although these type of weaknesses can be resolved with acknowledgements from the backup replicas, this naive solution introduces latency and is not based in a well defined model, motivating the need to explore alternative methods of replicating data.

Modern database replication schemes (Pedone, Guerraoui, & Schiper 2003; Patiño Martínez, Jiménez-Peris, Bettina, & Alonso 2000; Cecchet, Marguerite, & Zwaenepole 2004; Bettina & Alonso 1998) rely on an Atomic Broadcast (ABcast) primitive (Guerraoui & Rodrigues 2006), typically provided by some Group Communication System (GCS) (Miranda, Pinto, & Rodrigues 2001; Amir, Danilov, & Stanton 2000). ABcast plays a key role to enforce, in a non-blocking manner, a global transaction serialization order without incurring in the scalability problems affecting classical eager replication mechanisms based on distributed locking and atomic commit protocols, which require much finer grained coordination and fall prey of deadlocks (Gray, Helland, O'Neil, & Shasha 1996). Existing ABcast-based database replication literature can be coarsely classified in two main categories, depending on whether transactions are executed optimistically (Pedone, Guerraoui, & Schiper 2003; Bettina & Alonso 1998) or conservatively (Kemme, Pedone, Alonso, & Schiper 1999).

In the conservative case, which can be seen as an instance of the classical state machine/active replication approach (Schneider 1993), transactions are serialized through ABcast prior to their actual execution and are then deterministically scheduled on each replica in compliance with the serialization order determined by the ABcast. This prevents aborts due to concurrent execution of conflicting transactions in different replicas and avoids the cost of broadcasting the transactions' read and written data. On the other hand, the need for enforcing deterministic thread scheduling at each replica requires a careful identification of the conflict classes to be accessed by each transaction, prior to its actual execution. In practice, it is very hard to predict the data-sets that are to be accessed by a newly generated transaction. This is particular troublesome, given that a labeling error can lead to inconsistency, whereas coarse overestimations can severely limit concurrency and hamper performance.

Optimistic approaches avoid these problems (Pedone, Guerraoui, & Schiper 2003). In these approaches, transactions are locally processed on a single replica and validated after its execution through an ABcast based certification procedure aimed at detecting remote conflicts between concurrent transactions. The certification based approaches can be further classified into voting and non-voting schemes (Bettina & Alonso 1998; Rodrigues, Miranda, Almeida, Martins, & Vicente 2002), where voting schemes, unlike non-voting ones, need to atomic broadcast only the written objects (which is typically much smaller than the set of read objects in common workloads), but on the other hand incur the overhead of an additional uniform broadcast (Guerraoui & Rodrigues 2006) along the critical path of the commit phase.

2.2.5 Examples of Replicated Databases

There are many replication solutions available nowadays. This section describes a couple of examples, namely: Slony-I (Slony Development Group 2011), which is not based on group communication and Postgres-R (Kemme & Alonso 2000), which is based on group communication.

Slony-I The Slony-I (Slony Development Group 2011) is one of the replication products available for the PostgreSQL DBMS. It implements lazy replication with a "master to multiple slaves" replication and is divided in three phases: capture, distribution, and apply. The capture phase involves obtaining updates performed to replicated objects in a format suitable for publication. It is implemented using triggers that log the changes made against the published objects. The distribution phase propagates changes in published objects to relevant replicas. The changes are periodically distributed using a replication daemon that connects directly to the publisher, reads the logged changes and forwards it to the subscribers. It allows to connect several subscribers in cascade. Applying updates is done by restricting modification of each data item to a designated master copy. Otherwise, an application specific procedure for reconciliation must be initiated, which is supported by sorting conflicting updates and triggering the necessary events.

Postgres-R In practice classical approaches to database replication often suffer from high deadlock rates, message overhead and poor response times. Several works have proposed implementing eager replication on top of group communication middleware. Among them Kemme and Alonso introduced Postgres-R (Kemme & Alonso 2000), a toolkit that implements update everywhere replication protocols, which use group communication primitives to ensure ordering and atomicity of transactions. This is a real implementation of the previously described optimistic approach protocols based on group communication. Postgres-R is composed by an extended version of the PostgreSQL DBMS with hooks to extract relevant information about the transactions to replicate and a replication manager, which interacts with a group communication service. This communication service ensures that all replicas receive all (local or remote) transactions by the same order.

2.3 Software Transactional Memory

Memory synchronization mechanisms are useful to develop thread-safe single objects, but are of little help in more complex operations. Ensuring, with lock-based mechanisms, that all the objects accessed during a complex operation remain in a consistent state is difficult and highly error-prone. Recent work on Transactional Memory (Shavit & Touitou 1995; Herlihy, Luchangco, Moir, & Scherer 2003; Harris, Marlow, Peyton-Jones, & Herlihy 2005) propose to integrate transactions in programming languages to mediate the access to shared memory, thereby ensuring the consistency of the data. There are hardware and software transactional memory implementations of transactional memory. This work focus on Software Transactional Memory (STM).

This section overviews the characteristics of the STM systems. More specifically, it will be described the consistency properties and the state of the art on centralized STMs.

2.3.1 Consistency Properties

This section describes the existing consistency models in the scope of STM systems. As in Distributed Shared Memory and Database Management Systems, concepts such as *Linearizability* and *Serializability* can also be applied to the memory transactions. In the STM context, these properties are applied to transactions that are composed of read and write operations on memory objects. The next paragraphs informally describe specific properties applied to memory transactions.

Atomicity Properties

Transactions should be atomic with respect to each other, but their relationship to non-transactional code is not so clear. This ambiguity is not merely an implementation detail, since legal programs may contain unprotected references to shared variables (i.e., outside transactions) without creating malignant data races, so both transactional and non-transactional code can refer to the same data. To take these two cases into account, two models for reasoning about scope of atomicity were defined (Martin, Blundell, & Lewis 2006): *Strong Atomicity* and *Weak Atomicity*. Strong atomicity is a semantics in which transactions execute atomically with respect to both other transactions and non-transactional code. Strong atomicity has two components: it requires both non-interference and containment from non-transactional code. In essence, strong atomicity implicitly treats each instruction appearing outside a transaction as its own singleton transaction. Weak atomicity is a semantics in which transactions are atomic only with respect to other transactions and their execution may be interleaved with non-transactional code, therefore violating either non-interference or containment (or both). This is one of the main differences between DBMS and STM transactions: all the operations in a DBMS run in the scope of a transaction and it do not have to deal with non-transactional code.

Liveness Properties

Non-blocking synchronization ensures that transactions competing for the access of a shared object do not have their execution indefinitely postponed by mutual exclusion. There are mainly three liveness properties already introduced in shared memory objects: Obstruction-freedom (Herlihy, Luchangco, & Moir 2003), Lock-freedom (Herlihy & Moss 1993) and Wait-freedom (Herlihy 1991). When applied to STM systems, these properties are known respectively as *Solo-progress*, *Global-progress* and *Total-progress* (Guerraoui & Kapalka 2009).

In a system that provides *Solo-progress*, if all other threads are suspended, a single transaction in a single thread will eventually complete. What this entails is that a system that ensures solo-progress must have the ability to abort a transaction that is holding some object needed by the transactions that is running on the non-suspended thread and be guaranteed to do so after some amount of time. Live-locks are not denied with solo-progress; two transactions can repeatedly be aborted because of the other one, forever.

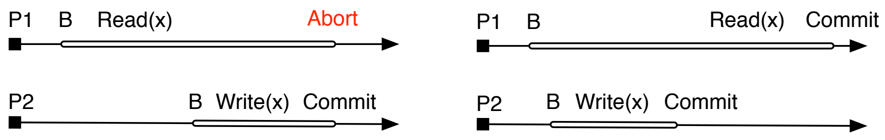


Figure 2.5: Progressiveness example.

Global-progress ensures system wide progress. An algorithm that ensures global-progress, ensures solo-progress and it also ensures that every step taken moves a step forward by committing transactions. In practice, in the context of STMs, this means that one transaction can be aborted due to a conflict with another transaction, but only if some transaction is guaranteed to commit. However, it does not prevent a single transaction from suffering of starvation.

Total-progress is the strongest liveness guarantee, combining guaranteed system-wide throughput (global-progress) with starvation-freedom. An algorithm ensures total-progress if every operation has a bound on the number of steps the algorithm will take before the operation completes, meaning that every correct transaction eventually commits.

Safety Properties

There are two main properties that state the circumstances where transactions must commit: *Opacity* and *Progressiveness* (Guerraoui & Kapalka 2008a). The Opacity property captures the requirements that (*i*) all operations performed by every committed transaction appear as if they happened at some single, indivisible point during the transaction lifetime, (*ii*) no operation performed by any aborted transaction is ever visible to other transactions, and (*iii*) every transaction always observes a consistent state of the system.

The Progressiveness property is defined as a safety property from the commit/abort termination point of view and defines an execution pattern that prevents a transaction from aborting another arbitrary transaction. Two transactions conflict if they access the same object and one of them updates it. A STM system satisfies the progressiveness property if it aborts a transaction T_1 only when there is a time t at which T_1 conflicts with another concurrent transaction T_2 that is not committed or aborted by time t. In all other cases, the system cannot abort T_1. The Figure 2.5 shows an execution example that indicates when a transaction should be aborted or not. The scenario on the left is a valid conflict, but the scenario on the right cannot be considered a conflict if the system ensures the progressiveness property. The transactions on

$P1$ and $P2$ are both active, $P2$ writes to variable X and terminates. When $P1$ reads X (after $P2$ terminated), it cannot be considered a conflict, otherwise the Progressiveness property is not ensured.

2.3.2 Examples of STM Systems

This section describes some representative state of the art STM systems, namely JVSTM (Cachopo & Rito-Silva 2006), TL2 (Dice, Shalev, & Shavit 2006), LSA (Riegel, Felber, & Fetzer 2006), Lock-based STM (Imbs & Raynal 2008) and Strong-STM (Abadi, Harris, & Mehrara 2009). Finally, it is also important to briefly describe several existing experimental frameworks for building and testing different software transactional memory implementations, namely DSTM2 (Herlihy, Luchangco, & Moir 2006a), Deuce (Korland, Shavit, & Felber 2009) and TMunit (Harmanci, Gramoli, Felber, & Fetzer 2010).

Versioned STM This approach is based on boxes, that may hold multiple versions of their contents (Cachopo & Rito-Silva 2006), implementing multi-version concurrency control. The Versioned Software Transactional Memory (VSTM) approach allows the execution of read-only transactions that never conflict with other concurrent transactions and, at the same time, provides Linearizability and Opacity at the cost of having as much versions of the same object as the ones needed by the active transactions, but it does not provide the Progressiveness property. The VSTM was implemented as a Java library: the JVSTM (Java Versioned Software Transactional Memory). This library is part of the Fenix-framework (IST 2009a) and is used in the FénixEDU web application.

Transactional Locking II The TL2 (Dice, Shalev, & Shavit 2006) STM implements an algorithm based on a combination of commit-time locking and a global version-clock based validation technique. In TL2, a memory variable has an associated lock and a version number. The version number of each lock is the number of the last committed transaction. Each transaction has a read-set and a write-set. The transactions execute optimistically and the transaction keeps track of all the values read and written. At commit time, the locks for the values written are acquired and the read-set is validated using the version number. If the transaction succeeds both the operations, the values of the write-set are written in the memory, the locks are released and the global transaction number is incremented. With this approach, TL2 provides Opacity, but also does not provide the Progressiveness property.

Lazy Snapshot Algorithm LSA (Riegel, Felber, & Fetzer 2006) was implemented using an object-based STM (LSA-STM) and it ensures Linearizability and Opacity. It is multi-versioned, constructing a consistent snapshot for transactions where reads of a transaction are invisible to other transactions. The consistency of a transaction is verified by maintaining a validity interval for snapshots. A STM that implements this algorithm can verify during each object access that the snapshot of the objects that a transaction has seen so far is consistent. It also ensures the Progressiveness property.

Lock-based STM The STM presented by Imbs & Raynal (2008) is a lock based STM that satisfies the opacity and progressiveness properties and the following features. It uses visible reads, does not require the shared memory to manage several versions of each object and does not use timestamps or version numbers. When compared with other existing STMs, this is the one that provides more safety properties, but this can be achieved at the cost of performance.

Strong-STM The work by Abadi, Harris, & Mehrara (2009) presents a STM that will be called Strong-STM in this document and is implemented in C#. This STM is based on the Bartok compiler and runtime system (Harris, Plesko, Shinnar, & Tarditi 2006) and ensures strong atomicity by means of off-the shelf memory protection hardware. Each memory location in the physical address space is mapped by a virtual address space. The virtual address space is composed by a normal and transactional heap. This is used to distinguish between normal and transactional accesses to memory. When a transaction writes to some object, reads from other code (transactional or non-transactional) are disabled until the transaction commits.

Dynamic Software Transactional Memory DSTM2 (Herlihy, Luchangco, & Moir 2006b) represents, to the best of our knowledge, the first generic framework proposed to simplify and homogenize the development and comparison of alternative *non-distributed* STM schemes, while keeping application code that uses them the same to allow head-to-head comparisons. Specifically, the focus of the DSTM2 framework is on the comparison of different contention management algorithms (namely on the policy to be adopted by an STM upon detection of a conflict between transactions) in a single versioned STM. It contains already some STM implementations built using the framework, as well as some example benchmarks.

Deuce STM A more recent generic STM framework is Deuce (Korland, Shavit, & Felber 2009). Using Deuce, no modifications to the java virtual machine or extensions to the language are necessary. By default, it uses a locking design that detects conflicts at the field level without a significant overhead in the memory footprint. It allows to accommodate implementations of new STM algorithms, including multi-versioned STMs. Unlike DSTM2, Deuce makes extensive usage of *byte-code* injection and dynamic *byte-code* rewriting in order to maximize transparency towards applications.

TMunit TMunit (Harmanci, Gramoli, Felber, & Fetzer 2010) is an extensible transactional memory testbed that provides a domain specific language to build transactional memory workloads, so different STM implementations can be easily evaluated and compared, as well as to validate the behavior of a specific implementation. The benchmarks specified in this testbed can be interpreted and map the transactional accesses to an underlying STM or can be compiled to achieve better performance.

2.4 Discussion

DSM is a programming model that eases the development of distributed applications by hiding the network information exchange among processes on different nodes of the system. Typically, a DSM system attempts to keep consistency at a very fine grain level, which often resulted in disappointing performance. To improve the performance of the DSM systems on write operations, more relaxed memory consistency models have been proposed. With these models, the system performance was improved, but these new models are harder to understand by programmers and can result in error prone coding. Fault tolerance in DSM systems is usually achieved using checkpointing techniques. Another way to achieve fault tolerance is using replication, which was widely studied in the context of DBMSs.

A DBMS is a technology able to ease the management, storage and retrieval of large amounts of information. It uses a standardized language for applications that eases its portability. It uses the abstraction of a *transaction* with well defined properties to maintain the data consistent. Many protocols for distribution and replication of database transactions have been studied in the last years. Since DBMSs share with STM systems the notion of transactions, the replication protocols used in DBMSs can serve as inspiration to build fault tolerant distributed STM solutions.

STM systems also support transactions but do not restrict the programming model to a query language, and support arbitrary constructs. In this more general context, transactions are still a very good alternative to the explicit lock management. Several STM systems have been designed over the last years. These systems differ on the properties that they provide and on what mechanisms are used to provide those properties. Some systems are lock based, others rely on timestamps to ensure data consistency. Multi-versioning is also used in some STM systems, that are able to ensure safety properties more efficiently, but with the memory cost for keeping several versions of the same object.

Most of the STM systems were designed to be used only on a single machine, but some more recent systems are already taking into account distribution and replication mechanisms. There is a tradeoff between scalability and fault tolerance, since fault tolerance implies more coordination of replicas. The extra coordination can imply a reduction of performance and scalability. The next chapter is focused on distributed dependable STMs and discusses the tradeoffs of the existing systems.

2.5 Summary

This chapter provided an overview of the main systems that served as inspiration for the work presented in the thesis. It has discussed the consistency models of the Distributed Shared Memory Systems and Database Management Systems and related them with distribution and fault tolerance. This chapter has also presented the relevant properties and features of existing Software Transactional Memory systems.

3 Distributed and Replicated STMs

Among the Distributed Software Transactional Memories (DSTM) that have been proposed so far, one can find replicated and non-replicated systems. Non-replicated systems are more focused on scalability with regard to the number of nodes and each data item is usually maintained in a single node. In sharp contrast, in fully replicated systems, each data item is replicated in every node, making the system fault tolerant. When the transaction executes in one node, there is no need for fetching data from remote locations but, on the other and, all the nodes must synchronize upon the commit of a transaction in order to maintain the memory consistent (according to some given consistency criteria).

This chapter discusses the specific issues related with the development of distributed and replicated STMs. The chapter starts by characterizing the system model and the middleware that can be used to build such systems, then it surveys STM systems that have addressed distributed, and later it identifies the main challenges in adding replication to STMs. Finally, the chapter draws a roadmap of interesting avenues for research, some followed by the work reported in this dissertation and other followed by other members of the Grupo de Sistemas Distribuídos at INESC-ID.

3.1 System Model

This work considers an asynchronous distributed system model consisting of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate via message passing and can fail according to the fail-stop (crash) model (Guerraoui & Rodrigues 2006). This work assumes that a majority of processes is correct and that the system ensures a sufficient synchrony level to permit implementing a View Synchronous Group Communication Service (GCS) (Chockler, Keidar, & Vitenberg 2001). A GCS integrates two complementary services: *membership* and *multicast communication*. Informally, the role of the membership service is to provide, each participant in a distributed computation with information about which process is active (or reachable) and which one is failed (or unreachable). Such information is called a *view* of the group of participants. The multicast service allows a member to send a message to the group of participants with different reliability and ordering properties.

The work described in this thesis assumes that the GCS provides a primary-component group membership service (Bartoli & Babaoglu 1997), which maintains a single agreed view of the group at any given time and provides processes with information on whether they belong to the primary component. Specifically, the GCS delivers to the application a *viewChange* event to notify the alteration of the (primary component) view, and an *ejected* event to notify the exclusion of the process from the primary component (typically because of a false failure suspicion). It is said say that a process is v_i-correct in a given view v_i if it does not fail in v_i and if v_{i+1} exists, it transits to it. It is assumed that a GCS ensuring the following properties on the delivered views:

Self-inclusion: if process p delivers view v_i, then p belongs to v_i.

Strong virtual synchrony: messages are delivered in the same view in which they were sent.

Primary component view: the sequences of views delivered are totally ordered and for any two consecutive views v_i, v_{i+1} there always exists a v_i-correct process p such that $p \in v_i$ and $p \in v_{i+1}$.

Non-Triviality: when a process fails or becomes partitioned from the primary view, it will be eventually excluded from the primary component view.

Accuracy: a correct process that is not is partitioned from the primary view, is eventually included in every view delivered by the GCS.

The GCS offers two communication services, namely: Optimistic Atomic Broadcast (OAB) (Guerraoui & Rodrigues 2006) and Uniform Reliable Broadcast (URB) (Guerraoui & Rodrigues 2006). URB is defined by the primitives *UR-broadcast*(m) and *UR-deliver*(m). Three primitives define OAB: *OA-broadcast*(m), which is used to broadcast message m; *OA-Opt-deliver*(m), which delivers message m without providing ordering guarantees; *OA-Final-deliver*(m), which delivers message m in the final total order. *OA-Opt-deliver*(m) provides an early estimate of the final order of the corresponding *OA-Final-deliver*(m); this estimate may be inaccurate.

The properties of the OAB are as follows:

Validity: If a v_i-correct process p *OA-broadcasts* message m in v_i, then p *OA-Opt-delivers* and *OA-Final-delivers* m.

Integrity: Any message m is *OA-Opt-delivered* and/or *OA-Final-delivered* by a process p at most once, and only if it had been previously *OA-broadcast*.

Optimistic Order: If a node p *OA-Final-delivers* m, then node p has previously *OA-Opt-delivered* m.

Uniform Agreement: If process p *OA-Final-delivers* m in view v_i, then any v_i-correct process *OA-Final-delivers* m in view v_i.

Total Order: If two processes p and q *OA-Final-deliver* messages m and m', then they do so in the same order.

The properties of the URB are as follows:

Validity: If a v_i-correct process p *UR-broadcasts* message m in v_i, then p *UR-delivers* m.

Integrity: Any message m is *UR-delivered* by a process p at most once, and only if it had been previously *UR-broadcast*.

Uniform Agreement: If process p *UR-delivers* m in view v_i, then any v_i-correct process $q \in v_i$ *UR-delivers* m in view v_i.

Causal Order: If a process p *UR-delivers* m and m' such that m causally precedes m', according to Lamport's causal order (Lamport 1978) (denoted $m \rightarrow m'$), then p *UR-delivers* m before m'.

Note that in addition to the URB properties, OAB ensures total order of the *OA-Final-deliver* events, preceded by a guess of the final order through the *OA-Opt-deliver* event. Providing the total order property is more expensive than causal order in terms of exchanged messages and communication latency.

3.2 Distributed STMs

This section surveys distributed STM systems that offer no support for replication. The main goal of these systems is to increase the scalability of the STM, by allowing different data items to be maintained at different nodes. Therefore, each node is responsible for a different sub-set of the entire address space. When a transaction executes, communication is needed with the nodes that store the data: either by shipping code to those nodes, or by reading the values of those data items. Also, when a transaction wants to commit, coordination among the nodes that store items accessed by that transaction needs to be performed. As it will be seen, most systems use some form of 2 Phase Commit (2PC) for coordination and locks to implement concurrency control.

Cluster-STM Cluster-STM (Bocchino, Adve, & Chamberlain 2008) focuses on the problem of how to partition the dataset across the nodes of a large scale distributed Software Transactional Memory. This is achieved by assigning to each data item a home node, which is responsible for maintaining the authoritative version (and the associated metadata) of the data item. The home node is also in charge of synchronizing the accesses of conflicting remote transactions. Cluster-STM delegates caching or replication to the application level, which is then required to take explicitly into account the issues related to data fetching and distribution, with an obvious increase in the complexity of the application development. Further, Cluster-STM treats the processes as a flat set, not distinguishing processes that execute in the same node from processes that execute on different nodes and, therefore, it does not exploit the availability of shared memory connected to multiple cores in a single node to speed up intra-node communication. Finally, Cluster-STM does not use multi-versioning local concurrency control to improve the performance of read-only transactions, and is constrained to run only a single thread for each processor.

DiSTM DiSTM (Kotselidis, Ansari, Jarvis, Luján, Kirkham, & Watson 2008) uses a distributed mutual exclusion mechanism scheme to coordinate the commit of transactions. Mutual exclusion is aimed at ensuring that at any time there are no two nodes attempting to simultaneously commit conflicting transactions. To shield nodes form the cost of participating in a distributed mutual exclusion protocol for each transaction commit, nodes are granted leases on datasets, based on the their data access pattern. This partially alleviates the performance problems incurred by the need to serialize the whole (distributed) commit phase. However, this phase may still become a bottleneck in face of conflict intensive workloads. Additionally, in DiSTM the lease establishment mechanism is coordinated by a single, centralized, node which is likely to become a performance bottleneck for the whole system as the number of nodes increases; In fact, the experimental evaluation reported by Kotselidis, Ansari, Jarvis, Lujan, Kirkham & Watson (2008) relies on a dedicated node for lease management and does not report results for more than four nodes.

DMV In DMV (Manassiev, Mihailescu, & Amza 2006) each node keeps a (single) local copy of the data items it reads or writes. This effectively creates the existence of multiple copies of each data item, opening the door for the implementation of a distributed multi-versioning concurrency control scheme (DMV). Like centralized multi-version concurrency control schemes (Bernstein, Hadzilacos, & Goodman 1987), DMV allows read-only transactions to be executed in parallel with conflicting updating transactions. This is achieved by ensuring that the former is able to access older, committed snapshots of the dataset. However, in DMV each node maintains only a single version of each data granule, and explicitly delays propagating (local or remote) updates to increase the chance of not having to invalidate the snapshot of currently active read-only transactions (and to consequently abort them). This allows DMV to avoid maintaining multiple versions of the same data at each node, unlike in conventional multi-version concurrency control solutions (although DMV requires buffering the updates of not yet applied transactions). On the other hand, while multi-version concurrency control solutions provide deterministic guarantees on the absence of aborts for read-only transactions, the effectiveness of the DMV scheme depends on the timing of the concurrent accesses to data by conflicting transactions (actually, with DMV a read-only transaction may be aborted also due to the concurrent execution of a "younger", local read-only transaction). Another characteristic of DMV is that it requires each committing transaction to acquire a cluster-wide unique token, which globally serializes the commit phases of transactions. Unfortunately, given that committing a transaction imposes a two communication step synchronization phase (for updates propagation), the token acquisition phase can introduce considerable overhead and seriously hamper performance (Kotselidis, Ansari, Jarvis, Luján, Kirkham, & Watson 2008).

Fenix-framework The VSTM (see Section 2.3.2) was augmented with support for persistence using a DBMS, which allows also to run a distributed version by executing several instances of the application server that synchronize using the persistence storage. This software package is called Fenix-framework and is part of FénixEDU (IST 2009b). The current version that is being used in the production environment uses a load balancer to distribute the client requests among several application servers. The application servers use a logically centralized DBMS to persist the data. The DBMS is also used as a synchronization mechanism to maintain the cache of the servers consistent. The Fenix-framework distributes several instances of VSTM and is being used in production today. Unfortunately, the current solution still relies on the access to a logically centralized database to enforce the global synchronization required to ensure the VSTM correctness. Thus, albeit more scalable than a centralized STM, the current solution still has many limitations to scalability.

3.3 Towards STM Replication

From the area of replication data management, the state of the art protocols that are most relevant to the goal of building a replicated STM are certainly those that have been developed to support database replication. First, as it has been shown in the previous chapter, there are many similarities among replicated databases and replicated STMs. Secondly, protocols for database replication have been a prolific research area in the last decade, and the most successful protocols already leverage on the advances that have been made in the area of distributed computing, including a better understanding of consensus and related problems, such as total order multicast. Therefore, database replicated protocols, namely state-machine database replication and certification-based database replication protocols are natural candidates to build a Distributed Replicated STM.

However, it is important to understand the main differences among replicated databases and replicated STMs, in order to identify the key challenges in applying protocols developed for the database setting in the STM context. These differences can be listed as follows:

- STMs often favor opacity as a consistency model, which is more restrictive than serializability.

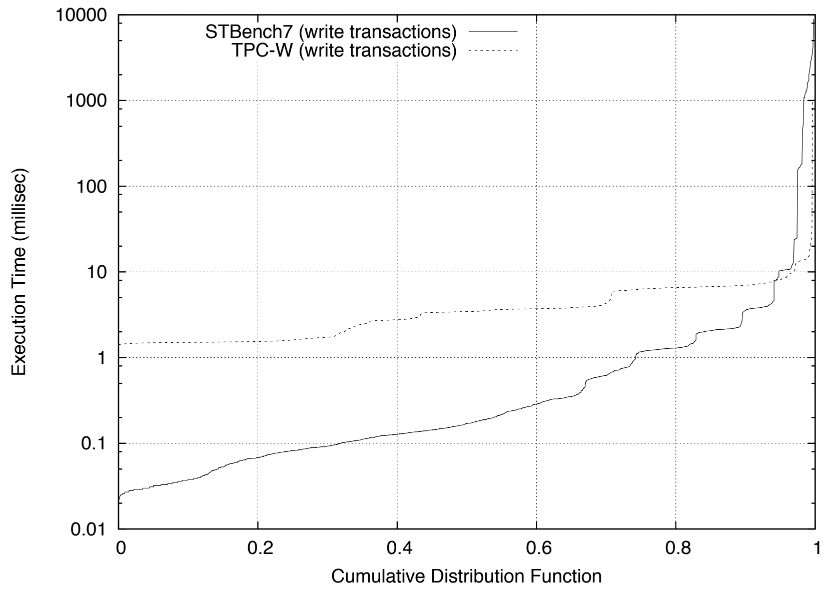


Figure 3.1: Transaction execution time: STMBench7 vs TPC-W.

- STMs do not necessarily support durability. This makes software transactions potentially shorter than database transactions.
- Software transactions are written in a fully-fledged programming language (not restricted to SQL), and can have a characterization different from a typical database transaction.

The first difference it not a major impairment to the development of replicated STMs, given that most replication protocols do introduce additional threats to opacity. For instance, in certification based protocol, remote transactions are less complex to manage than local transactions, as they apply all updates at once.

To quantify the other two differences, Romano, Carvalho, & Rodrigues (2008) have performed a study, where the workload imposed by a popular benchmark for web-based transactional applications, namely TPC-W (Transaction Processing Performance Council 2002) is compared with the workload generated by a typical STM benchmark, namely STMBench7 (Guerraoui, Kapalka, & Vitek 2007).

Figures 3.1 and 3.2 compare the execution latency and read-set/write-set size of sequentially submitted transactions for STMBench7 and TPC-W when executing the two benchmarks on top of a 4 Xeon CPUs machine using Linux 2.6.8-24.18 and equipped with 2 SCSI disks in

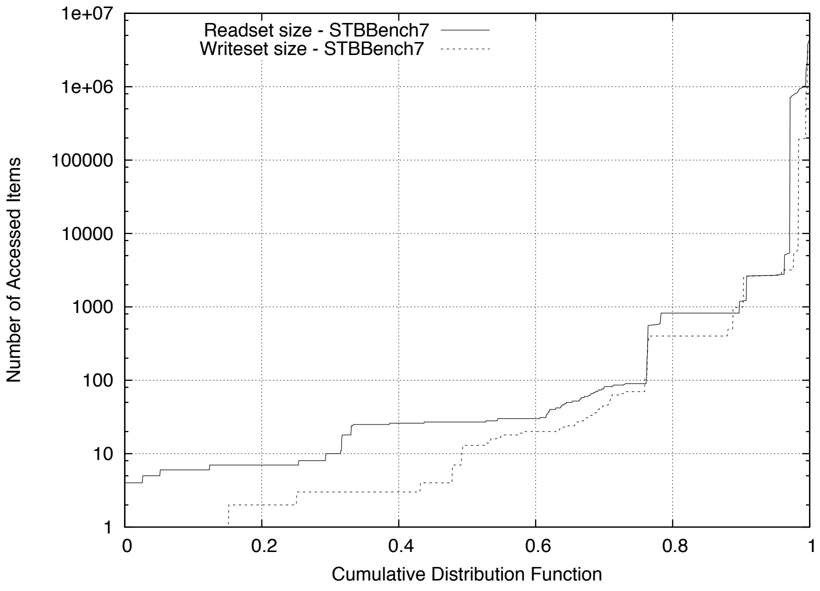


Figure 3.2: Readset/writeset size of STMBench7's transactions.

RAID-0 configuration and 4GB of main memory. The tests run with JVSTM (Cachopo & Rito-Silva 2006) as the STM for STMBench7, and PosgreSQL 8.1 (PostgreSQL Global Development Group 2011) as the relational database underlying TPC-W. This choice was motivated by the fact that both JVSTM and PostgreSQL have a similar approach to concurrency control, both of them relying on a multi-versioning scheme which allows read-only transactions to be executed without ever being blocked or aborted. Also, in order to fairly compare the performance of the two systems, the benchmarks have been configured to generate a very similar percentage of write transactions: specifically the tests were conducted with TPC-W *Ordering Mix* and the STMBench7 *Read-Write Workload Type*, whose percentage of write transactions is around the 50%.

From these results, it is possible to make the following observations. First, the transaction execution time for the STMBench7 is at least one order of magnitude smaller than for TPC-W for around the 60% of transactions. This is essentially due to the fact that, unlike conventional DBMS transactions, STM transactions only access in memory data items, hence not incurring the latencies proper of disk accesses. Also, in STMs the overhead associated with SQL parsing and plan optimization are absent, further contributing to shortening the transaction lifetime. The direct consequence of such a striking reduction of the transaction lifetime in STMs with respect to DBMSs is that, in a replicated STM system, the relative overhead of the atomic broadcast based synchronization schemes would be correspondingly amplified with respect to the scenario of conventional database replication. Another important feature characterizing the STM benchmark is the high heterogeneity of its workload. In fact, the transaction lifetime in STMBench7, as highlighted by Figure 3.1, spans over a wide range, with around the 30% of transactions executing for less than 100 microseconds, and about the 5% of transactions taking from hundreds of milliseconds up to several seconds. This is also reflected by the high heterogeneity of the sizes of the transaction read-sets and write-sets, see Figure 3.2, which range from a just a few items up to around 10 millions. Indeed, the presence of highly diversified components in the workload of STM applications, such as in STMBench7, severely challenges the state of the art on database replication solutions, which are designed to provide optimal performance under much more restricted workloads. Another interesting consideration that can be drawn by observing the write-sets' Cumulative Distribution Function (CDF) in the Figure 3.2, is that the write-sets' and read-sets' cardinality of STM transactions is quite similar, especially when considering long-running transactions (representing nearly the 5% of the STMBench7 workload) which reads and writes from tens of thousands to millions of data items. This strongly contrasts with classical database workloads, in which transactions are rather characterized by small write-sets and often very large read-sets, and for which existing database replication solutions have been optimized.

3.4 Challenges in STM Replication

Based on the observations reported above, three key challenges in developing a distributed replicated STM can be identified:

- *Challenge 1:* Very long read sets may saturate the network and need to be accounted for.
- *Challenge 2:* The distributed coordination costs may dominate the transaction execution time, so coordination should be minimized even further (it is worth noting that database replication protocols already attempt to achieve this goal to some extent).
- *Challenge 3:* Even if distributed coordination is minimized, there will be cases where it will be necessary. In those cases, it is likely that all cores will be idle waiting for the network.

Improvements	Active Replication	Certification
Baseline	(Kemme et al)	Voting and Non-Voting
Challenge 1	–	D^2STM
Challenge 2	–	**ALC**
Challenge 3	AGGRO	**SCert**

Table 3.1: Improving AB based replication protocols.

3.4.1 Research Roadmap

The research performed at the Grupo de Sistemas Distribuídos of INESC-ID, to which this thesis has contributed, allowed to identify three techniques, that were worth exploring as potential solutions for the challenges identified above.

These techniques are the following:

- To address *challenge 1*, we considered using techniques to compress the information that needs to be propagated on the network during coordination, namely the transaction read-set. This can be achieved using mechanisms such as Bloom filters.
- To address *challenge 2*, we considered the use of leases in the context of replication protocols. The details of this approach will be explained in detail later in this document.
- To address *challenge 3*, we considered the use of speculative transaction execution. The idea is to use processor cycles, that would otherwise be idle, to speculatively execute transactions while distributed coordination takes place.

Note that the first two techniques can only be applied to certification based protocols, while speculation can be applied to both state-machine and certification protocols. This creates the potential for four new replication protocols, as illustrated in Table 3.1.

All these research avenues have been explored by the Grupo de Sistemas Distribuídos of INESC-ID. ALC and SCert are reported in this thesis. D^2STM (Couceiro, Romano, Carvalho, & Rodrigues 2009) has been reported in the MSc Thesis of Maria Couceiro. AGGRO (Palmieri, Quaglia, & Romano 2010) has been the result of a collaborative effort between Paolo Romano and researchers from the University of Rome. For self-containment, D^2STM and AGGRO are briefly described in the next paragraphs. ALC and SCert are described in detail in the next chapters of this document.

D^{2}STM

D^{2}STM (Couceiro, Romano, Carvalho, & Rodrigues 2009) works as follows. Read-only transactions are executed locally, and committed without incurring in any additional overhead. Leveraging on the JVSTM multi-version scheme, D^{2}STM read-only transactions are always provided with a consistent committed snapshot and are spared from the risk of aborts. A committing transaction with a non-null write-set, is first locally validated to detect any local conflicts. This prevents the execution of the distributed certification scheme for transactions that are known to abort using only local information. If the transaction passes the local validation phase, the Replication Manager encodes the transaction read-set (i.e., the set of identifiers of all the objects read by the transaction) in a Bloom Filter, and ABcasts it along with the transaction write-set (which is not encoded in the Bloom Filter). The size of the Bloom Filter encoding the transaction's read-set is computed to ensure that the probability of a transaction abort due to a Bloom Filter's false positive is less than a user-tunable threshold.

AGGRO

AGGRO (Palmieri, Quaglia, & Romano 2010) addresses the issue of how to enhance dependability of STM systems via replication and is an Optimistic Atomic Broadcast-based (OAB) active replication protocol that aims at maximizing the overlap between communication and processing through an AGGRessively Optimistic concurrency control scheme. The key idea underlying AGGRO is to propagate dependencies across uncommitted transactions in a controlled manner, namely according to a serialization order compliant with the optimistic message delivery order provided by the OAB service. This protocol is based on active replication schemes, but it has the feature of not requiring a-priori knowledge about read-/write-sets of transactions, but rather to detect and handle conflicts dynamically, i.e. as soon as (and only if) they materialize. AGGRO speculatively processes transactions exploiting the optimistic delivery order notifications provided by an OAB service.

3.4.2 Used Benchmarks

The results reported in this thesis, which aimed at infer the performance of the proposed protocols, were conducted using the benchmarks described in this section.

Bank Benchmark

Bank Benchmark is a synthetic workload obtained by adapting the benchmark originally used by Herlihy, Luchangco, & Moir (2006b). It is composed by a configurable array of "bank accounts" and it transfers a value from one account to the other. This benchmark was adapted to the framework used to test the protocols and augmented with more configuration options, which allowed to tune the conflict rate and measure the behavior of each protocol under controlled scenarios. It is possible to configure the benchmark so each replica uses a different subset of the array, generating no conflicts between replicas, or to share the full array among all the replicas, thus generating an high conflict rate.

STMBench7

STMBench7 (Guerraoui, Kapalka, & Vitek 2007) is a non trivial benchmark that features a number of operations with different levels of complexity over an object-graph with millions of objects. It can be configured to use the "read-dominated" workload, where 90% of the transactions are read-only, the "read-write workload", where 60% of the transactions are read-only and the "write-dominated" workload, that generates only 10% of read-only transactions. It can be also configured to generate long traversals, which generate long running transactions with large read-sets and represent 5% of the generated transactions.

Lee Benchmark

Lee-TM (Ansari, Kotselidis, Watson, Kirkham, Luján, & Jarvis 2008) is a parallel, STM-based, implementation of the Lee algorithm for routing junctions in a circuit. The Lee-TM generates a very heterogeneous workload encompassing a wide range of transactions duration and length. More in detail, the benchmark starts by routing the shortest junctions in the circuit, generating transactions whose local processing lasts just a few milliseconds, and then progressively lays junctions of increasing length, generating transactions whose local processing lasts up to a few seconds. The set of operations that is assigned to each replica can be calculated using a simple round robin strategy, or by assigning a subset of the circuit to each replica. This second strategy ensures some locality on the data accessed by the transactions and generates initially a small conflict rate. The conflict rate increases as each replica makes progress and needs to access the region of other replicas to finish the algorithm.

3.5 Summary

This chapter provided the rationale, and the needed system model, to understand the research directions explored in this thesis. It started with an enumeration of the key system properties and also of available building blocks for developing distributed and replicated STMs. It then made a brief survey of the main distributed STM systems, the motivation for leveraging on database replication protocols for building replicated STMs and identified the key challenges in such task. Finally, it identified a number of techniques that are worth exploring, paving the way for the work to be presented in the next chapters, and enumerated the benchmarks used in the thesis to evaluate the proposed protocols.

Notes

The results reported in this chapter were accomplished in cooperation with other members of the GSD and ESW research groups, namely Luís Rodrigues, Paolo Romano, João Cachopo, António Rito-Silva and Maria Couceiro. The motivation of this work was proposed in the paper "Versioned Transactional Shared Memory for the FénixEDU Web Application", Proceedings of the second Workshop on Dependable Distributed Data Management (in conjunction with Eurosys 2008), in Glasgow, UK, March 2008. The challenges presented in this chapter were identified in the paper "Towards Distributed Software Transactional Memory Systems", Proceedings of the Workshop on Large-Scale Distributed Systems and Middleware, Watson Research Labs, Yorktown Heights (NY), USA, September 2008. Finally, the author also contributed to the development of D^{2}STM, a system that has been first presented in the paper "D^{2}STM: Dependable Distributed Software Transactional Memory", Proceedings of the 15th Pacific Rim International Symposium on Dependable Computing, Shanghai, China, November 2009.

4 Asynchronous Lease-based Certification

As observed in the Section 3.4, the overhead of previously published certification schemes based on Atomic Broadcast (AB) can be a dominating factor in the performance of replicated STMs. Further, distributed certification schemes are based on an inherently optimistic approach: transactions are only validated at commit time and no bound is provided on the number of times that a transaction will have to be re-executed due to the occurrence of conflicts. This can lead to undesirably high abort rates in high conflict scenarios or with heterogeneous workloads that contain mixes of short and long-running transactions (as it is actually the case for several well-known TM benchmarks (Guerraoui, Kapalka, & Vitek 2007; Ansari, Kotselidis, Watson, Kirkham, Luján, & Jarvis 2008)). In this case, long-running transactions may be repeatedly aborted due to the occurrence of (remote) conflicts with a stream of short-lived transactions, leading to fairness violation that might be regarded as unacceptable by the users of interactive applications.

This chapter tackles the above issues by presenting the Asynchronous Lease Certification (ALC) protocol. In the core of the ALC scheme is the notion of *asynchronous lease*. Analogously to classic lease schemes (Duvvuri, Shenoy, & Tewari 2000; Gray & Cheriton 1989), asynchronous leases are used by a replica to establish temporary privileges in the management of a subset of the replicated data-set. Specifically, in ALC, the ownership of an asynchronous lease on a set of data items provides a replica with two key benefits: (*i*) reducing the commit phase latency of the transactions that access those data items and; (*ii*) sheltering transactions by repeated abortions due to remote conflicts.

While the ALC protocol may rely on any STM for locally regulating the concurrent execution of transactions, it was integrated with a multi-versioned STM, namely JVSTM (Cachopo & Rito-Silva 2006). This allows sheltering read-only transactions from the possibility of aborts (due both to local or remote conflicts), as well as to prevent them from incurring in stalls due to concurrent conflicting accesses. Through an extensive experimental evaluation, based on both

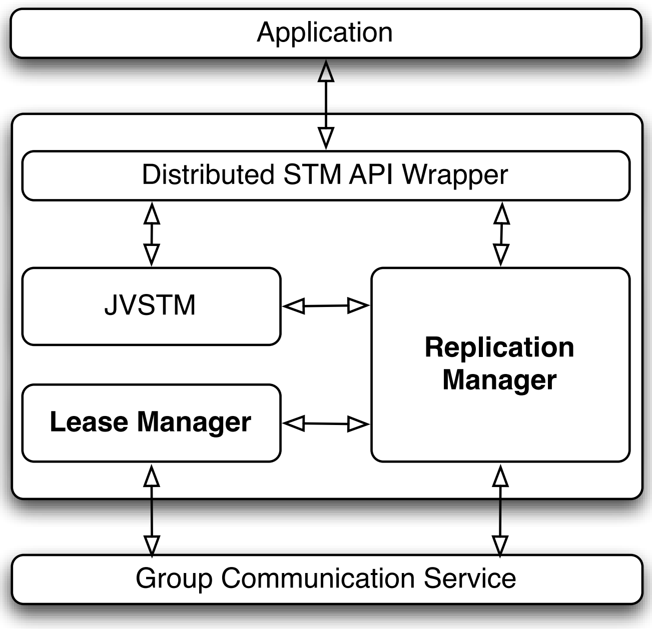


Figure 4.1: Architecture of an ALC replica.

synthetic micro-benchmarks, as well as complex STM benchmarks the results show that ALC permits to achieve higher throughput when compared with competing replicated STMs, such as D^{2}STM (Couceiro, Romano, Carvalho, & Rodrigues 2009).

The rest of this chapter is organized as follows. Section 4.1, describes the architecture of an ALC-based system and discusses the issues related to the integration with JVSTM. The ALC scheme is presented in Section 4.2 and Section 4.3 presents the results of a experimental evaluation study. Finally, Sections 4.4 and 4.5 conclude and sumarize this chapter.

4.1 The ALC Architecture

The architecture of the software running on each replica is illustrated in Figure 4.1. The top layer is a wrapper that intercepts the application level calls for transaction demarcation (i.e. to begin, commit or abort transactions), not interfering with the application accesses (read/write) to the transactional data items, which are managed directly by the underlying STM layer. This wrapper API interacts with the STM and Replication Manager layers. The STM layer can implement any STM protocol, but it was integrated with JVSTM (Cachopo & Rito-Silva 2006) in order to take advantage of the multiple versions, which ensure that read-only transactions are never aborted. The bottom layer is a Group Communication Service (GCS) (Chockler, Keidar, & Vitenberg 2001) which provides the view synchronous membership, Optimistic Atomic Broadcast (OAB) and Uniform Reliable Broadcast (URB) services. All these services are formalized in the Section 3.1.

The core components of ALC are the Lease Manager (LM) and the Replication Manager (RM). The role of the LM is to ensure that there are never two replicas simultaneously disseminating updates for conflicting transactions. To this end, the LM exposes an interface consisting of two methods, namely GETLEASE() and FINISHEDTRANSACTION(), which are used by the RM to acquire/free leases on a set of data items. The RM is responsible of managing the transactions commit phase, implementing a distributed certification scheme which leverages the local JVSTM replica to commit and certify local and remote transactions, as well as the services provided by the LM and the GCS.

Our target consistency criterion for replication is Extended Update Serializability (EUS) (Adya 1999), a consistency criterion based on Update Serializability (US). The US criterion was originally defined in (Hansdah & Patnaik 1986) (in terms of View Serializability (Hansdah & Patnaik 1986)) and later re-formalized by Adya (Adya 1999) (in the framework of conflict serializability), who also introduced EUS. Roughly speaking, US ensures that (*i*) when read-only transactions are removed from the history containing the transactions executed across the whole set of replicas, the resulting history is equivalent to a serial transaction execution history on a non-replicated system, and (*ii*) read-only transactions are always executed in a consistent snapshot of the data. More formally, US provides a semantic equivalent to classic 1-Copy Serializability (1CS) (Bernstein, Hadzilacos, & Goodman 1987) for update transactions, which guarantees the consistent evolution of the system's state. Analogously to 1CS, with US read-only transactions are also guaranteed to observe a snapshot equivalent to some serial execution (formally, a linear extension (Lamport 1978)) of the (partially ordered) history of update transactions. However, unlike 1CS, US allows concurrent read-only transactions to observe snapshots generated from different linear extensions the history of update transactions.

US is strictly weaker than 1CS, in the sense that US accepts a larger number of transaction schedules than 1CS. On the other hand, it is noteworthy to highlight that the only discrepancies in the serialization orders observable by read-only transactions are imputable to the re-ordering of update transactions that do not have any (direct or transitive) data dependency. In other words, the only perceivable discrepancies are associated with the ordering of logically independent concurrent events, which has typically no impact on the correctness of a wide range of real-world applications (Garcia-Molina & Wiederhold 1982). More precisely, the anomalies possible in US are detectable only by applications that allow direct communication between the replicas that executed read only transactions on the same data item. The relevance of US criterion stems from the fact that typical distributed applications rely entirely on the underlying layer to manipulate their shared state (Ansari, Kotselidis, Watson, Kirkham, Luján, & Jarvis 2008; Guerraoui, Kapalka, & Vitek 2007; Transaction Processing Performance Council 2002). Thus the adoption of US brings no harm to their correctness and, on the other hand, allows the design of high efficient protocols such as ALC.

EUS extends US semantic not only to transactions that commit, but to any executing transaction (even if it is later on aborted due to the detection of a non-serializable dependency). This sort of guarantees may be necessary to ensure that the application does not behave in an unexpected manner due to the observation of non-serializable snapshots (Guerraoui & Kapalka 2008b). If this happens, with US, the transaction can be aborted when it tries to commit. However, before the transaction reaches its commit point, the application program may behave in an unexpected manner, e.g., it may crash, go into an infinite loop, or output unexpected results. EUS is, indeed, a concept akin to opacity (Guerraoui & Kapalka 2008b), a safety criterion recently introduced in the area of (non-distributed) TMs, that formalizes analogous extended guarantees for serializable transactions (namely, 1-copy serializable, in a replicated environment such as the one considered in this paper). Note that since EUS rejects schedules in which transactions observe inconsistent snapshots and later abort (which are instead admitted by 1CS), 1CS and EUS are incomparable consistency criteria.

4.2 The ALC Protocol

For the sake of clarity, the ALC protocol is presented in an incremental fashion. First, it will be presented the baseline version that relies on a simple, yet quite inefficient, lease establishment scheme, in Section 4.2.1 and Section 4.2.2. Initially, it will be assumed that the set of data items accessed by transactions do not vary across different re-executions of a same transaction and show how to deal with the case of transactions accessing different sets of data items across different executions in Section 4.2.3. The Section 4.2.4 introduces three optimizations that permit to drastically reduce the communication latency associated with the lease transfer mechanism by achieving full overlapping with the distributed certification phase. Finally, Section 4.2.5 illustrates some examples of the protocol execution and the correctness arguments are described in Section 4.2.6.

The intuition behind the ALC approach is the following. Analogously to classic certification schemes, transactions are run based on local data, avoiding any inter-replica synchronization until they enter the commit phase. At this stage, however, ALC ensures to have established a lease for the accessed data items, prior to proceed with transactions validation. In case a transaction T is found to have accessed stale data, it is re-executed without releasing the lease. This ensures that, during T's re-execution, no other replica can update any of the data items accessed during the first execution of T, guaranteeing the absence of remote conflicts on the subsequent re-execution of T (provided that this deterministically accesses the same set of data items accessed during its first execution).

The ownership of the lease ensures that no other replica will be allowed to validate any conflicting transaction, making it unnecessary to enforce distributed agreement on the global transactional serialization order. ALC takes advantage of this by limiting the use of OAB exclusively for establishing the lease ownership. Subsequently, as long as the lease is owned by the replica, transactions can be locally validated and their updates can be disseminated using URB, which can be implemented in a much more efficient manner than OAB.

Unlike classic lease based approaches, where the lease duration is defined at the time of the lease establishment, in ALC leases are said to be asynchronous since the concept of lease is detached from the notion of time. Conversely, once that a replica acquires a lease on a set of data items, it holds the lease as long as it does not require an explicit lease request from another replica. In order to avoid distributed deadlocks during the lease acquisition phase, lease requests are disseminated via OAB, and atomically enqueued at each node in the TO-delivery order. Fairness is ensured by establishing leases in FIFO order and leases are transferred to a requesting replica as soon as the transactions (in execution at the lease-owner) to which those leases had been granted have committed.

Algorithm 1: ALC Replication Manager.

```
boolean commit(Transaction T)
  if (¬JVSTM.validate(T)) then // early validation
    JVSTM.abort(T)
    return false
  DataSet dataSet = JVSTM.getReadAndWriteSet(T)
  LeaseRequestID leaseID =
        LeaseManager.getLease(dataSet)
  // final validation
  if (leaseID=⊥ ∨ ¬JVSTM.validate(T)) then
    JVSTM.abort(T)
    return false
  else
    WriteSet ws = JVSTM.getWriteSet(T)
    trigger UR-broadcast([ApplyWS,T,leaseID,ws])
    wait until (committedXact(T) ∨ ejected)
    if ( ejected ) then
      JVSTM.abort(T)
      return false
    else
      return true

upon event UR-deliver([ApplyWS,T,leaseID,ws]) from $p_j$ do
  if ($p_j = p_i$) then
    JVSTM.commitLocalXact(T)
    trigger committedXact(T)
    LeaseManager.finishedTransaction(leaseID)
  else
    JVSTM.commitRemoteXact(ws)
```

4.2.1 Replication Manager

As already stated, transactions are executed locally, without any inter-replica synchronization, until the commit phase is reached. At this stage, if the committing transaction did not issue any write operation, it can be locally committed given that the JVSTM multi-versioned concurrency control scheme ensures the serializability of the observed snapshot. On the other hand, if the transaction is not read-only, the STM API wrapper invokes the commit method of the Replication Manager, triggering the execution of the ALC protocol.

The pseudo-code describing the behavior of the RM is shown in Algorithm 1. Following an early validation phase, aimed at detecting any conflict developed with (local or remote) transactions already committed since the activation of the committing transaction, the RM requires the LM to acquire the leases corresponding to the set of data-items read and written during the transaction execution. The lease acquisition phase (described in the following section) eventually terminates returning either a lease identifier, or the special value $\perp$ notifying the RM about the impossibility to acquire the requested leases. As it will be seen, the only case in which the LM ever fails to acquire leases is when the process is excluded from the primary component view (due to a wrong failure suspicion). In such a case, for the RM it is only safe to keep on processing read-only transactions, and will therefore abort the current transaction. On the other hand, in absence of failures or failure suspicions, the lease manager will eventually succeed in acquiring the requested set of leases and return a lease request identifier to the RM. In this case, p_i is guaranteed to have already installed the updates of every remotely (and locally) executed transaction, and can therefore proceed with the validation. If this is successful, the transaction's write-set (and the corresponding lease request identifier) is disseminated using URB.

The properties of URB ensure that if p_i self-delivers the transaction's write-set in the current view, any other v_i-correct process will also deliver it in view v_i (even if p_i is subject to a failure right after the write-set delivery). This allows to safely commit the local transaction. Finally, the RM informs the LM of the successful execution of the transaction by invoking the FINISHEDTRANSACTION method specifying, as input parameter, the identifier of the lease request previously returned by the GETLEASE method.

The RM is also responsible of applying the write-set of remotely executed transactions, which are triggered by the corresponding *UR-deliver*. Note that the Causal Order property of the primitive ensures that the sequence of local transactions committed by a process p_i is delivered in FIFO order (i.e. in the same order in which p_i committed them) by any replica that deliver them.

4.2.2 Lease Manager

The LM's pseudo-code for process p_i is depicted in Algorithm 2 and Algorithm 3. Let us start by analyzing the pseudo-code in Algorithm 2, which represents the core of the lease establishment protocol. As already hinted, in order to establish/relinquish leases, the LM exposes two interfaces, namely the GETLEASE and FINISHEDTRANSACTION methods. Leases are associated with data items indirectly, namely through conflict classes. This allows to flexibly control the granularity of the leases abstraction. The mapping between a data item and a conflict class (which can in practice be implemented through classic hashing schemes since each transactional object is already uniquely identified) is abstracted through the `getConflictClasses`() primi-

Algorithm 2: ALC Lease Manager at process p_i: basic algorithm.

```
FIFOQueue<LeaseRequest>
 CQ[NumConflictClasses]={⊥,...,⊥}
View currentView={$p_1$,...,$p_i$,...,$p_n$}
boolean inPrimaryComponent=true

LeaseRequestID getLease(Set DataSet)
 if (¬inPrimaryComponent) then return ⊥
 ConflictClass[] CC = getConflictClasses(DataSet)
 if (∃req∈CQ s.t. req.proc=$p_i$ ∧ ¬ req.blocked
     ∧ (∀cc∈CC : cc∈req.cc) ) then
   req.activeXacts++
 else
   LeaseRequest req = new LeaseRequest($p_i$,CC)
   trigger OA-broadcast([LeaseRequest,req])
 wait until isEnabled(req) ∨ ¬inPrimaryComponent
 if (¬inPrimaryComponent) then return ⊥
 else return req.getID()

void finishedTransaction(LeaseRequestID reqID)
 getLeaseReqFromId(reqID).activeXacts−−

upon event TO-deliver([LeaseRequest, req]) from $p_k$ do
 freeLocallyEnabledLeases(req.cc)
 ∀ cc∈req.cc do CQ[cc].enqueue(req)

upon event UR-deliver([LeaseFreed, reqs]) from $p_k$ do
 ∀req∈reqs do
   ∀ cc∈req.cc do CQ[cc].dequeue(req)

void freeLocalLeases(LeaseRequest req)
 Set<LeaseRequest> locallyEnabledLeases
 ∀req$^\dagger$ ∈CQ s.t. req$^\dagger$.proc=$p_i$∧ (req$^\dagger$.cc∩req.cc)≠ ∅ do
   req$^\dagger$.blocked=true
   if (req$^\dagger$.isEnabled()) then
     locallyEnabledLeases=locallyEnabledLeases ∪ req$^\dagger$
 if (locallyEnabledLeases ≠ ∅) then
   wait until ∀ req* ∈locallyEnabledLeases : req*.activeXacts=0
   trigger UR-broadcast([LeaseFreed,locallyEnabledLeases)]

boolean isEnabled(LeaseRequest req)
 return ∀cc∈req.cc : CQ[cc].isFirst(req)
```

tive, taking a set of data items as input parameter and returning a set of conflict classes. The trade-off between coarse and fine lease granularity is in that coarse granularity is prone to false sharing, i.e. lease requests associated with disjoint data sets may be mapped to common conflict classes, generating unnecessary lease migrations across replicas. On the other hand, fine granularity schemes may generate larger communication and processing overhead, since they impose the transmission of larger lease request messages among replicas and the management of larger local data structures for detecting conflicts among lease requests.

The data structures maintained by replicas for regulating the establishment/release of leases are the following: CQ, namely an array of FIFO queues, one per conflict class, that serves as a lock table to keep track of the conflict relations among lease requests; $currentView$, namely the set of processes belonging to the current view; $inPrimaryComponent$, a boolean flag which indicates whether p_i is in the primary component or not. A `LeaseRequest` type is a structure containing the following fields: *cc*, namely the set of conflict classes associated with the lease request; $activeXacts$, an integer keeping track of the number of active transactions associated with the lease request, which is initialized to 1 when a lease request is created; *blocked*, a boolean variable indicating whether new transactions can be associated with this lease request or not, which is initialized to $false$ when a lease request is created; a unique identifier, which is transparently generated by p_i and is retrievable through the `getID`() primitive.

When the GETLEASE() method is invoked by the RM to establish a lease on the set of data items accessed by a committing transaction, the LM first checks whether p_i has already been ejected from the primary component. In this case it returns the special value $\perp$, notifying the RM that it is currently impossible to establish new leases. Otherwise, it determines, through the `getConflictClasses`() primitive, the set of conflict classes associated with the data-sets accessed by the transaction. Then it checks whether p_i has already enqueued in CQ a lease request *req* (*i*) associated with a super-set of the currently requested conflict classes, and (*ii*) which can still be associated with additional transactions (i.e. whose *blocked* field is set to false). In this case, it is not necessary to issue a new lease request, and the current transaction can simply be associated with *req*. Otherwise, a new lease request is created and *OA-broadcast*. In both cases, p_i waits either until the corresponding lease request is enabled (this happens when the lease request reaches the first position in all the FIFO queues associated with its conflict classes - see the `isEnabled`() function), or until p_i is ejected from the primary component. In the latter case, the LM returns the special value $\perp$. If the lease request is eventually enabled, on the other hand, its unique identifier is retrieved via the `getID`() primitive and returned to the RM.

The FINISHEDTRANSACTION() method takes as input parameter a lease request identifier (i.e. the identifier previously returned by the GETLEASE() method when a lease request was associated with the transaction), retrieves the corresponding lease request via the `getLeaseReqFromId`() primitive, and decrements the number of active transactions associated with the lease request.

Upon a *TO-deliver* event of a lease request *req*, p_i first of all checks whether some of his locally

Algorithm 3: ALC Lease Manager at process p_i: dealing with view changes.

```
upon event ViewChange(View newView) do
 if (¬ inPrimaryComponent
       ∨ $p_i$ joining group for the first time) then
    perform state transfer
    inPrimaryComponent=true
 else
    ∀$p_j$ s.t. ($p_j$ ∈currentView ∧ $p_j$ ∉ newView) do
       ∀ req∈CQ s.t. req.proc=$p_j$ do CQ.remove(req)
 currentView = newView

upon event ejected do
 inPrimaryComponent=false
```

issued lease requests need to be freed or blocked. This is done by invoking the `freeLocalLeases` procedure which, determines whether there is any of p_i's lease requests (denoted as req† in the pseudo-code) already enqueued in CQ which conflicts with *req* (i.e. whether req and req† have at least a conflict class in common). In this case, it sets the blocked field of these lease requests to true. This is the key mechanism employed to ensure the fairness of the lease rotation scheme: in order to prevent a remote process p$_j$ from starving while waiting for process p_i to relinquish a lease, in fact, p_i is prevented from associating new transactions with existing lease requests as soon as a conflicting lease request from p_j is *TO-delivered* at p_i (as explained while describing the GETLEASE method). Next, the LM waits for the successful completion of every transaction associated with any locally issued conflicting lease request that is also currently enabled (note that this implies that such transactions have been already allowed to proceed with the validation phase). When these transactions have successfully committed, the LM triggers a *UR-broadcast* specifying the set of locally owned lease requests that p_i is freeing. The handling of the *TO-deliver* event terminates by enqueueing the corresponding lease request in every associated conflict class.

The logic associated with *UR-deliver* events is very simple: every lease request specified in the uniformly broadcast message is removed from the corresponding conflict class queues.

View Changes

It remains to discuss the replicas behavior in the presence of view changes and ejections from the primary component view, which is formalized by the pseudo-code in Algorithm 3. Upon delivery of a `ViewChange` event, if the replica re-joins the primary component or is joining the group of replicas for the first time, it triggers a state transfer procedure that realigns the content of the local replica of the STM, as well as of the state variables of the ALC protocol. The details of the state transfer procedure are not detailed here, since it is out of the scope of this dissertation. A conventional state transfer mechanism, such as de one described by Jiménez-Peris, Patinõ Martínez, & Alonso (2002) may be used at this purpose. On the other hand, if upon a view change, some processes are eliminated from the current view (because they have crashed or are partitioned away from the primary component), all of their lease requests are purged from the local CQ. Recall also that, if a process gets disconnected from the primary component, it will fail to deliver any pending lease request. This will cause the failure of the lease acquisition procedure at this replica (see the GETLEASE method). Overall, these two mechanisms (the removal of lease requests issued by processes excluded from the primary component, and the failure of the acquisition of lease requests pending at a process that is ejected from the primary component) guarantee the liveness of the lease management protocol. Note also that replicas outside of the primary component may still continue processing read-only transactions, which will observe a serializable, albeit possibly obsolete, snapshot of the replicated STM.

4.2.3 Non-deterministic Re-executions

The above presented lease management scheme guarantees the absence of remote conflicts during the re-execution of a transaction as it avoids releasing the lease on the conflict classes accessed during the previous execution of the transaction until this is successfully commit. This scheme can deterministically guarantee the absence of remote conflicts only if the set of conflict classes accessed when re-executing the transaction do not vary. While this is not always true in general for real applications, on the other hand it is very likely (as also supported by the experimental evaluation) that two re-executions of the same transaction access a large number of conflict classes in common (especially if lease granularity is moderately coarse). In practical settings, therefore, the presented ALC scheme is still very likely to significantly reduce the transactions abort rate. A simple, albeit somewhat extreme, workaround to deterministically bound the number of aborts/re-runs undergone by "problematic" transactions dramatically altering their data access patterns upon re-execution would consist in requesting a lease on the whole set of conflict classes. This would clearly suffice to ensure their successful re-execution, at the price of a temporary, though significant, bridling of concurrency.

Algorithm 4: ALC Lease Manager: optimistic delivery optimization.

```
upon event Opt-deliver(LeaseRequest request) from $p_k$ do
  freeLocalLeases(request)

upon event TO-deliver(LeaseRequest request) from $p_k$ do
  foreach $cc \in request.cc$ do
    CQ[cc].enqueue([$p_k$,request])
```

Finally, it is important to highlight that the scheme presented in Section 4.2.2 can suffer of deadlocks in case the conflict classes accessed during transactions re-execution, say cc', are not a subset of those accessed during a previous execution, say cc. This is due to the fact that the LM won't relinquish the lease on cc granted during the first transaction's execution, and will issue a new lease request on cc'. The latter may block if some other replica is simultaneously retaining the lease on cc' while requesting a lease on cc.

Fortunately, such an issue can be resolved by using simple and lightweight deadlock avoidance or detection schemes. A possible deadlock avoidance scheme is to detect whether $cc' \nsubseteq cc$ as a transaction completes its re-execution, and to piggyback a LeaseFreed message to the lease request OA-broadcast for cc'. An alternative deadlock detection scheme could check for the presence of cycles in the wait-for graph of the lease requests locally enqueued in CQ, and use a deterministic rule for breaking the cycle by aborting one of the involved lease requests. Note that as the state of the CQ is consistently replicated by all replicas, the deadlock detection would not require any additional inter-replica coordination.

4.2.4 Analysis and Optimizations

This section presents several simple optimizations to the basic ALC protocol. The description of the optimizations are divided into two different sections. The first section shows a set of optimizations that aim to reduce the number of communication steps needed in the worst case scenario for this protocol (when a new lease request must be issued). After, it is described how can the performance in some workloads be optimized, by distinguishing between read leases and write leases.

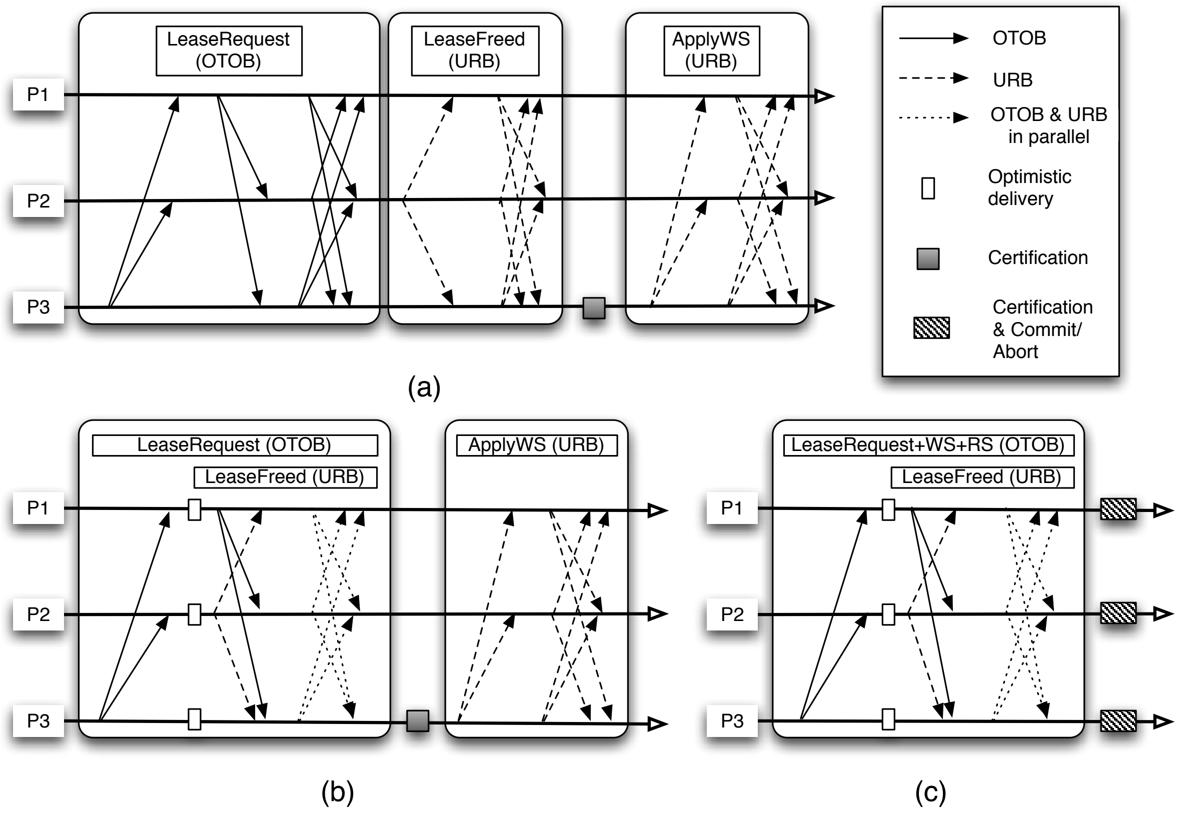


Figure 4.2: Message patterns for the ALC protocol.

Overlapping Messages

Provided that a replica owns a lease on the conflict classes accessed by a transaction, ALC allows committing the transaction using a single URB, which can be implemented incurring in a two communication steps latency (Guerraoui & Rodrigues 2006). This is in contrast with state of the art distributed certification schemes (Pedone, Guerraoui, & Schiper 2003), which incur in the latency of (at least) an AB during the commit phase (whose latency is of at least 3 communication steps latency[1]). On the other hand, in the presented ALC scheme, if a transaction has accessed data items for which its process does not hold a lease, it incurs in the latency associated with lease acquisition phase.

Figure 4.2 depicts the message patterns for the ALC protocol where $P3$ is requesting a lease owned by $P2$, as follows: (a) the baseline ALC protocol, (b) the optimization exploiting

[1]The only exception being AB protocols such as (Vicente & Rodrigues 2002) which, relying on additional system assumptions - such as the existence of a bound Δ on the minimum inter-arrival time of messages at the replicas, achieve a latency of to 2+Δ communication steps.

optimistic deliveries to free the leases, and (c) the optimization that piggybacks the read-set and write-set on the LeaseRequest message. As depicted in the Figure 4.2 (a), this entails one AB to deliver the lease request, plus one URB for delivering the lease granted messages, yielding a total latency of 5 communication steps. Including the final URB for the dissemination of the transactions write-set, the total number of communication steps is 7.

Two optimizations can be employed to reduce to just 3 communication steps latency the cost required for both committing a transaction and acquiring the corresponding lease. The first optimization, reported in Algorithm 4 and depicted in Figure 4.2 (b), consists in exploiting the *Opt-deliver* of the lease request (which incurs in a single communication step latency (Kemme, Pedone, Alonso, & Schiper 1999)) to immediately trigger the relinquishment of the required leases at a remote node (and the corresponding *URB* of a LeaseFreed message). This is safe since, even in the case of mismatches between the optimistic and the final delivery of two conflicting lease requests at some node p_i, the net effect would be anyway to trigger the relinquishment of the leases currently owned by p_i. This allows to totally overlap the execution of the OAB for the lease request and the URB for the lease granted, reducing to three communication steps the latency of the lease acquisition phase.

The second optimization consists in OA-broadcasting the set of data items accessed by a transaction T while issuing a lease request, rather than the corresponding conflict classes. This would allow each replica to validate T as soon as the corresponding lease request gets locally established, thus avoiding the *URB* of the transaction's write-set and reducing the latency for committing T to three communication steps (see Figure 4.2 (c)).

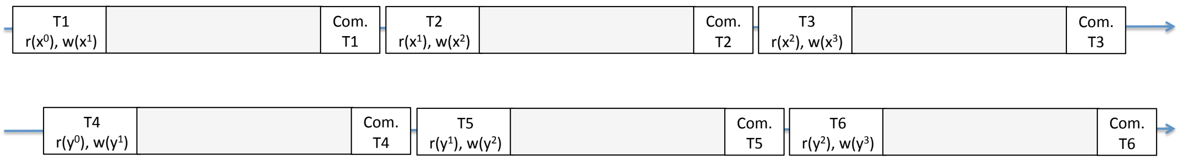
Read-Write Leases

The ALC protocol enables the requests by the same order in all the replicas. On each replica there is only one request enabled at a time, allowing the owner of that request to execute transactions without the risk of reading stale data. This happens even if the conflict classes of two requests overlap only in the leases of objects that were read and not written. One optimization that was made in the ALC protocol was to distinguish between requests for read and write leases.

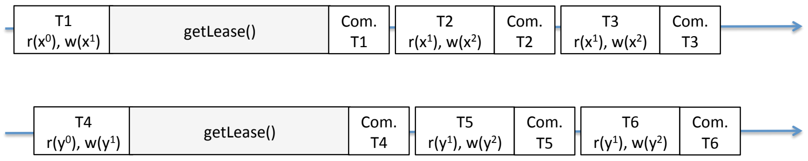
The pseudo-code in Algorithm 5 formalizes this optimization, by redefining, in a modular fashion, the logic of the `isEnabled` method. Essentially, a lease request *req* can be enabled either

Algorithm 5: ALC Lease Manager: Read-Write leases optimization.

```
boolean isEnabled(LeaseRequest req)
  if ∀cc∈req.cc : CQ[cc].isFirst(req) ∨
     ∀req_i ∈ CQ s.t. CQ[cc].index(req_i) < CQ[cc].index(req) :
     cc ∈ req_i.getReadSet() ∧ cc ∈ req.getReadSet() ∧
     cc ∉ req_i.getWriteSet() ∧ cc ∉ req.getWriteSet() then
        return true
  else
        return false
```



(a) Conventional certification protocol.



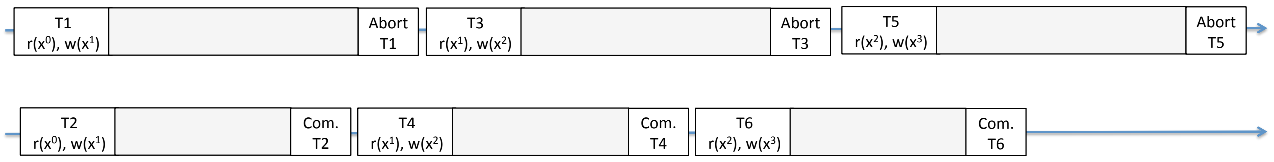
(b) ALC.

Figure 4.3: ALC execution sketch: parallel executions.

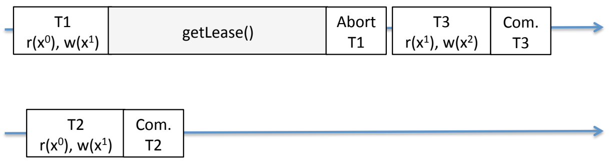
if it reaches the first position in the queue associated with the requested conflict classes (as in the original specification of the `isEnabled` method), but also if, for all the conflict classes for which (*i*) *req* requested a read lease and (*ii*) if *req* is not the first in queue in the corresponding queue, all the preceding lease requests are also for read leases. As it will be seen in Section 4.3, this simple optimization can be extremely effective in presence of workloads in which transactions have a non-minimal probability of having non-overlapping read-sets and write-sets, allowing to enhance the degree of concurrency and reducing the frequency of lease circulation across the replicas.

4.2.5 Example Execution Sketches

To illustrate the dynamics of the ALC protocol, this section shows two sketches of its execution, contrasting it with analogous execution sketches for a conventional certification protocol, such as the ones of Couceiro, Romano, Carvalho, & Rodrigues (2009) or Pedone, Guerraoui, & Schiper (2003). The Figure 4.3 shows the execution of two replicas, each one reading and writing



(a) Conventional certification protocol.



(b) ALC.

Figure 4.4: ALC execution sketch: transaction re-execution.

in different data items. In the conventional certification protocol (as shown in the Figure 4.3(a)) each transaction must be sent to the other replicas using AB so each transaction is ensured to be globally certified. Using the ALC protocol, replicas just need to ask for a lease for the data items on the first transaction. As shown in the Figure 4.3(b), the following transactions can safely be broadcast and committed without the need for the usage of Atomic Broadcast to synchronize the replicas. ALC avoids the use of heavy communication procedures, generating a higher throughput.

The second execution sketch is illustrated in the Figure 4.4 and shows how ALC ensures that long running transactions are able to make progress, even in the presence of shorter concurrent transactions. In the Figure 4.4(a), $T1$ is executed in a replica, and aborted due to the commit of $T2$ in a different replica. The application tries to re-execute the transaction, which is reincarnated as $T3$, also being aborted by concurrent transactions running on a different replica. With the ALC protocol, illustrated in the Figure 4.4(b), $T1$ is executed without being the lease owner. This is a normal case for ALC. Upon commit, the thread requests the leases for the transaction data items, certifies the transaction and aborts, due to a remote conflict. At this point, however, ALC holds the lease to allow the transaction re-execution. Since this transaction is reading and writing the same data items, and the replica currently owns the lease, the re-executed transaction will commit for sure, allowing both short and long running transactions to make progress.

4.2.6 Correctness Arguments

In this section we present a series of arguments concerning the correctness of ALC with respect to the update serializability (US) criterion. This isolation level ensures classic 1CS guarantees (that is equivalent to a non-replicated serial execution (Bernstein, Hadzilacos, & Goodman 1987)) on the history of committed update transactions, denoted as H^{up}. On the other hand, unlike 1CS, it allows different read-only transactions to observe different serial schedules of H^{up} or, more formally, different linearizations of the partial order defined by H^{up}.

In order to show that ALC guarantees US we will start by showing that the history restricted to update transactions is 1CS. Then we will discuss correctness of read-only transactions.

Lemma 1. *The history of update committed transaction, H^{up}, generated by ALC is 1CS.*

Proof: As a preliminary step, we show that the stream of write-sets associated with transactions accessing non-disjoint data items sets are applied in the same order at all replicas. In order for a transaction to propagate its write-set, it first needs to acquire a lease on the data items it read/wrote. The enqueuing of lease requests at the various replicas takes place in a common order, namely the one determined by the final delivery of OAB. Further, the logic for the advancement of the lease requests in the conflict classes queues is deterministic, and the sequence of ApplyWS and LeaseFreed messages is disseminated via URB, which ensures causally ordered delivery.

This guarantees the following two properties:

P.1 the stream of write-sets associated with transactions accessing non-disjoint data items sets are all applied in the same order at all replicas.

P.2 the order of dequeuing from the the conflict classes queues for each pair of conflicting lease request is the same at all replicas. Being the order of lease enqueuing globally agreed via the OAB primitive, it follows that the order of advancement of lease requests that conflict, either directly or transitively, is the same at all processes in Π.

In order to show that the history of committed update transactions H^{up} generated by ALC is 1CS, it remains to show that the reads performed by a committed transaction T in H^{up}, have observed the most recent snapshot produced by any conflicting transaction that precedes T in H^{up}. This stems from the fact that, whenever an update transaction T requests a commit at process p_i:

- it acquires a lease on the data items that it read/wrote. This guarantees that any conflicting update transaction (possibly running at a different replicas) will block on the lease acquisition phase until p_i explicitly relinquishes the lease currently associated with T, and will therefore be serialized (whether possible) after T.

  It also guarantees, based on properties P.1 and P.2, that p_i has already applied, for the data items read/written by T, all the updates produced by committed update transactions preceding T in H^{up} (i.e. by any transaction that has previously acquired a lease on any of the data items read/written by T).

- it performs a validation of its read-set. Based on the first bullet, allows detecting any conflict that T developed with any previously committed update transactions in H^{up}.

Q.E.D.□

Lemma 2 *A read-only transaction observes a linear extension of* H^{up}.

Proof: The update transactions that are committed by an ALC replica, say p_i are applied sequentially on the local JVSTM instance. Each committed update transactions, independently of whether it has executed locally or remotely, is assigned a monotonically increasing *commitTimestamp* via the COMMITLOCALXACT() or the COMMITREMOTEXACT() primitives (see Section 4.2.1).

The (total) order of commit of the update transactions at replica p_i, which we denote as H_i^{up}, corresponds to a linearization of the partially order history of committed update transactions, H^{up}, generated by ALC. ALC exploits JVSTM's MVCC-based concurrency control scheme in order to serialize any (read-only) transaction executing (possibly concurrently on different threads) on a given ALC replica p_i, according to H_i^{up}. *Q.E.D.*□

On the other hand, the history H obtained by adding to H^{up} the read-only transactions executing at any replica p_i is not 1CS. In fact, given that only conflicting transactions are totally ordered in H^{up}, it follows that two different replicas p_i and p_j $(i \neq j)$ may perceive two distinct linear extensions, respectively H_i^{up} and H_j^{up}, of history H^{up}. In other words, two different replicas p_i and p_j $(i \neq j)$ may order differently two non-conflicting update transactions, say T_{u1} and T_{u2}, respectively in H_i^{up} and H_j^{up} (for instance, $T_{u1} \rightarrow T_{u2} \in H_i^{up}$, and $T_{u2} \rightarrow T_{u1} \in H_j^{up}$). Finally, by encapsulating JVSTM and integrating it an US replication protocol, ALC inherits JVSTM strong atomicity property, and exploits its opacity guarantees to achieve EUS without the need for any additional mechanism.

4.3 Performance Evaluation

This section reports the results of an experimental study aimed at quantifying the performance gains achievable by the proposed ALC protocol with respect to state of the art transactional replication schemes. The baseline protocol is D^{2}STM (Couceiro, Romano, Carvalho, & Rodrigues 2009), described in Chapter 3, which is referred to as CERT. Analogously to ALC, CERT allows replicas to process transactions locally, avoiding any form of synchronization during transaction execution. This protocol permits to achieve better scalability than pessimistic approaches (Kemme, Pedone, Alonso, & Schiper 1999) that force all replicas to process every update transactions, does not rely on a-priori knowledge on transactions data access patterns and requires a single Atomic Broadcast to disseminate the read-set and write-set of a certifying transaction.

The ALC protocol has been implemented with all the optimizations described in Section 4.2.4. In order to prevent the possibility of incurring in deadlocks in the presence of transactions altering their data access pattern during transaction execution, the simplest deadlock avoidance scheme among those previously described in Section 4.2.3 has been implemented: the *LeaseFreed* message is piggybacked to the lease request message, OA-broadcast during the commit phase of the re-started transaction if the set of conflict classes accessed is not a subset of those accessed during its former execution. All the results reported in the following were obtained by setting the conflict class granularity to coincide with a single data item.

The prototypes of ALC and CERT have been deployed on a cluster of 8 nodes, each one equipped with two Intel Quad-Core XEON at 2.0 GHz, 8 GB of RAM, running Linux 2.6.32-26-server and interconnected via a private Gigabit Ethernet. The benchmarks used on the

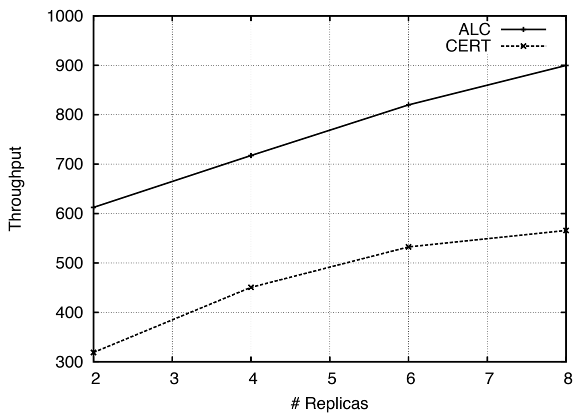


Figure 4.5: Bank benchmark running ALC: best case scenario.

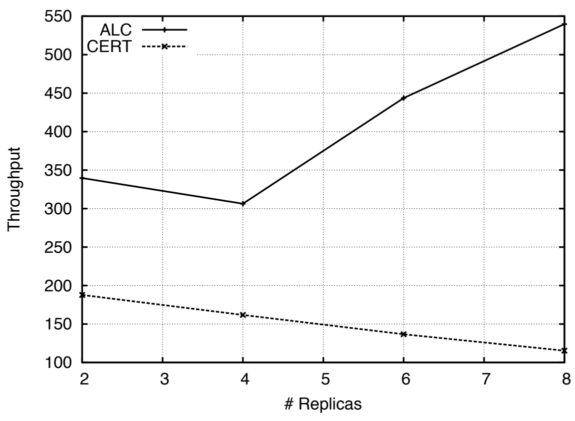
performance evaluation are described in the Section 3.4.2.

4.3.1 Bank Benchmark

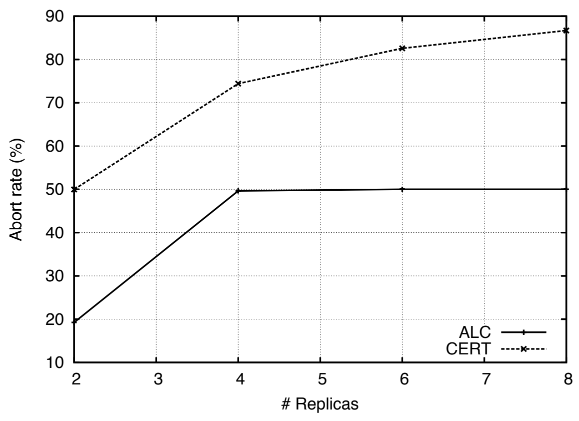
Lets consider first the Bank Benchmark, which serves for the purpose of quantifying the performance of the ALC scheme in two extreme scenarios for what concerns conflicts. In detail, the STM was initialized at each replica with an array of *numMachines*·2 items. In the first scenario, each machine reads and updates a distinct fragment of the array, thus never generating conflicts. In the second scenario, all the machines read and update the same data items, thus always conflicting.

Figures 4.5 and 4.6 show the throughput (committed transactions per second) and the abort rate as the number of nodes in the system varies. In the scenario with no conflicts (Figure 4.5), when using ALC, replicas disseminate transactions exclusively via URB (after establishing the lease upon their first transaction). This allows ALC to achieve a throughput up to 1,5x higher than CERT, which, requiring one AB per committed transaction, puts a significantly higher load on the GCS (which represents the bottleneck in this benchmark, being the transaction's logic extremely lightweight) especially as the number of replicas increase.

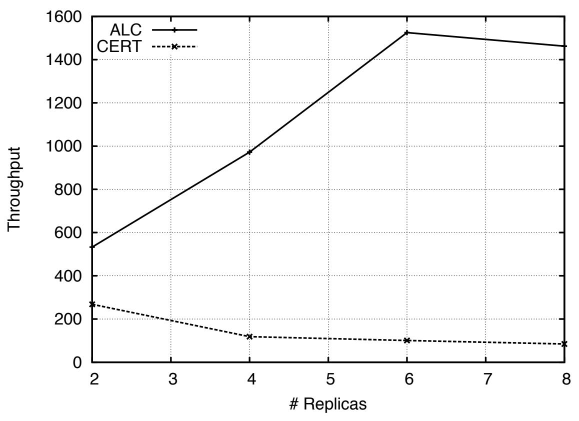
The high conflict scenario (Figure 4.6) represents a worst case scenario for ALC, since leases are constantly rotated across the replicas, and a lease request must always be OA-broadcast. Nevertheless, ALC's throughput is up to 3,6x higher with respect to CERT. This is explained by observing that, with CERT, the percentage of transactions that abort is significantly larger than with ALC. When more replicas are added, the throughput of ALC increases, showing that



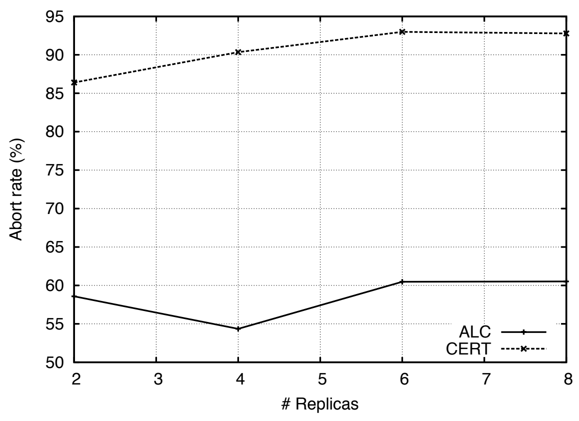
(a) Throughput (1 thread).



(b) Abort rate (1 thread).



(c) Throughput (4 threads).

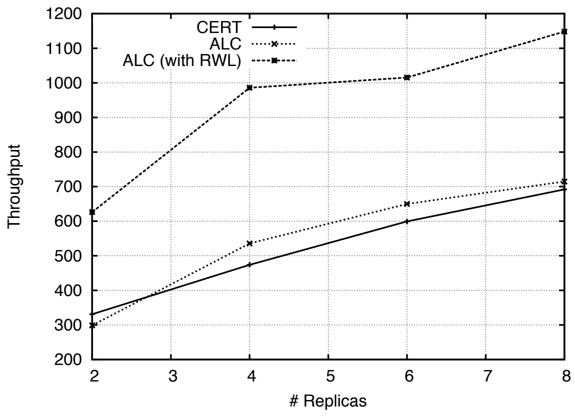


(d) Abort rate (4 threads).

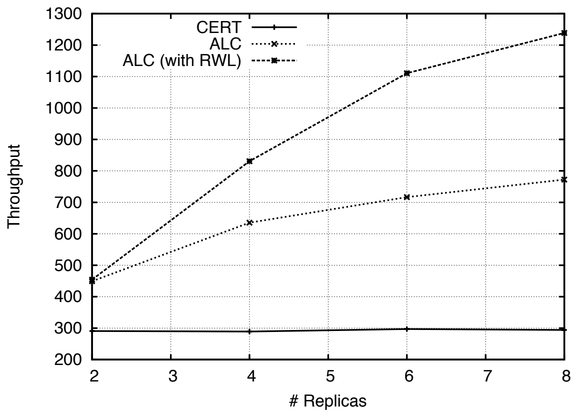
Figure 4.6: Bank benchmark running ALC: worst case scenario.

it can scale with the number of replicas. In the 8 replicas scenario, for instance, transactions are re-executed on average around 10 times before committing with CERT. On the other hand, ALC ensures that a transaction can be aborted at most once, as also proved by the fact that the abort rate for ALC never grows larger than 50% independently of the degree of global concurrency, as shown in the Figure 4.6(b). When the prototype is configured to run with more than one thread on each replica, ALC is not able to maintain the upper bound of 50% on the abort rate, since transactions can be aborted due to local conflicts, but ALC is still able to achieve lower rates (at most 60%) than the baseline protocol, as shown in the Figure 4.6(d).

Finally, the Figure 4.7 depicts the results achieved with the Read-Write Lease (RWL) optimization, described in the section 4.2.4. For this case, a scenario was built such that all replicas execute transactions that intercept only on the read-set. In this scenario, all the transactions write a disjoint set of items, but always read a common set of items. The CERT protocol



(a) Throughput (1 thread).



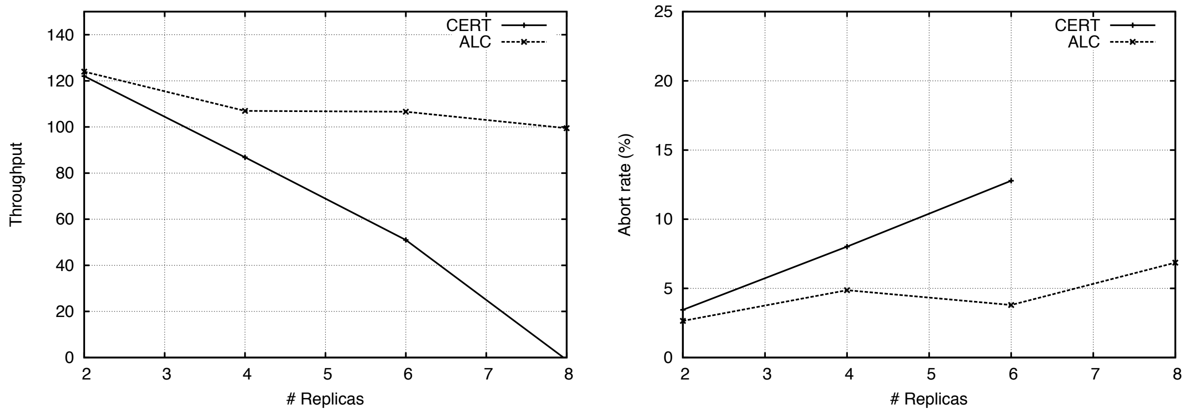
(b) Throughput (4 threads).

Figure 4.7: Bank benchmark running ALC: Read-Write Leases.

needs to send always one AB for each transaction. The ALC protocol also needs to send a new LeaseRequest every time it does not have the lease because of the interception of the read-set. Note also that the abort rate in the scenario is always zero, since there are no read-write conflicts between transactions. The Figure 4.7(a) shows the throughput of the system configured to run 1 thread on each replica. With only one thread, ALC without the RWL optimization achieves a throughput similar to CERT, since it must also use AB for LeaseRequests. When the RWL optimization is turned on, the enqueued request can already be enabled, even if it's not the first in the queue, as long as all the previous requests also enqueued a read-lease for the same set of read items. The Figure 4.7(b) shows the throughput of the system configured to run 4 threads on each replica. With 4 threads producing new transactions, CERT is already saturated and an increase of nodes does not reflect an increase in the throughput. The ALC protocol is able to achieve better throughput, even without the RWL optimization, since there is locality in the objects read and written by transactions that run on the same replica. When the RWL optimization is turned on, enqueued requests can be enabled earlier, increasing even more the system throughput.

4.3.2 STMBench7

This section considers a complex benchmark, namely the STMBench7 benchmark. ALC has been configured to use all the optimizations described in the Section 4.2.4. The benchmark was configured to use the "read-write workload" where 60% of the transactions are read-only and



(a) Throughput. (b) Abort rate.

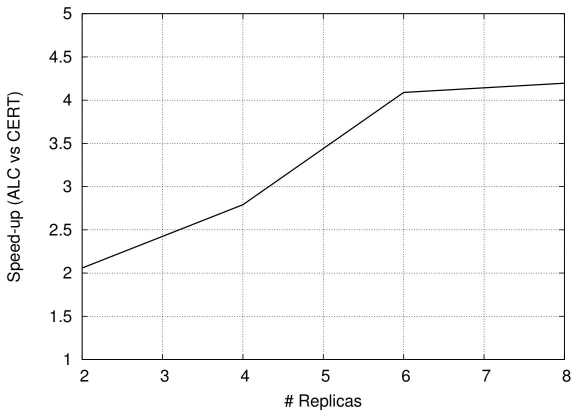
Figure 4.8: STMBench7 running ALC.

generates long running transactions with large read-sets. On each replica, there are 2 threads executing transactions and the number of replicas varied between 2 and 8. This shows the system behavior in a complex benchmark running in a setup that has a mix of local and global concurrency.

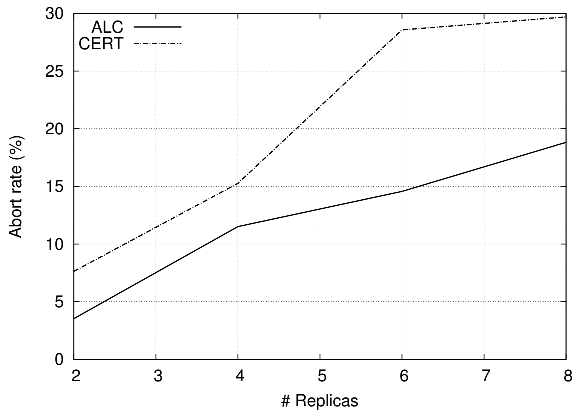
The Figure 4.8 depicts the throughput and abort rate when executing the ALC and CERT protocols on the described setup. As shown in the Figure 4.8(a), the performance of the CERT protocol decreases linearly with the number of replicas. The CERT protocol sends an AB for each transaction and the setup has more than one thread executing transactions on the same node. The results of CERT in the 8 replicas scenario are not reported because this kind of scenarios saturates the Group Communication Service and, with the CERT protocol, the benchmark is not able to finish its execution. With ALC, the AB is not used on all the transactions and the system can achieve a higher degree of scalability. As shown in the Figures 4.8(a) and 4.8(b), when the number of replicas is increased, the ALC protocol is able to maintain the throughput between 100 and 120 transactions per second and the abort rate is always around 5%. The CERT protocol generates also a bigger abort rate, that increases linearly until its saturation.

4.3.3 Lee Benchmark

Finally, this section shows results with another complex benchmark, namely Lee-TM. In this benchmark, multiple re-runs of a transaction have a non-negligible probability of accessing different data-sets, permitting to evaluate the performance of the ALC's deadlock avoidance



(a) Speed-up.



(b) Abort rate.

Figure 4.9: Lee benchmark running ALC.

mechanisms proposed in Section 4.2.3.

Figure 4.9(a) reports the speed-up achieved by ALC with respect to CERT computed considering the time required to route the whole set of junctions of the *mainboard circuit* (Ansari, Kotselidis, Watson, Kirkham, Luján, & Jarvis 2008) when using the two protocols. Also in this case, the performance gains achieved by ALC are clear, ranging from around 2x to more than 4x and growing along with the number of replicas in the system. Being the inter-transaction data locality of this benchmark pretty low (i.e. the likelihood to re-use a previously acquired leases when running two different transactions on a same replica was found to be less than 10%), the reason underlying the performance boost achievable by ALC is mainly imputable to its ability to reduce the transaction abort rate (see Figure 4.9(b)), and, in particular, to shelter long-running transactions from repeated aborts. Despite the lack of deterministic guarantees on the immutability of the data accessed during transactions re-runs, in fact, ALC guaranteed to execute transactions at-most once in the 98% of the cases. On the other hand, with CERT, long running transactions are very likely to be aborted tens of times before being successfully committed, causing a huge waste of computing resources.

4.4 Discussion

This chapter introduced ALC, a novel STM replication scheme that relies on the notion of asynchronous lease to boost the performance of existing AB-based transaction certification schemes. ALC was integrated in the framework that will be presented in Chapter 6, which allowed STM applications to transparently leverage the computational resources available in commodity clusters and shown the significant performance benefits achievable by ALC via a fully fledged prototype.

The ALC protocol is able to increase the performance of applications with very heterogeneous workloads, including systems that generate concurrent short and long running transactions. In ALC, the ownership of an asynchronous lease on a set of data items ensures the reduction of the commit phase latency, for transactions that access those data items, and shelter transactions from repeated abortions due to remote conflicts.

4.5 Summary

This chapter introduced the ALC protocol. It started by presenting the motivation for this protocol, followed by its architecture. The protocol is composed by two components, the Lease Manager and the Replication Manager. After presenting the basic protocol, several optimizations were proposed, to reduce the communication steps needed in the worst case scenario. The chapter finished with a performance evaluation of the ALC protocol.

Notes

The results presented in this chapter were accomplished in cooperation with Luís Rodrigues and Paolo Romano. The motivation for the ALC protocol was first presented as a fast abstract, with the title “Bridling Concurrency to Boost Performance In Distributed STMs”, in the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Chicago, USA, June 2010. The ALC protocol was proposed in the paper “Asynchronous Lease-based Replication of Software Transactional Memory”, Proceedings of the 11th International Middleware Conference, Bangalore, India, December 2010.

5 Speculative Certification

The replication protocol presented in this chapter, named SCert (Speculative Certification), aims at reducing the time to disseminate the updates generated by committing transactions in order to achieve the following two complementary goals:

- to provide executing transactions with fresher snapshots, thus reducing the probability of abort due to reads from stale data;
- to detect conflicts earlier during transaction execution, thus reducing the amount of wasted computation and useless waiting time caused by transactions doomed to abort.

This is achieved via a speculative approach, which leverages on the service provided by the Optimistic Atomic Broadcast (OAB) layer, formalized in the Section 3.1. OAB allows to propagate the post-images of committing transactions well before their final serialization order is defined. In addition to the final, total delivery order notification, which is available only after several communication steps (typically at least three (Guerraoui & Rodrigues 2006)), an OAB service also provides an earlier guess of the final total order. This guess, called optimistic delivery order, normally corresponds with the spontaneous network delivery order and can therefore be made available after a single communication step. Also, as discussed in (Pedone & Schiper 2003) and confirmed by the experimental study, the probability of mismatch between optimistic and final delivery order is typically fairly low in LANs ($< 15\%$).

SCert takes advantage of this property in a twofold way. First, it propagates the updates in a speculative serialization order that corresponds to the sequence of optimistically delivered messages. Second, it allows speculatively activated transactions to further propagate the snapshots they generate across chains of speculative transactions. This provides an effective pipelining of speculative transactions that allows to maximize the gains achievable via speculation. On the other hand, speculation exposes SCert to risks of cascading abort in case of mismatches between the optimistic and final delivery order of two conflicting transactions. As it will be demonstrated in the experimental study, this represents an advantageous trade-off. In all the tested scenarios, including those generating higher loads (and consequently mismatches between optimistic and final delivery orders) the performance penalty associated with the occurrence of mismatches between final and optimistic delivery orders is compensated by the benefits achievable by the aggressive propagation of speculative snapshots.

It should be highlighted that the benefits of speculatively propagating the snapshots are higher in the context of STMs than in conventional databases. In fact, unlike classical database systems, STMs incur neither in disk access latencies nor in the overheads of SQL statement parsing and plan optimization. This makes the execution time of typical STM transactions two or three orders of magnitude shorter than in database settings, as shown in Section 3.4. Since the ratio between coordination times and transaction processing times is higher in STMs, there are also more opportunities to obtain performance gains from optimistic schemes that shorten the coordination phase.

In order to evaluate the actual speed-ups offered by SCert, a prototype was developed based on JVSTM (Cachopo & Rito-Silva 2006) and the APPIA Group Communication System (Miranda, Pinto, & Rodrigues 2001). While the SCert scheme could be in principle coupled with STMs that employ different concurrency control policies, the choice to integrate SCert with JVSTM is motivated by a twofold reason. First, the multi-versioning concurrency control mechanism adopted by JVSTM allows maximizing the performance of read-only transactions, preventing them from aborting or ever blocking due to conflicts with write transactions. Further, since JVSTM already maintains and manages multiple data item versions, it lends itself naturally to be extended to support the additional, speculative data item versions exploited by SCert. The experimental evaluation shows that SCert achieves speed-ups of up to 4.5x when compared with competing replicated STMs (Couceiro, Romano, Carvalho, & Rodrigues 2009).

The remainder of this chapter is structured as follows. Section 5.1 describes the architecture. Section 5.2 introduces SCert and discusses the issues associated with its integration with JVSTM. Section 5.3 presents the results of an experimental study. Finally, Sections 5.4 and 5.5 conclude and summarize this chapter.

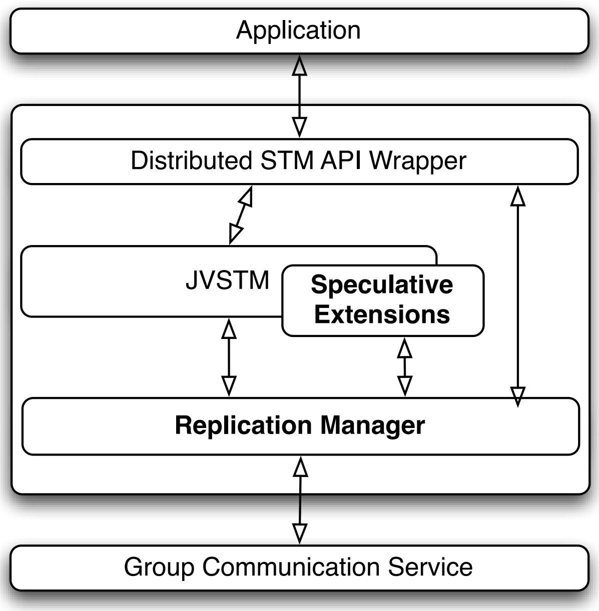


Figure 5.1: Architecture of a SCert replica.

5.1 The SCert Architecture

The architecture of the software deployed on each replica is illustrated in Figure 5.1. The top layer is a wrapper that intercepts the application level calls for transaction demarcation (i.e. to begin, commit or abort transactions), not interfering with the application (read/write) access to the transactional data items, which are managed directly by the underlying STM layer. This approach allows for transparently extending the classic STM programming model to a distributed setting.

The mechanisms for maintaining and managing speculative data item versions are provided by the two core components of the SCert protocol: the STM's Speculative Extensions (SE) and the Replication Manager (RM). The Speculative Extensions were implemented for a multi-versioned STM, namely JVSTM (Cachopo & Rito-Silva 2006). JVSTM maximizes the performance of read-only transactions and, since it already embodies mechanisms to maintain multiple copies of the same data, it lends itself naturally to support the additional speculative data item versions required by SCert. These mechanisms are detailed in Section 5.2.2. The RM is responsible of coordinating the commit phase, implementing the speculative certification scheme by leveraging on the services provided by the STM, the SE and the Group Communication Service (GCS) (Chockler, Keidar, & Vitenberg 2001), which is the bottom layer. The GCS provides the view synchronous OAB services. All the experiments described in this chapter have been performed using the Appia GCS (Miranda, Pinto, & Rodrigues 2001).

5.2 The SCert Protocol

Before delving in the detailed description of the SCert protocol, a brief and informal overview of its key mechanisms is provided. The considered properties and system model are described in the Section 3.1. As in conventional certification protocols, e.g. (Couceiro, Romano, Carvalho, & Rodrigues 2009), in SCert transactions are run locally, without incurring in any replica coordination during their execution. Once a transaction reaches its commit phase, it is first locally validated and then its read-set and write-set are disseminated to all replicas by means of the OAB service. Unlike conventional certification protocols, however, SCert does not wait until the final delivery order of the atomic broadcast is known to certify the transaction. Instead, SCert speculatively certifies a transaction as soon as the broadcast is optimistically delivered. If the validation succeeds, the transaction is *speculatively committed.*

Note that the application call to commit a transaction does not return if the transaction is only *speculatively committed.* Therefore, user-level code is not affected by mispeculations that may result from a mismatch between the optimistic and final delivery orders. Still, the post-images (i.e. the values of the write-set) of a speculatively committed transaction are applied (added) to the STM and marked as *speculative.* A speculatively committed transaction will eventually be *finally committed*, its updates marked as *committed* and the user-level code allowed to return from the invocation of the commit method. Speculative values only become committed values if there is no mismatch between the optimistic and the final order of the OAB or, when a mismatch occurs, if the transaction can be safely re-ordered. Roughly speaking, the latter case corresponds to scenarios in which the transaction did not develop any read-from dependency from transactions that were speculatively committed in a serialization order not conciliable with the final delivery order. Speculatively committed versions of data items (simply named speculative versions) are immediately made available to new transactions. Therefore, new transactions are tentatively serialized after the last speculatively committed transaction, thus improving their chances to observe a non-stale snapshot. In the following, transactions that are activated while the local STM maintains speculative versions will be denoted as *speculative transactions.*

In SCert, speculative transactions that enter their commit phase can also atomically broadcast, in their turn, a certification request. Upon the optimistic delivery of a speculative transaction T, T is validated to detect conflicts not only against committed transactions, but also against speculatively committed transactions that were optimistically delivered before T. This allows to generate a chain of speculatively committed transactions that are serialized in an order compliant with the sequence of optimistic deliveries. In other words, during the time window that starts with the optimistic delivery of a transaction T and ending with its final delivery, SCert strives to serialize any concurrently executing $T^1, \ldots, T^n$ according to their optimistic delivery order, achieving an overlap between communication and processing that is not possible with a conventional (non-speculative) certification scheme.

Furthermore, SCert also exploits speculative versions to implement early conflict detection. As soon as a transaction T is speculatively committed, any other local transaction that (*i*) was serialized before T, and that (*ii*) has read, or reads, a data item updated by T is immediately aborted. Whenever the optimistic order matches the final delivery order, this early abort mechanism prevents the waste of time/computational resources with respect to conventional certification schemes, where conflicts are only detected upon the final AB delivery.

The remainder of this section is structured as follows. Section 5.2.1, starts by providing an overview of the key mechanisms of JVSTM, followed by a discussion, in Section 5.2.2, on how JVSTM has been extended to maintain and manage speculative versions. Next, Section 5.2.3, describes how the Replication Manager orchestrates the execution of transactions across the distributed STM platform. Section 5.2.4 highlights the performance benefits of SCert, by illustrating some of its execution sketches. Finally, Section 5.2.5 provides some informal arguments on its correctness.

5.2.1 Overview of JVSTM's internals

JVSTM implements a multi-version scheme which is based on the abstraction of a *versioned box* (VBox). A VBox is a container that keeps a tagged sequence of values - the history of the versioned box. Each of the history's values corresponds to a change made to the box by a successfully committed transaction and is tagged with the timestamp of the corresponding transaction. The versions of VBox are arranged into a linked list, whose head maintains the version created by the last transaction that committed (and issued a write on the VBox).

To keep track of the serialization order of transactions, JVSTM maintains a global integer timestamp, *commitTimestamp*, which is incremented whenever a transaction commits. Each transaction stores its timestamp in a local *snapshotID* variable, which is initialized at the time of the transaction activation with the current value of *commitTimestamp*. This information is used both during transaction execution, to identify the appropriate values to be read from the VBoxes, and, at commit time, during the validation phase, to determine the set of concurrent transactions to check against possible conflicts.

More in detail, when a transaction T having *snapshotID=s* issues a read operation on a VBox X, JVSTM returns the version stored in X associated with the largest timestamp smaller or equal to s. In other words, it returns the version created by the last transaction that (i) has issued a write on X and (ii) was serialized before T. For what concerns write operations, JVSTM stores the values written by a transaction in a private buffer, and applies them to the corresponding VBoxes only at commit time, provided that the transaction passes a validation phase.

The validation is performed by checking whether any of the VBoxes read by a transaction T has been updated by some committed transaction T' with a larger timestamp. In this case T is aborted. Otherwise, T is committed by atomically executing (within a critical section) the following operation. The *commitTimestamp* variable is increased, and the transaction's *snapshotID* is set to the new value of *commitTimestamp*. Finally the new values of all the VBoxes written by the transaction are appended to the linked list of versions tagged with the current value of *commitTimestamp*. As a final note, JVSTM integrates a garbage collection mechanism that detects if there are versions stored within some VBox that are no longer visible by any currently active transaction. The interested reader may refer to the work of Cachopo & Rito-Silva (2006) for a detailed description of this mechanism.

5.2.2 JVSTM Extensions for Speculative Transactions

In order to maintain and manage speculative versions, the following extensions have been integrated in JVSTM. In addition to *commitTimestamp*, JVSTM now maintains an additional global timestamp called *speculativeTimestamp* that is incremented whenever a transaction is speculatively committed. Note that since a transaction is only committed after it is final delivered, and given that a final delivery for a message is always preceded by its optimistic delivery,

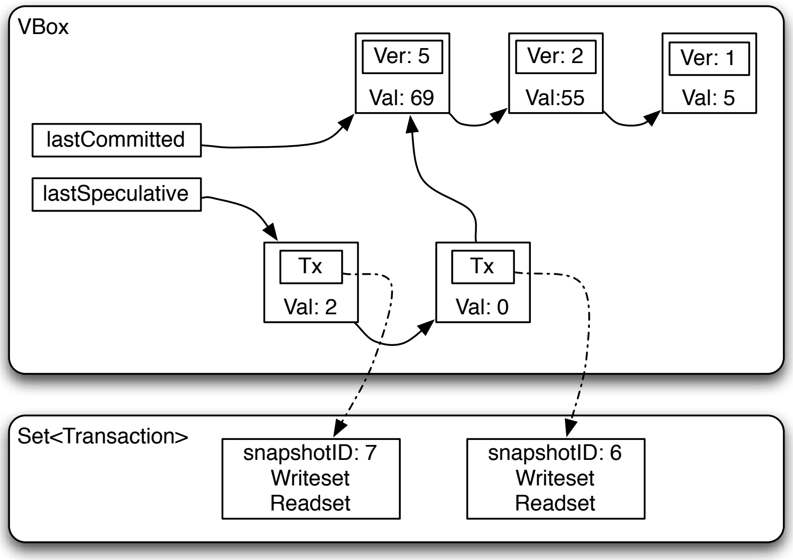


Figure 5.2: VBox with speculative versions.

it follows that *speculativeTimestamp* ≥ *commitTimestamp*. Also, *speculativeTimestamp* = *commitTimestamp* only if currently there are no speculatively committed transactions (and consequently speculative data items' versions). Whenever a transaction is activated, its *snapshotID* variable is assigned the current value of *speculativeTimestamp*, thus serializing it after the last speculatively committed transaction.

To distinguish between speculative and non-speculative data item versions, the original JVSTM VBox data structure was extended, as shown in Figure 5.2. The VBox stores both committed and speculative versions into a single linked list, maintaining one reference to the most recent committed version (*lastCommitted*) and one to the most recent speculative version (*lastSpeculative*). Unlike final committed versions, versions created by speculatively committed transactions are not associated with a version timestamp; instead, they store a reference to a data structure that keeps the *snapshotID*, the read-set, and the write-set of the (local or remote) transaction that created them. As it will be seen in Section 5.2.3, this indirection mechanism allows to manage efficiently the case in which, due to mismatches between optimistic and final delivery orders, speculative transactions need to be final committed in an order different from that in which they had been originally speculatively committed. A key advantage of the presented design lies in its non-intrusiveness. It in fact allows to reuse, with minimal changes, the JVSTM's original mechanisms for garbage collecting obsolete versions, determining versions' visibility during read operations, and validating transactions.

These additional data structures are used to implement early conflict detection. In JVSTM, if during a read operation a transaction T realizes that a version tagged with a timestamp larger than its own *snapshotID* is already stored within the VBox, T is not aborted right away. Conversely, T navigates the linked list of versions to retrieve the version generated by the last transaction that committed before T started. This prevents read-only transactions from aborting (by serializing them in the past). Unfortunately, for update transactions, even though at this stage T is already doomed to abort, JVSTM will only abort it during the validation phase taking place during T's commit phase. In the speculative version of JVSTM, this suboptimal behavior was changed to immediately abort an update transaction T that reads a VBox for which a (committed or speculatively committed) version exists with a timestamp larger than T's *snapshotID*.

Furthermore, in order to allow the RM to orchestrate the SCert replication protocol, the JVSTM API has been extended with the following primitives:

SPECCOMMIT(**Transaction T**) This method speculatively commits a transaction T. To this end, it increases *speculativeTimestamp*, assigns the new value to T's *snapshotID*, and updates the VBoxes of all items in T's write set. The VBoxes are updated as follows: (*i*) a new speculative version is added to the head of the versions' list tagged with the current value of *speculativeTimestamp*, and (*ii*) the *lastSpeculative* pointer is set to this new version.

SPECABORT(**Transaction T**) This method is used to abort a previously speculatively committed transaction. To this end, it eliminates the corresponding speculative versions of data items in T's write-set from their VBoxes (updating the *lastSpeculative* pointer accordingly).

SPECVALIDATE(**Transaction T**) This method validates T by iterating over its read-set and returning true only if T has read the most recent *speculatively or finally* committed version of each data item.

VALIDATE(**Transaction T**) This method validates T by iterating over its read-set and returning true only if T has read the most recent *finally* committed version of each data item.

COMMIT(**Transaction T**) This method finalizes the commit of a transaction T that is currently the oldest of the speculatively committed transactions. To this end, it increases the *commitTimestamp* and updates the VBoxes of all items in T's write-set. The VBoxes are updated as follows: (*i*) the speculative version previously created by T is replaced by a non-speculative version tagged with the current value of *commitTimestamp*, and (*ii*) the *lastCommitted* pointer is set to this new version.

ABORT(**Transaction T**) This method aborts the transaction T. Since the transaction will not be applied to memory, all the data of this transaction is discarded, including the read-set and the write-set.

SPECOUTOFORDERCOMMIT(**Transaction T**) This method is used to speculatively commit a transaction without adding its write-set to the head of the linked list of versions. To this end, it first increases *speculativeTimestamp* and assigns it to T's *snapshotID*. Next, for each data item in T's write-set, it inserts into the corresponding VBoxes a speculative version after the speculative version created by the transaction with the largest not null *snapshotID*. If no such transaction exists, T's version is inserted after the last committed version. Finally, the *lastSpeculative* pointer is set to refer to the version created by T.

OUTOFORDERCOMMIT(**Transaction T**) This method is used to commit a transaction T that is not currently the oldest of the speculatively committed transaction. This is possible either because T was not previously speculatively committed (upon its optimistic delivery), or because it was speculatively committed in a different order. In both cases the *commitTimestamp* is increased and the *lastCommitted* pointer is set to refer to a new non-speculative data item version that is inserted between the last finally committed version and the first speculatively committed version (if any). If T had previously been speculatively committed, however, any speculative version it had previously stored in JVSTM is also erased.

5.2.3 Replication Manager

The pseudo-code describing the behavior of the RM is shown in Algorithm 6, Algorithm 7 and Algorithm 8. As outlined before, transactions execute in a single machine, accessing the most recent speculatively committed snapshot available at the time they were activated. The RM is activated whenever a local transaction requests to commit. At this point, the transaction

Algorithm 6: SCert Replication Manager (Part I).

```
FIFOQueue<Transaction> optDel = $\emptyset$
Set<Transaction> specComm = $\emptyset$
Set<Transaction> specAborted = $\emptyset$

void commit (Transaction T)
  if ( $\neg$ JVSTM.specValidate (T) ) then
    JVSTM.abort (T)
  else
    trigger OA-broadcast [T]
    wait until ( isTransactionFinished (T) $\vee$ ejected )
    if ( ejected ) then
      JVSTM.abort (T)

boolean isTransactionFinished (Transaction T)
  return ( JVSTM.isAborted (T) $\vee$ JVSTM.isCommitted (T) )

upon event Opt-deliver ([Transaction T]) atomically do
  optDel.add (T)
  if ( $\neg$JVSTM.validate (T) ) then
    JVSTM.abort (T)
  else
    if ( $\neg$JVSTM.specValidate (T) ) then
      specAborted.add (T)
    else
      specComm.add (T)
      JVSTM.specCommit (T)

upon event TO-deliver ([Transaction T]) atomically do
  if ( JVSTM.isFinalAborted (T) ) then
    optDel.remove (T)
  else
    if ( optDel.getFirst () $\neq$ T ) then
      handleOutOfOrder (T)
    else
      optDel.removeFirst ()
      if ( specAborted.contains (T) ) then
        specAborted.remove (T)
        JVSTM.abort (T)
      else
        specComm.remove (T)
        JVSTM.commit (T)
```

undergoes first a local validation. Conflicts with concurrent transactions that have already locally (speculatively or finally) committed are detected at this stage. If this validation fails, the transaction is immediately aborted. Otherwise, its read-set, write-set, and *snapshotID* are sent to all replicas using the OA-broadcast primitive (described in the Section 3.1). At this point, the user call becomes blocked until the transaction outcome is defined.

A transaction is received by all nodes twice. The first time, it is received by the Opt-deliver primitive, which provides an early estimate of the final delivery order. As already discussed, SCert leverages on the observation that in a local network, the spontaneous order of delivery of the messages from the network coincides, with high probability (Kemme, Pedone, Alonso, &

Algorithm 7: SCert Replication Manager (Part II).

```
void handleOutOfOrder (Transaction T)
 optDel.remove (T)
 boolean outcome = JVSTM.validate (T)
 if  ( $\neg$ outcome $\wedge$ specAborted.contains (T)) then
   // avoid revalidate other txs
   specAborted.remove (T)
   JVSTM.abort (T)
 else
   temporarily block activation of new transactions
   abort local transactions not yet in their commit phase
   if ( $\neg$ outcome ) then
      specComm.remove (T)
      JVSTM.abort (T)
   else // tx out of order, but still committable
      if ( specAborted.contains (T) ) then specAborted.remove (T)
      if ( specComm.contains (T) ) then specComm.remove (T)
      JVSTM.outOfOrderCommit (T)
   revalidateOptDelTxs ()
   unblock activation of new transactions
```

Schiper 1999), with the final total delivery eventually determined by the OAB service.

Optimistic Delivery When the transaction is optimistically delivered, it is validated to detect possible conflicts with the transactions that committed so far, either finally or speculatively. This phase is called *speculative validation.* If it successfully passes this phase, the transaction is speculatively committed and the transaction is appended to the *specComm* set. Otherwise, the transaction is added to the *specAborted* set. Note that at this stage the transaction is not aborted yet. The transaction may in fact be still committed if, upon its final delivery, a mismatch between the optimistic and final delivery orders is detected, and if the serialization order determined by the final delivery order results to be equivalent to the one in which the transaction was originally processed. In both cases, the transaction is added to the *optDel* queue. This queue will be later used to detect possible mismatches between the optimistic and final delivery orders.

Final Delivery of Aborted Transactions Upon TO-delivery of a transaction T, it is first checked (via the isFinalAborted method) if the transaction has already been aborted. This can happen in case T had observed the speculative snapshot generated by a transaction T' that was later on aborted, generating the cascading abort of T. In this case, T is simply removed from the *optDel* queue.

Algorithm 8: SCert Replication Manager (Part III).

```
void revalidateOptDelTxs ()
 JVSTM.lastSpeculativeTimestamp =
   JVSTM.lastCommittedTimestamp
 foreach Transaction $T \in optDel \wedge \neg$ JVSTM.isFinalAborted (T) do
   // reset snapshotIDs before re-assigning them
   T.snapshotID = null
 foreach Transaction $T \in optDel \wedge \neg$ JVSTM.isFinalAborted (T) do
   if ( ¬JVSTM.validate (T) ) then
     JVSTM.abort(T)
   else
     if ( ¬JVSTM.specValidate (T) ) then
       if ( specComm.contains (T) ) then
         // Tx prev. speculatively committed
         specComm.remove (T)
         specAborted.add (T)
         specAbort (T)
     else // Tx passed speculative validation
       if ( specAborted.contains (T) ) then
         // Tx prev. speculatively aborted
         specAborted.remove (T)
         specComm.add (T)
         JVSTM.specOutOfOrderCommit (T)
       else // Tx already spec. committed, update its snapshotID
         T.snapshotID = ++JVSTM.lastSpeculativeTimestamp
```

Final Delivery with "Matching-Order" If the outcome of T has still to be determined, it is checked whether T is at the head of the *optDel* queue. If it is true, this means that the final delivery order matches the optimistic delivery order. In this case, the transaction's outcome (abort or commit) can be easily determined by checking whether the transaction has been placed in the *specComm* set or in the *specAborted* set. If the transaction had executed locally and was speculatively aborted, the local instance of JVSTM is notified. This last step is not necessary if the transaction has been executed remotely, as the local instance of JVSTM has no knowledge of the transaction. If the transaction is committed, the commit() method is called to update the VBoxes as detailed in Section 5.2.2.

Final Delivery with "Mismatching-Order" On the other hand, T is not at the head of the *optDel* queue, then a mismatch between the optimistically delivery and final delivery has occurred. Naturally, this is the most complex scenario that has to be managed by SCert. The pseudo-code for this case is depicted in the handleOutOfOrder() method (see Algorithm 7).

After removing T from the *optDel* queue, T is validated to detect whether, despite the misalignment between the optimistic and final delivery orders, it can still be serialized immediately after the last finally committed transaction. If this validation fails and T had not been previously speculatively committed, T can be aborted right away, since no other transaction may have ever observed its snapshot.

Additional care is needed in the following two cases:

- T had previously been speculatively committed, but it needs to abort. In this case, in fact, T's snapshot may have already been observed by other transactions, that may possibly be still executing (i.e. not yet in their commit phase).
- T may be (finally) committed. In this case, either T had been previously speculatively aborted, or had been speculatively committed in a different serialization order. Either way, this can impact both the speculative decision (commit/abort) already taken for the remaining optimistically delivered transactions, and the snapshots observed by currently executing transactions.

In order to avoid currently executing transactions from accessing inconsistent snapshots and suffering of anomalies due to the loss of opacity (Guerraoui & Kapalka 2008b), the required readjustments of the speculative snapshots are done only after having blocked new transactions from starting and after having aborted any ongoing transaction. Also, only after having concluded readjusting the speculative snapshots, the activation of new transactions will be allowed again.

The snapshot realignment consists of the following steps. First, the outcome of transaction T is finalized either via the ABORT() or the OUTOFORDERCOMMIT(), depending on the output of its validation phase. At this point, in the REVALIDATEOPTDELTXS() method (shown in Algorithm 8), the remaining optimistically delivered transactions are revalidated to take into account the unexpected order in which T was committed. This also includes reassigning the *snapshotID* timestamps to every transaction which were found to be speculatively committable. To achieve this result, SCert starts by setting the *lastSpeculativeTimestamp* to *lastCommitTimestamp* and resetting the *snapshotID*s of all the optimistically delivered transactions. This has the effect of resetting the STM to the state it had before having speculatively committed any of the optimistically delivered transactions. Next, SCert iterates over these transactions following their (updated) order of optimistic delivery. Each of them is first validated against the already committed transactions and, in case the first validation succeeds, against those that have already been speculatively committed. If the speculative validation fails, the transaction is simply speculatively aborted. If it succeeds, however, it is checked whether the transaction had previously

Algorithm 9: SCert Replication Manager at process p_i: dealing with view changes.

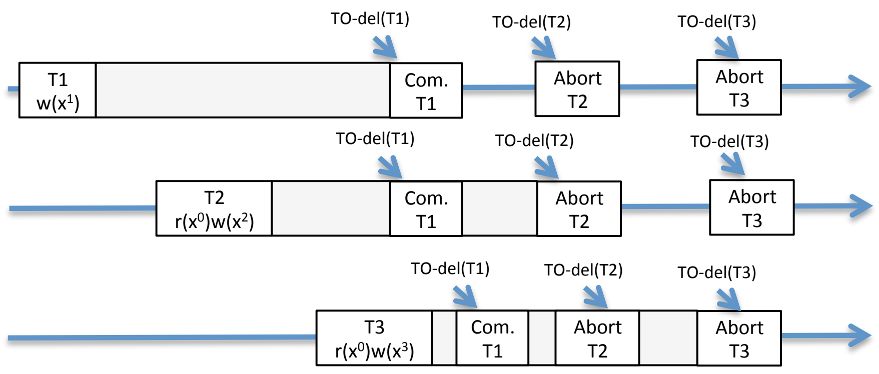
```
View currentView=$\{p_1,\ldots,p_i,\ldots,p_n\}$
boolean inPrimaryComponent=*true*

**upon event** *ViewChange(View newView)* **do**
 **if** ($\neg$inPrimaryComponent $\vee$ $p_i$ is joining for the first time) **then**
   perform state transfer
   inPrimaryComponent=*true*
 **else**
   $\forall p_j$ s.t. ($p_j \in$ currentView $\wedge$ $p_j \notin$ newView) **do**
     $\forall$ T $\in$ specComm s.t. T.proc = $p_j$ **do**
       specComm.remove (T)
     $\forall$ T $\in$ specAborted s.t. T.proc = $p_j$ **do**
       specAborted.remove (T)
     $\forall$ T $\in$ optDel s.t. T.proc = $p_j$ **do**
       optDel.remove (T)
 currentView = newView

**upon event** *ejected* **do**
 inPrimaryComponent=*false*
```

been speculatively committed or aborted. In the former case, it means that its snapshot is already present in memory. Thus it suffices to increase the *lastSpeculative* timestamp and assign its updated value to the transaction's *snapshotID.* If the transaction was previously speculatively aborted, instead, its write-set must be applied, in the right order, in the linked list of versions maintained by the corresponding VBoxes. This is done using the SPECOUTOFORDERCOMMIT() primitive (see Section 5.2.2).

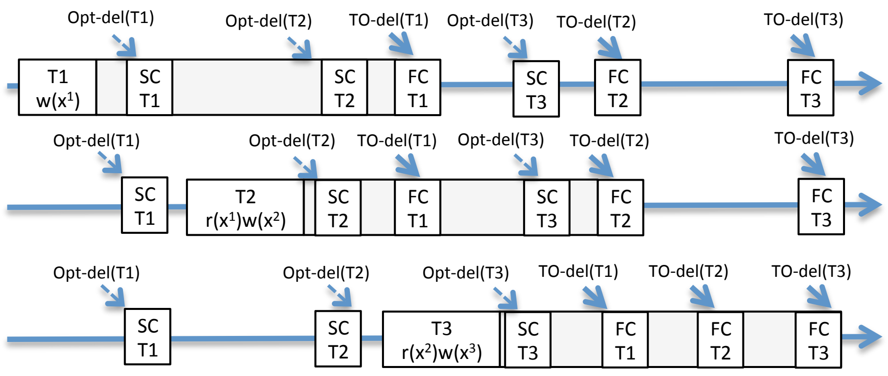
Dynamic Membership It remains to discuss the replicas' behavior in the presence of view changes and ejections from the primary component view, which is shown in the pseudo-code in Algorithm 9. Upon delivery of a new view event, if the replica re-joins the primary component or is joining the group of replicas for the first time, it triggers a state transfer procedure that realigns the content of the local replica of the STM, as well as of the state variables of the replication protocol. The state transfer procedure is a complex task that can be solved using several existing mechanisms (e.g. (Jiménez-Peris, Patiño Martínez, & Alonso 2002)).

5.2.4 Example execution sketches

To clarify the dynamics of the SCert protocol, this section illustrates two sketches of its execution, contrasting it with analogous execution sketches for a conventional certification protocol, such as Couceiro, Romano, Carvalho, & Rodrigues (2009) and Pedone, Guerraoui, &



(a) Conventional certification protocol.



(b) SCert.

Figure 5.3: SCert execution sketch: cascading commits.

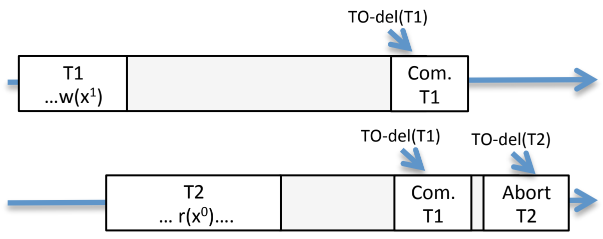
Schiper (2003). In the Figures 5.3 and 5.4, SC and FC stand for, respectively, *Speculative Commit* and *Final Commit*.

On the diagram illustrated in the Figure 5.3, it is shown the execution of three conflicting concurrent transactions, T1, T2 and T3, which all issue a read and write operation on a data item X. Let us assume that the transactions are executed on different replicas, even though the same considerations drawn in the following would apply in case the transactions were all executing in the same machine. Note that the execution times of transactions and atomic broadcast are not in scale as, in typical STM applications, the average transaction execution time is normally several orders of magnitude smaller than the completion time of atomic broadcast.

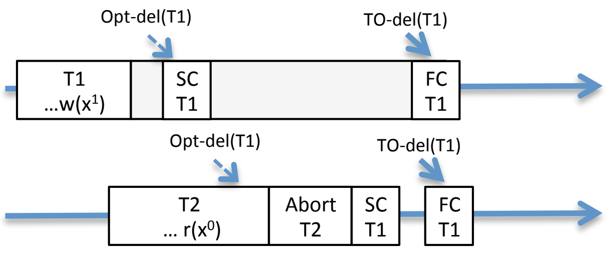
In non speculative certification schemes, the post-images of the data updated by transactions are propagated only after the corresponding message is final delivered. As the level of concurrency among transactions grows, the chances that transactions miss the snapshots generated by previously completed transactions increase significantly, leading to a corresponding increase of the abort rate. Due to this, in the example reported in Figure 5.3(a), both transactions T2 and T3 would need to abort, as both have read an obsolete version of X. In SCert, conversely, transactions T2 and T3 can benefit from the early propagation (via optimistic delivery) of speculatively committed snapshots and can be successfully committed if, as considered in this example (see Figure 5.3(b)), there is no mismatch between optimistic and final delivery orders. Note that, in this example, the speculative propagation of snapshots takes place through a chain of transactions, as T2 reads the version of X generated by T1, and T3 observes the version of X written by T2. This brings two main benefits: (*i*) it reduces the abort rate of concurrent transactions by exposing fresh data to the system sooner and (*ii*) it allows overlapping the processing of transactions with the commit process.

The execution sketch shown in the Figure 5.4 illustrates the benefits deriving from the early abort notification scheme provided by SCert. Even in scenarios where it is not possible to propagate the snapshots of a concurrent transaction in time, as in the case of T2 that has already issued a read operation on X before T1 is optimistically delivered, SCert exploits speculation to abort immediately transactions that will certainly abort once that they will be final delivered in absence of mismatches between the optimistic and final delivery orders.

Clearly, the effectiveness of SCert depends significantly on the probability that the optimistic order matches the final (total) order and, consequently, it results particularly attractive in Local



(a) Conventional certification protocol.



(b) SCert.

Figure 5.4: SCert execution sketch: early aborts.

Area Networks, where the probability that the network spontaneous order will match the final order is high. When considering, for instance, the scenario on the left side of Figure 5.3, had the final delivery order been {T3,T2,T1}, SCert would have induced the abort of T3 and (the cascading abort of) T2, committing only T1. It is noteworthy to highlight that this worst case scenario for SCert would not have been outperformed by a conventional certification protocol, since a non-speculative protocol would also have committed only one transaction (namely T3).

5.2.5 Correctness arguments

Our target consistency criterion for replication is 1-copy serializability (Bernstein, Hadzilacos, & Goodman 1987), which ensures that the execution history of committed transactions across the whole set of replicas is equivalent to a serial transaction execution history on a not replicated STM. In SCert a transaction returns from an application's commit request only if the OAB service has established its final delivery order for which group-wide consensus is ensured. Further, a replica final commits a transaction T only if it passes a deterministic validation phase that ensures that T has been serialized in an order compliant with the OAB's final delivery order. To this end, SCert performs a first speculative validation upon the optimistic delivery of transactions. At this stage, however, no irreversible decision on the transaction's outcome is taken, or is externalized to user level applications. This only occurs upon final delivery of transactions. If at this point, it is found out that the optimistic and final delivery orders coincided, SCert avoids re-validating the transaction (as this would yield the same result of the speculative validation), and simply confirms the outcome of the speculative validation, final committing or aborting the transaction. If, on the other hand, upon the final delivery of transaction T a replica detects that the optimistic delivery order has been contradicted by the final delivery order, a corrective action is taken which re-validates both T and every optimistically (but not yet finally) delivered transaction. This ensures that the final decision taken on T's outcome is identical at each replica. Also, it guarantees that the outcome of the speculative validation for optimistically delivered transactions is consistent with the updated optimistic and final delivery orders. This allows to safely avoid further validations in the future, if optimistic and final delivery orders were to no longer diverge.

SCert preserves the strong atomicity (Martin, Blundell, & Lewis 2006) and opacity (Guerraoui & Kapalka 2008b) properties. Strong atomicity is ensured by JVSTM at the language level, via the VBox abstraction, which prevents the possibility for any non-transactional manipulation of its state. Layering on top of JVSTM, and sharing the reliance on the VBox abstraction, SCert simply inherits this property. Opacity (Guerraoui & Kapalka 2008b), on the other hand, can be informally viewed as an extension of the classical database serializability property with the additional requirement that also non-committed transactions are prevented from observing inconsistent states, namely snapshots that could not be generated in any sequential transaction execution history.

Opacity in SCert is guaranteed since every transaction is forced, *since its start*, to observe a consistent snapshot obtained by sequentially executing transactions according to the order defined by the delivery of messages of the OAB service. More in detail, SCert forces the underlying STM to serialize all transactions according to the optimistic delivery order. The validation phase performed as transactions are optimistically delivered, in fact, allows speculatively committing a transaction only if it can be serialized after the last speculatively committed transaction.

In absence of mismatches between the optimistic and final delivery orders, this serialization order is then simply confirmed as the OAB establishes the final delivery order. Were the two message delivery orders differ, SCert atomically aborts any transaction that was speculatively

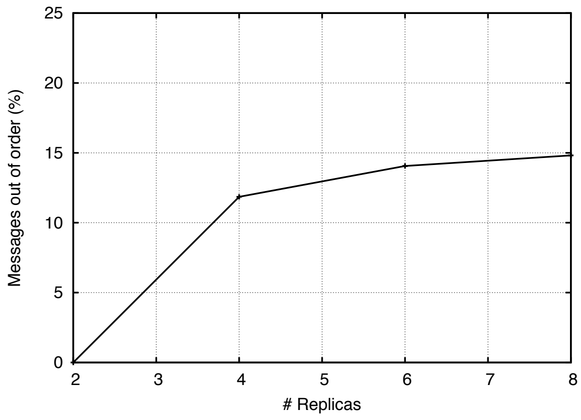


Figure 5.5: Mismatches between optimistic and final message deliveries.

serialized in an order that is not conciliable with the final delivery order and reassigns timestamps to the remaining speculatively transactions. Before doing this, however, SCert aborts every ongoing transaction and prevents new transactions to start until this reconciliation phase is completed. This clearly rules out the possibility that ongoing transactions can observe any inconsistent state during this phase.

5.3 Performance Evaluation

This section reports results from an experimental study aimed at quantifying the performance gains achievable by SCert when compared with non-speculative certification based protocols. To this purpose, the baseline protocol used was D^{2}STM (Couceiro, Romano, Carvalho, & Rodrigues 2009), described in the Section 3.4.1. The remaining of the text refers to this protocol solely as "CERT".

The testbed platform consists of a cluster of 8 nodes, each one equipped with two Intel Quad-Core XEON at 2.0 GHz, 8 GB of RAM, running Linux 2.6.32-26-server and interconnected via a private Gigabit Ethernet. The results shown in this section are an average of 10 runs. The benchmarks used on the performance evaluation are described in the Section 3.4.2.

5.3.1 Bank Benchmark

The first scenarios to be considered use synthetic workload, obtained by the Bank Benchmark. It is a simple benchmark that has the advantage of providing a fine control on the conflict

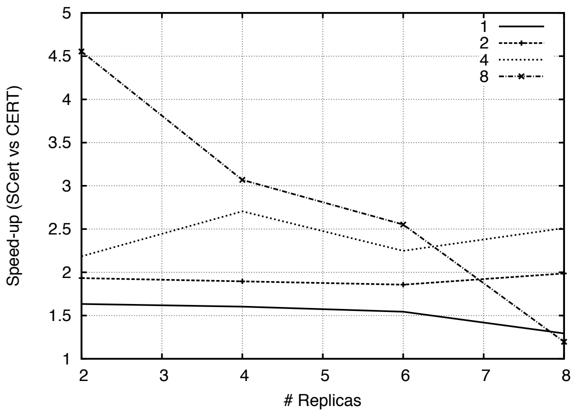


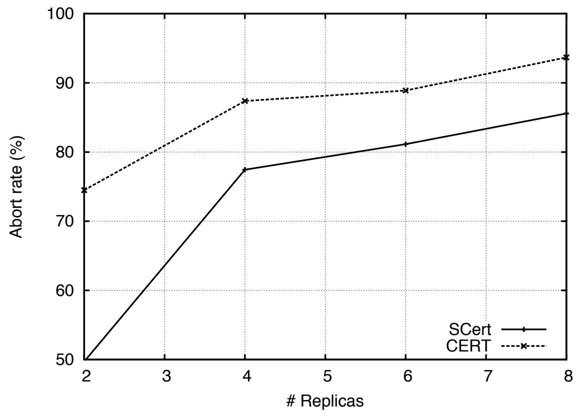
Figure 5.6: Bank benchmark running SCert: speed-up.

rate. The benchmark was initialized such that all nodes replicate an array of accounts of size *numMachines*·*numThreads*·2 items. Depending on the accounts accessed by each transaction, the benchmark can generate from 0% to 100% conflicts among concurrent transactions.

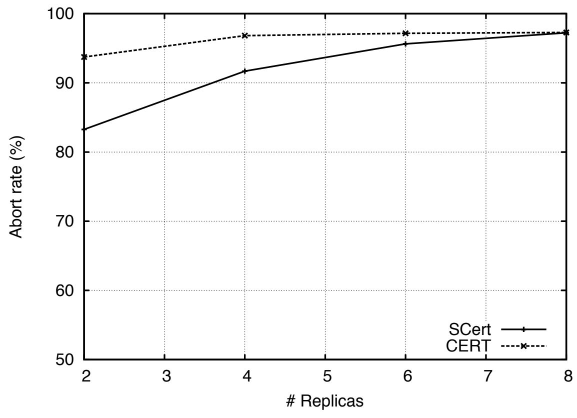
The experiments were done to evaluate the performance of SCert in a scenario where all transactions touch the same accounts, i.e., with 100% conflicts. The number of replicas varied between 2 and 8, and the number of threads on each replica was fixed. Depending on the test, the system was configured to run with a number of threads between 1 and 8. Since all nodes are continuously processing very small transactions and sending OAB messages, this quickly saturates the group communication service and generates a significant amount of contention in the network.

Figure 5.5 shows the number of messages that were delivered out of order by the Opt-delivery primitive when 8 threads are used. As it is shown in the figure, even in a high network contention scenario the number of messages optimistically delivered out of order never goes over 15%.

Figures 5.6 and 5.7 report the speed-ups achieved by SCert with regard to CERT, as well as the observed abort-rate for both SCert and CERT. It can be observed that SCert is able to improve the system performance up to a 4.5x factor (see Figure 5.6) when compared with CERT, even for small reductions in the abort rate. SCert provides the best results when the conflict rate is high but the network is not saturated (this is achieved by using 8 threads on just 2 replicas). When the network load increases (more replicas are used), the performance advantages of SCert decrease but, in most cases, SCert is still able to reduce significantly the

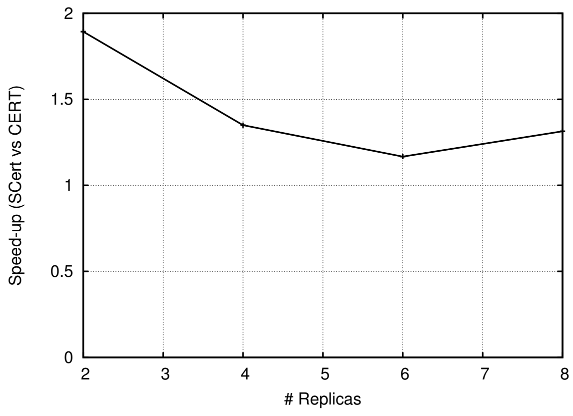


(a) Abort rate (1 Thread).

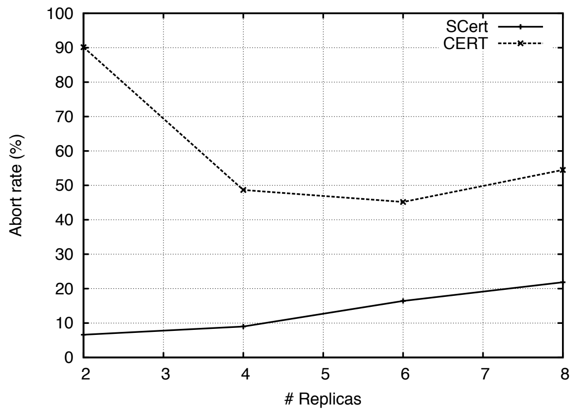


(b) Abort rate (8 Threads).

Figure 5.7: Bank benchmark running SCert: abort rate.



(a) Speed-up.



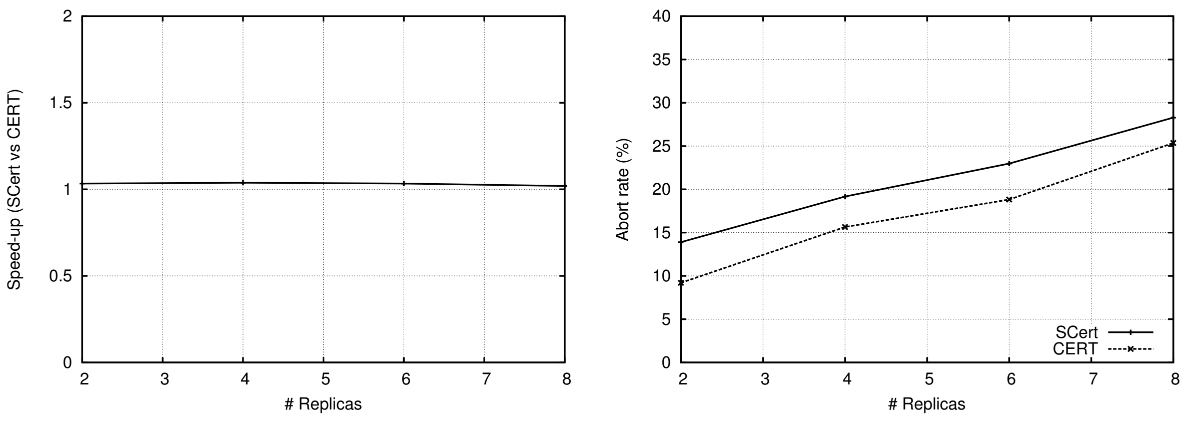
(b) Abort rate.

Figure 5.8: STMBench7 running SCert.

abort rate in the system (as shown in the Figure 5.7). In the scenario where 8 threads are running in each of the 8 replicas, the degree of concurrency and the contention in the network is so high that both protocols end up delivering similar performance.

5.3.2 STMBench7

This section shows results using STMBench7. Figure 5.8 depicts the performance of both protocols using the "write dominated" workload without long running transactions. As before, each plot shows the speed-up of SCert over CERT and the abort rate of both protocols (SCert and CERT). The number of replicas varies between 2 and 8 and the number of threads was fixed to 2.



(a) Speed-up. (b) Abort rate.

Figure 5.9: Lee benchmark running SCert.

Unsurprisingly, the speed-ups achieved by SCert (Figure 5.8(a)) are higher in the scenarios where CERT suffers from higher abort rates (Figure 5.8(b)). This shows that also with realistic and complex applications, like STMBench7, the speculation mechanism employed by SCert succeeds in significantly reducing the abort rate, boosting the throughput, on average, by about 45%.

5.3.3 Lee Benchmark

Finally, this section considers another complex benchmark, namely Lee-TM. Figure 5.9 reports the results achieved by SCert with respect to CERT computed considering the number of transactions required to run the benchmark for a fixed amount of time (180 seconds), when using the two protocols. The number of replicas varies between 2 and 8 and the number of threads was fixed to 2. In this case, the performance gains achieved by SCert are negligible. The initial phase of the Lee benchmark is composed by small transactions that have low conflict rates among different replicas, but high conflict rates within the same replica. In such scenario, the erroneous optimistic deliveries generate cascading aborts in the same replica, leading to the increased abort rate shown in the Figure 5.9(b). This performance degradation is compensated by (*i*) the overlap between transaction executions and communication, and (*ii*) early conflicts detection, generating similar throughput results as the one reported by CERT (see Figure 5.9(a)). This indicates that, although SCert is not always a better solution for workloads as the one just shown, it does not outperform the baseline protocol.

5.4 Discussion

This chapter has introduced a new Speculative Certification protocol, named SCert, to implement distributed replicated STMs. SCert leverages on Optimistic Atomic Broadcast (OAB) protocols to speed-up the propagation of write-sets, reducing the number of transactions that read stale data and allowing early detection of conflicts among transactions. This novel manner of exploiting OAB is much more suited for STM implementations than previous strategies designed for database replication, that were based on active replication. By aggressively using speculation, SCert is able to achieve performance gains of up to 4.5x when compared to non-speculative certification schemes. SCert was integrated in the framework that will be presented in Chapter 6, which allowed STM applications to transparently leverage on the performance benefits achievable by SCert via a fully fledged prototype.

5.5 Summary

This chapter introduced the SCert protocol. It started by presenting the motivation for this protocol and the SCert architecture. The protocol is composed by two important components, namely the STM extensions to support speculative versions and the Replication Manager. The chapter finished with a performance evaluation of SCert.

Notes

The results presented in this chapter were accomplished with a joint work with other members of the GSD research group, namely Luís Rodrigues and Paolo Romano. The SCert protocol was proposed in the paper: "SCert: Speculative Certification in Replicated Software Transactional Memories", Proceedings of the 4^{th} Annual International Systems and Storage Conference, Haifa, Israel, June 2011.

6 GenRSTM Architecture

The work described in this dissertation aims at building efficient distributed and replicated STMs. The different techniques explored in the previous paragraphs represent a step towards this goal. However, as noted in the discussion of each of the previous protocols, each solution is more favorable for a specific operational envelope. For instance, ALC excels when each node exhibits some data locality in its access pattern and SCert requires the network delays to have low variance.

This chapter starts by comparing the performance of these and other related protocols under different conditions. The results from these experiments highlight that is very unlikely that a single protocol will be able to outperform all the other protocols for all system configurations and workloads. Therefore, any STM system designed and implemented to support a single configuration is likely to perform poorly as the system evolves with time. To address the problem above, an architecture is proposed to support multiple distributed STM configurations, in a single extensible and generic framework. The framework makes easy to replace different replication protocols, communication services, and form of local STM support. Therefore, a distributed and replicated STM can be easily reconfigured to match an evolving operational envelope.

The remaining of the chapter is organized as follows. Section 6.1 shows results obtained with several configurations of distributed and replicated STMs. Section 6.2 describes the proposed generic replicated STM architecture, including the minimum set of interfaces to achieve a generic framework. Section 6.3 illustrates the execution of some instances that can be generated using the proposed architecture and Section 6.4 exemplifies how to build applications and protocols for the proposed framework. Finally, Section 6.5 concludes this chapter.

6.1 Performance of STM Replication Under Different Scenarios

This section shows a simple performance evaluation of several algorithmic approaches for building replicated STMs across a set of heterogeneous workloads. The design space of replicated STM platforms encompasses the exploration of a number of complex trade-offs across three main software layers:

- the replication layer, namely the module in charge of ensuring transactional consistency among distributed replicas;
- the local STM, which is in charge of regulating concurrency among parallel threads executing on a given replica;
- the Group Communication Service (GCS), namely the software component that implements lower level abstractions/services such as, Atomic Broadcast and Group Membership (Guerraoui & Rodrigues 2006).

This study will focus exclusively on the evaluation on the first two of the above mentioned layers. Given the abundance of published solutions, for what concerns both STMs and replication of transactional systems, an exhaustive evaluation of the cross-product of all their possible combinations is clearly prohibitive. Therefore, for each of the two layers, the analysis is restricted to some of the most relevant alternatives, which are representative of several fundamental trade-offs in the design of a replicated STM platform.

The STM runtime support system selected for evaluation are *JVSTM* and *TL2*, (see Section 2.3.2). Regarding the replication layer, three different certification-based replication schemes are considered, in addition to ALC a SCert: the *Non-Voting*, the *Voting* and the *Bloom Filter Certification* (BFC). The first two are described in the Section 2.2.4. BFC is included in D^{2}STM, which is described in the Section 3.4.1.

To conduct the experiments, aimed at assessing the differences of the system performance with different configurations, different distributed replicated STM configurations were deployed on a cluster of 8 nodes, each one equipped with two Intel Quad-Core XEON at 2.0 GHz, 8 GB of RAM, running Linux 2.6.32-26-server and interconnected via a private Gigabit Ethernet. In fact, the tested configurations were deployed using the generic framework that will be

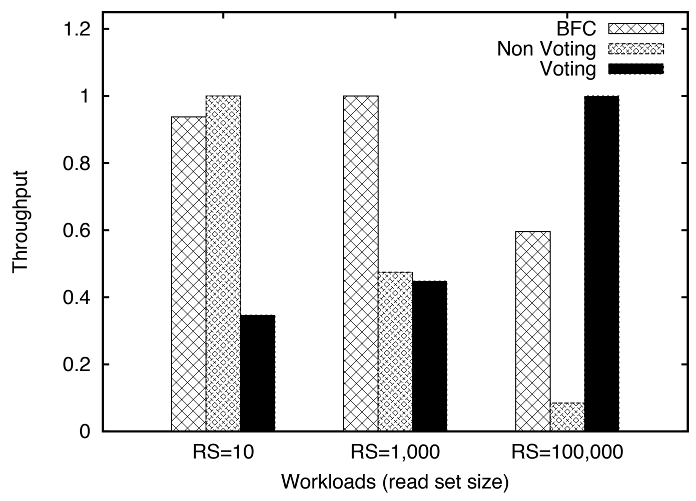


Figure 6.1: Throughput of replication protocols.

described in subsequent session: this avoided the need to re-implement the same configurations using different software integration middleware. To better understand the difference among the tested protocols, the results shown in the following are normalized with respect to the protocol exhibiting best performance in each considered workload scenario. The benchmarks used on the performance evaluation are described in the Section 3.4.2.

6.1.1 Impact of the Replication Protocol

The Bank Benchmark was used to assess the impact of changing the replication layer. This is a simple benchmark that has the advantage of providing a fine control on the conflict rate. The benchmark was initialized such that all nodes replicate an array of accounts of size $numMachines \cdot numThreads \cdot 240,000$ items. Depending on the accounts touched by each read-write transaction, the workload generated can have from 0% to 100% conflicts among concurrent transactions.

The experiments to evaluate the performance of the system were performed using different workloads. The tests focused on a contention-free test case to factor out one additional variable characterizing the workload and simplifying the analysis of the results. The number of items read by each transaction was configured to 10, 1.000 and 100.000 data items, and the number of items written was fixed to one. These three different workloads have an impact on the traffic generated to the network, and the time it takes to certify a transaction. The number of replicas was fixed to 8 and the number of threads on each replica was always 3. Figure 6.1 compares the system throughput of the three different replication protocols, on the specified scenarios. On this scenario the STM layer was configured to use always JVSTM and the replication layer to use BFC, Non-Voting and Voting. BFC was configured to have at most 1% of aborts due to false positives.

The first scenario is composed of transactions with a read-set of 10 items. In this scenario, the non-voting protocol is able to achieve better throughput than the other two protocols, because it generates small messages and does not saturate the network. The BFC protocol generates also small messages, but the overhead caused by the alternative validation scheme affects the system throughput. In the voting protocol each transaction generates two communication steps, increasing the time to commit a transaction. With small message sizes, this protocol is never able to achieve good performance when compared with non-voting protocols due to the extra communication phase. In the second scenario, where the read-set of transactions is composed of 1.000 items, the non-voting protocol generates big messages, saturating the network. The BFC protocol encodes the read-set in a Bloom Filter and is able to generate small messages, avoiding network congestion and achieving better performance than the non-voting and the voting protocols. In the third scenario, where the read-set of each transaction contains 100.000 items, the performance of both the BFC and the non-voting protocols decrease due to the saturation of the network. The voting protocol avoids sending the read-set to the other replicas at the cost of an extra communication step. This cost pays off when the read-set is very large.

Experiments comparing directly the ALC and SCert (proposed in the previous chapters) protocols were also conducted. The performance results conducted individually for each protocol presented in this thesis showed that both SCert and ALC can achieve better (or, in the worst case, the same) performance results than its baseline, the non-voting protocol. But there are also scenarios where one protocol can achieve better throughput results than the other.

As an example, of this behavior, the Figure 6.2 depicts two scenarios of ALC and SCert running STMBench7. In this scenario, the benchmark was configured to use the "read-write" workload. Both protocols were executed with JVSTM in the STM layer, and with 2 and 4 replicas. In the scenario with only 2 replicas, the network is not saturated and the SCert protocol is able to generate memory snapshots that accelerate the execution of other concurrent trans-

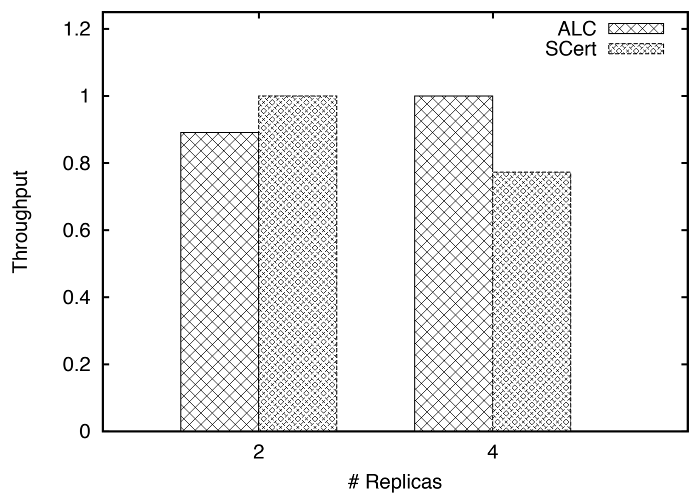


Figure 6.2: Throughput of ALC and SCert (STMBench7).

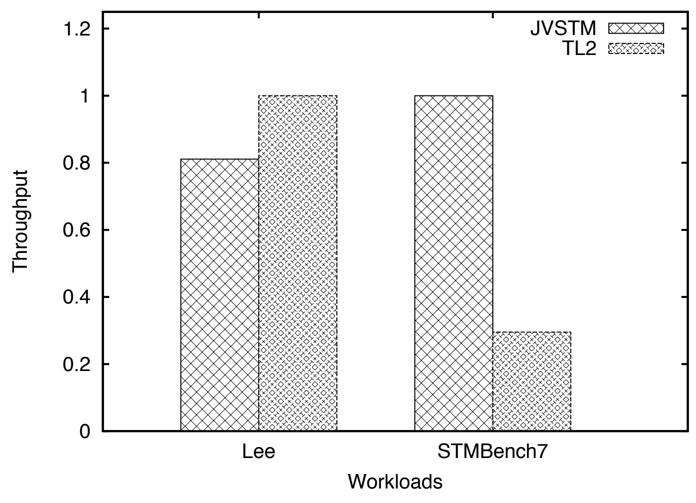


Figure 6.3: Throughput of JVSTM and TL2 (Lee-TM and STMBench7).

actions. The ALC protocol needs to send a LeaseRequest for the majority of the transactions, generating a throughput similar to the baseline Non-Voting protocol (see also Figure 4.8). With 4 replicas, the network starts to be saturated and the SCert protocol is more sensitive to this, due to the network reorder. The ALC protocol has mechanisms to avoid that its abort rate is increased due to network saturation, being able to achieve performance improvements 20% better than SCert (and the baseline Non-Voting protocol).

6.1.2 Impact of the STM Layer

The next two scenarios, depicted in the Figure 6.3, show the effects of changing the STM layer. The replication protocol was fixed, using always BFC, and the STM layer was configured to use TL2 and JVSTM. These tests were run using the Lee-TM and the STMBench7 benchmarks. The Lee-TM scenario shows that TL2 is able to achieve 20% better performance results than JVSTM. This is because in TL2 validation is performed at every read operation. On the contrary, with JVSTM transactions are only validated in the commit phase. The STMBench7 scenario, also shown in the Figure 6.3, was configured to use the "read dominated" workload. In a scenario composed by a mix of long running read only transactions and short write transactions, the multi-versioned scheme adopted by JVSTM is able to achieve better performance results. In JVSTM, read-only transactions never abort, since even if new versions of one object are created by concurrent transactions, long running read only transactions will always read a consistent memory snapshot. In this scenario, TL2 is forced to abort read-only transactions.

6.2 The GenRSTM Architecture

The results presented in the previous section show that, to achieve the best results, the different components of a distributed and replicated STM system must be carefully selected. The problem is further exacerbated by the fact that workloads can dynamically change over time, for instance, if the population of users grows, or as the system is enriched with new functionalities. Unfortunately, given the high heterogeneity of APIs exposed by existing solutions for distributed STMs, the choice of the STM platform is currently a committing one, locking the user application to specific middleware solutions.

GenRSTM, a Generic framework for Replicated STMs, was created to tackle these issues. GenRSTM has been designed in order to support, in an efficient and modular fashion, a wide range of heterogeneous algorithms across the various layers composing the software stack of a replicated STM platform, and specifically (*i*) replica consistency, (*ii*) local concurrency control and (*iii*) group communication system. Flexibility is achieved via a set of neat, reflective interfaces (Kiczales 1991; Maes 1987), which allow the replication manager to be notified of information/events reflecting the internal state of the local STM (such as the read-set/write-set of committing transactions, or the activation of new local transactions), as well as to alter the state of (possibly heterogeneous) local STMs on the basis of the outcome of the selected replica coordination protocol. Efficiency is achieved via the adoption of the *observer* (Gamma, Helm, Johnson, & Vlissides 1995) software design pattern, which allows to restrict the notifications exchanged between the STM and replication modules exclusively to the ones that are strictly needed by the specific configuration of the entire middleware stack.

The internal software architecture of GenRSTM relies on the *Inversion of Control* (Gamma, Helm, Johnson, & Vlissides 1995) and *Dependency Injection* (Gamma, Helm, Johnson, & Vlissides 1995) design patterns. By effectively separating the development of functional behavior from dependency resolution, this allows to enhance reusability by reducing coupling among software modules. It also allows to reduce the complexity of the individual components by sparing developers of new replication/STM modules from developing boiler-plate code to hard-code dependencies. GenRSTM is implemented using the Java programming language and is available for download as open source[1].

Overall, GenRSTM allows system administrators to seek optimal performance as a function of the workload/deployment scenario by reconfiguring the replicated STM middleware platform, in a transparent fashion for the user level application. Thanks to its modular and extensible design, and by making available a number of building blocks required by replicated STM solutions, GenRSTM aims at simplifying the development of new STM replication algorithms and at integrating the results from the growing community of researchers working in this area.

The goals of GenRSTM are defined in Section 6.2.1 and the architecture overview is described in Section 6.2.2. The APIs and interaction between the architecture components are described in Sections 6.2.3 and 6.2.4. The decoupling between the several components is described in Section 6.2.5.

6.2.1 GenRSTM Goals

The GenRSTM architecture focus on several goals, important to conduct some of the research directions pointed in the Section 3.4.1. The goals are the following:

[1] `http://code.google.com/p/genrstm/`

Goal #1: Simplify the development and testing of new replication protocols and STMs It is a simple way to test new protocols, by using useful building blocks which satisfy common requirements of replication protocols and STMs.

Goal #2: Provide high decoupling between the architecture building blocks It allows each building block to be implemented independently by using well defined interfaces.

Goal #3: Support multiple implementations of the architecture building blocks It allows each building block to be composed with other existing components, so that the system can be tuned to achieve the best performance of a specific workload and network characteristics.

These goals are possible to achieve by building an architecture that is able to integrate several key software components while, at the same time, maintaining a high degree of decoupling in each component to be able to implement new algorithms independently. The following paragraphs show how these goals are achieved.

6.2.2 Architecture Overview

The components of a node of the generic platform, depicted in the Figure 6.4, are structured into three main logical layers. The bottom layer is a Group Communication Service (GCS) (Guerraoui & Rodrigues 2006) which can provide two main building blocks: view synchronous membership (Guerraoui & Rodrigues 2006), and a set of interfaces that provide communication services. The ordering and fault tolerance guarantees offered by these services depend on the configuration of the underlying implementation. For instance, full replication protocols based on certification, such as the ones described in this dissertation, require the underlying toolkit to provide an Atomic Broadcast service (Guerraoui & Rodrigues 2006). The architecture uses a generic Group Communication Service for Java (jGCS) (Carvalho, Pereira, & Rodrigues 2006).

The core component of the generic architecture is represented by the Replication Manager (RM), that implements the distributed coordination protocol required for ensuring replica consistency. The RM interacts, on one side, with the GCS layer and, on the other side, with a local instance of a STM. Finally, the top layer of the architecture is a wrapper API that intercepts the application level calls for transaction demarcation (e.g., to begin and commit transactions) and

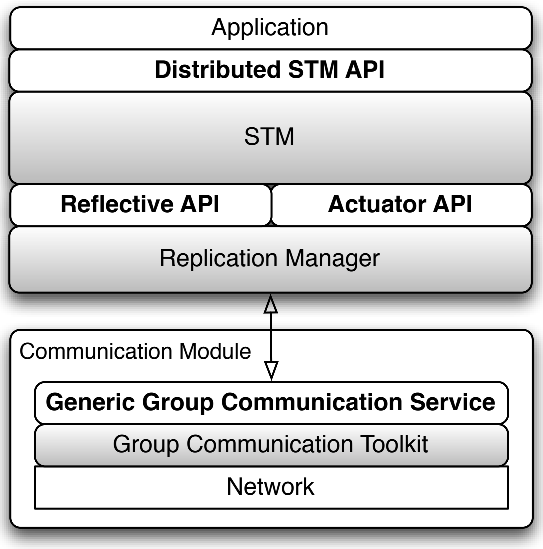


Figure 6.4: GenRSTM: components of a replica.

read/write operations. This approach allows to extend the classic STM programming model, allowing the concurrent execution of an arbitrary number of threads on each replica.

The integration of an STM within this generic architecture requires the implementation of five main extensions to allow the RM to interact with the STM without requiring re-implementing several components strictly related to the STM itself. More specifically, the STM should export its behavior by means of a reflective (Kiczales 1991) mechanism including the procedures needed by the replication protocols, namely:

1. add listeners for the transaction demarcation procedures (begin, commit and abort) and to read/write operations that can be used by the RM. This allows the RM to trigger the distributed coordination protocol required for ensuring replica consistency;

2. extract information concerning internals of the transaction execution, i.e., its read-set, write-set, and snapshot ID;

3. explicitly trigger the transaction validation procedure, that aims at detecting any conflict raised during the execution phase of a transaction T_x with any other (local or remote) transaction that committed after T_x started;

4. atomically apply the write-set of a remotely executed transaction and simultaneously increasing the STM's timestamp;

5. permit cluster wide unique identification of the memory objects created and updated by (remote) transactions. This is achieved by tagging each STM object with a unique identifier. A variety of different schemes may be used to generate universal unique identifiers (UIDs), as long as it is possible to guarantee the cluster-wide uniqueness of UIDs generated independently at each replica.

This will allow a more efficient solution, since the distribution protocols will use the mechanisms already implemented by the STM. The needed mechanisms already exist in most STM systems; they just need to be exported so they can be used by any external component. In the case of the presented architecture, they will be used by the RM. Each component of GenRSTM needs to have well defined interfaces. This allows that each component can be developed independently and composed with several compatible implementations. The interfaces needed in the system are the Distributed STM API, the Actuator API, the Reflective API and the Group Communication Service API. For the latest, GenRSTM uses the already existing Generic Group Communication Service for Java (jGCS) (Carvalho, Pereira, & Rodrigues 2006). This service is composed by a set of interfaces to send and receive messages to a group of processes. jGCS can be configured for using several communication paradigms, depending on the requirements of the replication protocol. The other APIs are described in the next sections.

6.2.3 Programmer's API

The STM Runtime Context API encapsulates the STM implementation and provides, to the application, methods to perform the following operations:

- `begin()` a new transaction – this operation starts a new transaction in the current thread;
- `commit()` a transaction – this operation attempts to commit a transaction previously started by the current thread. An exception is raised in the case that the transaction cannot be committed;
- `abort()` a transaction – this operation aborts a transaction previously started by the current thread;
- `GenRSTMObject` abstract class – must be extended to create replicated objects. The transactional objects must be serializable and uniquely identifiable among the distributed sys-

tem. The API provides a factory to generate unique identifiers. To create such identifiers, the API exports a factory that is implemented by the specific STM.

- `createBox()`, that creates a replicated Box – this operation is used to store and manage replicated objects or primitive types that are compatible with the framework. This method is provided by a specific factory, which is implementation dependent.

GenRSTM follows an Object Oriented model and was targeted to provide *field granularity* in its transactions. This is materialized by having each field encapsulated within a `Box`. A Box has methods to *read* (`get()`) the latest value, *write* (`put()`) a new value, get the data version of the latest committed value and commit the latest written value, which means making it visible to other transactions. Each `Box` needs also to be uniquely identified. A Box can contain a primitive value or a reference to a transactional object. Note that a `Box` could be replaced by a *byte-code* rewriting mechanism, but the practical result is the same, since the programmer still needs to mark somehow the fields in the objects that must be replicated.

With this simple, but intuitive, interface the framework provides a generic mechanism to shield applications from the details of the STM being used, either replicated or not. This contributes to achieve the Goal #1.

6.2.4 APIs Between RM and STM Layers

An STM creates and executes transactions, reflecting its internal behavior to the lower layer. This is abstracted by the `Transaction` class. This class keeps all the information needed by the underlying protocols, such as a `ReadSet`, a `WriteSet`, a transaction identifier (ID) and a data snapshot timestamp. A read-set is composed by a list of Box IDs. A write-set is composed by a list of the pairs (BoxID, value), where the value can be a primitive value, an ID of an existing distributed object, or a Serializable version of a new Object. If the object is new (was created inside the specified transaction), the whole object must be included in the write-set, so it can be sent through the GCS and applied in the remote replicas.

The `Transaction`, `ReadSet` and `WriteSet` interfaces are used in the Actuator and Reflective APIs. The Reflective API reflects the behavior of the transaction to the Replication Manager. This is achieved by adopting the Observer Design Pattern. The Replication Manager can implement the `TransactionListener` and/or the `OperationListener` interfaces, and register them on the STM layer. This way, the Replication Manager is notified on the `TransactionListener` when the following events occur:

- `onBegin(Transaction t)` – when a new transaction is started on the STM layer;
- `onCommitting(Transaction t)` – when a transaction is starting the commit phase; and
- `onFinished(Transaction t, boolean committed)` – when the transaction finished, either by committing or aborting.

If the Replication Manager registers also the `OperationListener`, it is notified when the following events occur:

- `onRead(Transaction t, Box b)` – when a `Box` is read; and
- `onWrite(Transaction t, Box b)` – when a `Box` is written.

These methods pass the execution control to the lower layer (RM) which can execute some task related to the received notification or pause the execution of the transaction, if it needs to wait for an external notification (e.g. a message to be delivered from the GCS layer).

The Actuator API is used by the RM to act on the STM, for two operations: atomically apply the write-set of a remote transaction and certify a (local or remote) transaction. These two primitives are materialized by the following methods:

- `apply(WriteSet ws)` – this method is used only to apply a remote and valid transaction and should be implemented by the STM, using its already existing internal mechanisms. It exposes new values on the Boxes and increments the data version, if the STM needs to maintain one.
- `validate()` – this method is also implemented and exposed by the STM and should be able to validate any transactions' read-set against the current memory state, returning `true` if the transaction is valid.

The `apply` method belongs to the STM instance and applies a new memory snapshot on the STM it self. The validate method is reached through the `Transaction` class and validates a specific transaction against the current committed memory snapshot.

In the case that the system is configured to use a replication protocol that uses speculative executions, the Actuator API has a set of methods to expose a new (optimistically committed) memory snapshot, that is not visible to applications, but is visible to new transactions that read these new snapshots based on the assumption that this new snapshot will be finally committed. For this class of protocols, the API exposes the `specCommit()` method, that exposes a new optimistic snapshot to new transactions, `specValidate()`, that validates a transaction taking into account also the optimistically committed memory snapshots, `commit()` that – in the case of speculative transactions that were already optimistically committed – exposes a final committed snapshot to the system, and the `specAbort()` that discards an optimistically committed snapshot. The execution of these methods is described with more detail in Section 5.2.2.

6.2.5 Enhancing Decoupling Between Components

To achieve Goals #2 and #3, the framework adopted not just the previously presented interfaces, but also the Inversion of Control (IoC) and Dependency Injection (DI) design patterns. IoC allows to achieve a high degree of decoupling among the previously described architecture components. By adopting the IoC design pattern, this framework allows the accommodation of several implementations of each module of the architecture (STM, Replication Manager and Group Communication Service). This has several benefits. First of all, the programmer is able to implement each building block independently, focusing only on the task of that specific component of the architecture. For instance, when a programmer is implementing a new replication protocol, he/she does not have to deal with ensuring message ordering or the details of applying a write-set of a remote transaction on the STM layer. Secondly, the several components can be reused and composed for a specific scenario or application. Using the same example, if a programmer needs to implement a new replication protocol, this new protocol can be composed with already existing implementations of STMs and Group Communication toolkits. Finally, using IoC means that replacing one component of the architecture by another that ensures the same type of guarantees will have no side effect on other components. The only effect of replacing an architecture component by another that provides the same guarantees is the resulting performance change, when executing the same application with a different setup.

For each building block, an implementation needs to define its dependencies and the system starts its execution after filling all those dependencies. This is achieved in the presented framework by making use of the DI design pattern. DI is used to ensure that all the dependencies among modules are met during the bootstrap of the system. Upon its instantiation, each module is provided a reference for the specified dependencies, avoiding the explicit creation of the required modules. The current prototype uses the Google Guice[2] DI framework. Note also that DI is used only when all the objects are instantiated, affecting the performance of the system only in the bootstrap. The performance of the system is not affected during its execution.

6.3 Configuration Examples

This section shows some examples that illustrate how the framework can be used and how several components can be configured to satisfy different application requirements. This is done by describing three configuration examples. The first example composes TL2 with BFC. In this example only the `TransactionListener` is registered in the STM layer. The second example describes the composition of JVSTM and the ALC replication protocol, running on top of the Appia GCS, illustrating the behavior of the system when the replication manager needs a GCS that provides different ordering guarantees. The third example shows the composition of JVSTM and SCert and illustrates a case where both the `TransactionListener` and `OperationListener` listeners are registered in the STM layer. All the examples describe the execution of a replicated transaction that increments the value on the variable x.

6.3.1 Example #1: Composing TL2 with BFC

This example illustrates how can an administrator combine TL2 with the BFC replication protocol, in the proposed architecture. The STM layer is instantiated to use the TL2 implementation, the RM layer is instantiated with BFC, and the jGCS layer can be configured to use any group communication toolkit that provides Atomic Broadcast (e.g. the APPIA toolkit (Miranda, Pinto, & Rodrigues 2001)).

Figure 6.5 depicts the execution of a replicated transaction being executed in this composition example. Here, the local STM notifies the RM about the beginning of a new transaction and when the transaction is preparing to commit. In the BFC protocol, the global validation

[2]http://code.google.com/p/google-guice/

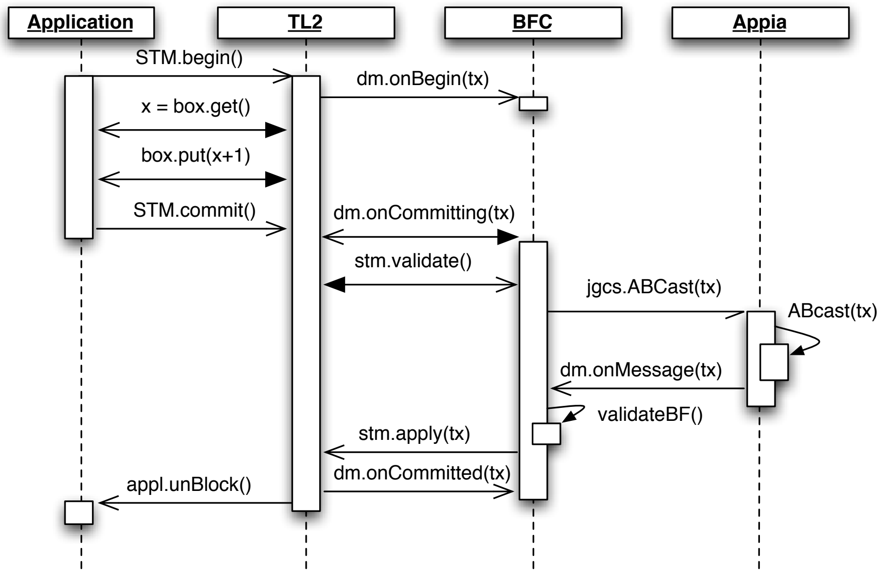


Figure 6.5: Sequence diagram for example #1.

process must use Bloom Filters, which is not supported natively by the STM layer. Transactions are validated by querying the Bloom Filter (of the transaction being validated) for object IDs contained in the write-sets of concurrent transactions. For this purpose, BFC must keep a log of concurrent transactions. This is done by adding to a hash map the reference of each starting transaction and removing it when a transaction finishes. This is done in the `onBegin()` and in the `onFinished()` notifications. In the `onCommitting()` notification, BFC starts by validating locally the transaction using the interface exported by the STM layer. If the transaction is valid, its write-set and (encoded) read-set is sent to the group via atomic broadcast. When the message is received, the transaction is globally validated using the Bloom Filter and the transaction is applied on the local STM layer (if it passes the Bloom Filter validation). Finally, BFC is notified that the transaction was finished and successfully applied, and the application thread is unblocked from the `commit()` method.

This example illustrates the behavior of the system in the case that the `OperationListener` is not registered, avoiding the extra overhead when reading and writing values on Boxes. It also shows that alternative ways can be used to perform operations on the STM, if they are not supported natively by the STM. In this case, the Replication Manager uses an alternative procedure to validate transactions with Bloom filters.

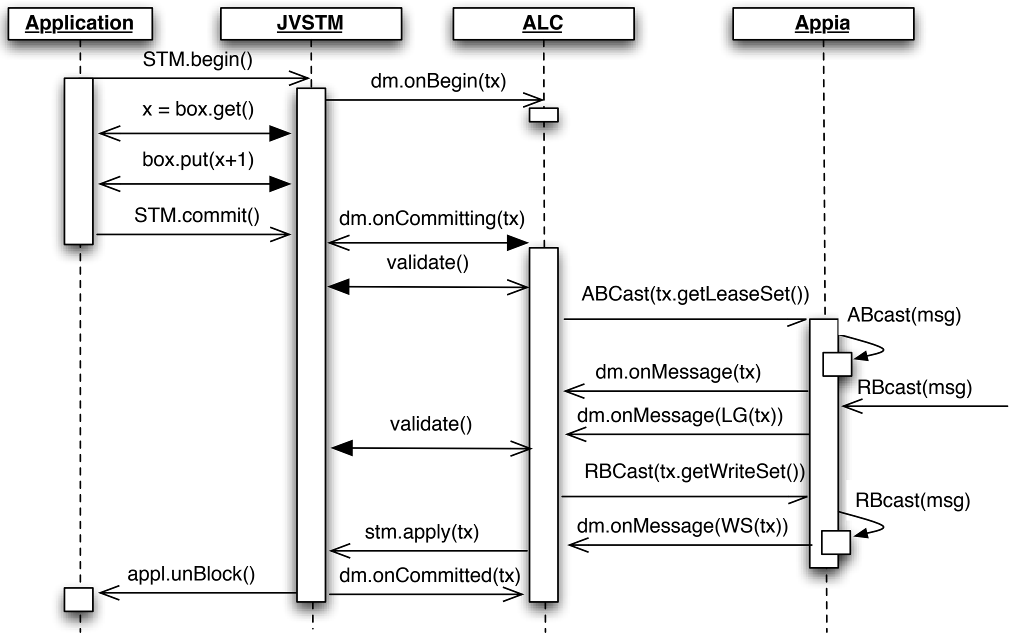


Figure 6.6: Sequence diagram for example #2.

6.3.2 Example #2: Composing JVSTM with ALC

The second example illustrates how can JVSTM and the ALC replication protocol be combined, in the proposed architecture. The STM layer is instantiated to use the JVSTM implementation, the RM layer is instantiated with ALC, and the jGCS layer can be configured to use any group communication toolkit that provides atomic broadcast and reliable broadcast within the same group of replicas (e.g. APPIA (Miranda, Pinto, & Rodrigues 2001) or Spread (Amir, Danilov, & Stanton 2000)).

Figure 6.6 depicts the execution of the same replicated transaction being executed in this composition example. The application executes the transaction, interacting only with the local STM. The local STM notifies the RM about the beginning of a new transaction, and when the transaction is preparing to commit. In the `onCommitting()` notification, ALC validates locally the transaction to avoid sending it to the group in the case that it is not valid at this point. If the transaction is valid, ALC internally checks if the current replica owns all the leases needed to validate and commit the transaction. In this case, let us assume that the replica does not hold the leases and the optimizations described in the Chapter 4 are turned off. The replica must send the Lease Request message through atomic broadcast and will receive a Lease Granted message

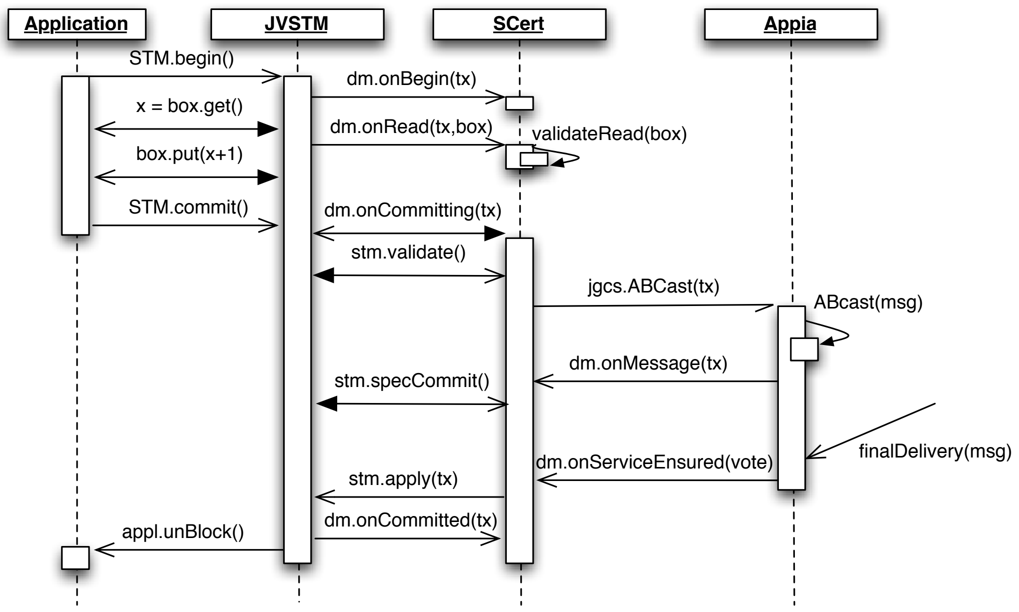


Figure 6.7: Sequence diagram for example #3.

from a remote replica, through the reliable broadcast communication channel. After enabling this Lease Request, the replica can validate the transaction and send the write-set to the other replicas. All the replicas apply this write set and the owner replica unblocks the thread.

This example shows the behavior of the system when the Replication Manager needs to have a Group Communication toolkit capable of sending messages with different ordering guarantees.

6.3.3 Example #3: Composing the extended version of JVSTM with SCert

The third and final example illustrates how the extended version of JVSTM is combined with the SCert replication protocol (presented in Chapter 5), in the proposed architecture. The STM layer is instantiated to use the JVSTM implementation, the RM layer is instantiated with SCert, and the jGCS layer can be configured to use any group communication toolkit that provides atomic broadcast with optimistic message deliveries. Currently, the only toolkit that provides this service is APPIA (Miranda, Pinto, & Rodrigues 2001).

Figure 6.7 depicts the execution of the same replicated transaction being executed in this composition example. The application executes the transaction, interacting only with the local STM. The local STM notifies the RM about the beginning of a new transaction, when a box is read, and when the transaction is preparing to commit. Note that, as discussed before, JVSTM validates transactions only in the commit phase. This behavior can be changed by the replication layer by registering a `OperationListener` to receive notifications about read boxes and checking if it is reading an obsolete value. In the `onCommitting()` notification, SCert validates locally the transaction to avoid sending it to the group in the case that it is not valid at this point. If the transaction is valid, its write-set is sent to the group via optimistic atomic broadcast. When the message is optimistically delivered to the replication protocol, the transaction is validated according to this order and a new snapshot is generated for new incoming transactions. Note that the application remains blocked until the final delivery of the message. Upon the transactions' final delivery, the order is checked and the transaction is revalidated (if needed). The transaction is finally applied on the local STM layer. Finally, the SCert protocol is notified that the transaction was finished and successfully applied, and the application thread is unblocked from the `commit()` method.

This example shows the behavior of the system when the Replication Manager registers the `OperationListener` to intercept also read and write operations. The need to have a Group Communication toolkit capable of providing optimistic message deliveries is also illustrated.

6.4 Using the GenRSTM Framework

This section shows how to use the GenRSTM framework by illustrating two examples that are more relevant in the context of this thesis: a simple application compatible with GenRSTM and a replication protocol built for the proposed framework.

6.4.1 Building a Simple Application

This section shows how can a programmer build a simple application with the proposed framework. Listing 6.1 depicts the code needed to build an application that increments a distributed shared counter.

Listing 6.1: Example Application.

```
 1  public class ExampleAppl {
      class Counter extends GenRSTMObject {
 3      private Box<Integer> value;
        public Counter() {
 5        value = STMRuntime.getRuntime().getBoxFactory().
            createBox(OidFactory.getStaticOID("ID"),
 7          new Integer(0));
        }
 9      int getCounter() {
          return value.get();
11      }
        void setCounter(int value) {
13        this.value.put(value);
        }
15    }

17    private Counter counter;
      public ExampleAppl(){counter = new Counter();}
19
      public void incrementCounter(){
21      STMRuntime.getRuntime().begin();
        counter.setCounter(counter.getCounter()+1);
23      STMRuntime.getRuntime().commit();
      }
25
      public static void main(String[] args){
27      PropertyReader.setProperty("ReplicationProtocol",
                                          args[0]);
29      PropertyReader.setProperty("GCNumInitialMembers",
                                          args[1]);
31      ExampleAppl appl = new ExampleAppl();
        STMRuntime.getRuntime().boot();
33      appl.incrementCounter();
      }
35  }
```

In this example there is only one thread executing a transaction, but the programmer is free to create concurrent threads accessing shared objects. The proposed framework detects both local and remote conflicts. The example presents two classes: an application implemented by the `ExampleAppl` class and the inner class `Counter` that represents a replicated application object.

To build an application, the programmer needs to create the replicated objects by extending the basic `GenRSTMObject` (lines 2 to 15). This provides already the skeleton code needed for the distribution of the objects. Since boxes are dependent from the STM implementation, they must be created through a factory provided by the STM runtime environment (lines 5 to 7). The framework provides also a factory to create object identifiers. These identifiers must be unique in the cluster and identify one single replicated object. There is also a method to create an Object with a specific ID, which is used in the provided example. This is important to bootstrap the system by creating shared root domain objects in all the nodes. Root domain objects are always needed in these kind of systems to allow that any replica can access the replicated objects.

The STM runtime environment provides also methods to begin a new transaction in the current thread and to commit a transaction that started in the current thread. Replicated boxes are read and written through the `get()` and `put()` methods (lines 10 and 13). These methods must be called within a transaction, as illustrated in the lines 20 to 24.

The system is started as shown in the lines 27 to 32. The first code lines are optional and show how to dynamically override default properties, initially configured in a static configuration file. After all the properties are set, and the root domain objects are created, the programmer simply calls the `boot()` method (line 32) from the STM runtime environment. This will inject all the needed dependences, create all the needed objects and join to the previously configured group of replicas. Once all the replicas joined, the `boot()` method returns and the application can start executing replicated transactions.

6.4.2 Building a Replication Protocol

This example illustrates how to build a replication protocol on the proposed framework. Listing 6.2 depicts the code needed to implement a simplified version of the NonVoting replication protocol.

Listing 6.2: Example Replication Protocol.

```java
public class NonVotingProtocol implements ReplicationProtocol {
 private GroupCommunication groupCommunication;
 private DataSession dataSession;
 private BlockSession controlSession;
 private CountDownLatch waitForMembers;
 private AtomicBoolean booted;
 private Service everythingEnsured = null;
 private ApplyProcess apply;
 private RuntimeContext runtimeContext;

 @Inject
 public NonVotingProtocol(GroupCommunication gc, RuntimeContext ctx){
  waitForMembers = new CountDownLatch(1);
  booted = new AtomicBoolean(false);
  groupCommunication = gc;
  runtimeContext = ctx;
 }

 public void boot(){
  try {
   everythingEnsured = groupCommunication.getAllEnsuredService();
   GroupConfiguration config = groupCommunication.getConfiguration();
   dataSession = groupCommunication.getProtocol().openDataSession(config);
   controlSession = groupCommunication.getProtocol().openControlSession(config);
   dataSession.setExceptionListener(this);
   dataSession.setMessageListener(this);
   dataSession.setServiceListener(this);
   controlSession.setBlockListener(this);
   controlSession.setMembershipListener(this);
   controlSession.join();
   Configuration conf = new DefaultConfiguration(controlSession.getLocalAddress());
   ObjectIDFactory.instance().initialize(conf);
   apply = new ApplyProcess(runtimeContext);
   try {
    waitForMembers.await();
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
   runtimeContext.setLocalID(controlSession.getLocalAddress().toString());
  } catch (JGCSException e) {
   e.printStackTrace();
  }
 }

 // TRANSACTION PROCESSORS
 public void onBegin(Transaction tx) {}

 public Response onCommitting(Transaction tx) {
  WriteSet ws = tx.getWriteSet();
  ReadSet rs = tx.getReadSet();
  if(!runtimeContext.validate(rs, ws))
   return Response.ABORT;
  try {
   broadcastReadWriteSet(tx.getTxID(), ws, rs, tx.getNumber());
  } catch (IOException e) {
   return Response.EXCEPTION;
  }
  return Response.WAIT;
 }

 public void onFinished(Transaction tx, boolean committed) {}

 // GROUP MESSAGE PROCESSORS
 public Object onMessage(Message message) {
  ProtocolMessage pm = null;
  try {
   pm = createMessageInstance(message.getPayload());
   pm.unmarshal();
   pm.setSenderAddress(message.getSenderAddress());
  } catch (IOException e) {
   e.printStackTrace();
  } catch (ClassNotFoundException e) {
   e.printStackTrace();
  }
  return pm;
 }

 public void onServiceEnsured(Object msg, Service service) {
  try {
   if(service.compare(everythingEnsured)>=0){
    ProtocolMessage pm = (ProtocolMessage) msg;
    if(pm instanceof ReadWriteSetMessage){
     ReadWriteSetMessage rwsMsg = (ReadWriteSetMessage) pm;
     apply.orderIncomingXact(rwsMsg);
    }
   }
  } catch (UnsupportedServiceException e) {
   e.printStackTrace();
  }
 }

 // MEMBERSHIP PROCESSORS
 public void onMembershipChange() {
  Membership membership = null;
  try {
   membership = controlSession.getMembership();
  } catch (NotJoinedException e) {
   e.printStackTrace();
  }
  if(!booted.get() && membership.getMembershipList().size() ==
     groupCommunication.getNumInitialMembers()){
   booted.set(true);
   waitForMembers.countDown();
  }
 }

 public void onBlock() {
  try {
   controlSession.blockOk();
  } catch (NotJoinedException e) {
   e.printStackTrace();
  } catch (JGCSException e) {
   e.printStackTrace();
  }
 }
}

class ApplyProcess {
 private ConcurrentLinkedQueue<OrderedXactEntry> orderedTransactions =
         new ConcurrentLinkedQueue<OrderedXactEntry>();
 private Thread thread;
 private ReentrantLock lock;
 private Condition waitingValidation;
 private RuntimeContext runtimeContext;

 ApplyProcess(RuntimeContext rc){
  runtimeContext = rc;
  lock = new ReentrantLock();
  waitingValidation = lock.newCondition();
  thread = new Thread(new Runnable(){
   public void run(){
    processXacts();
   }
  });
  thread.setName("Applier Thread");
  thread.start();
 }
```

```
139   public void orderIncomingXact(ReadWriteSetMessage msg){
        lock.lock();
141     try{
          OrderedXactEntry entry;
143       entry = new OrderedXactEntry(msg.getTxId(),msg.getWriteSet(),
                      msg.getReadSet(),msg.getDataVersion());
145       orderedTransactions.add(entry);
          waitingValidation.signal();
147     }
        finally {
149       lock.unlock();
        }
151   }

153   protected void processXacts(){
        while(true){
155       lock.lock();
          try{
157         while(orderedTransactions.isEmpty()){
              try {
159             waitingValidation.await();
              } catch (InterruptedException e) {
161             e.printStackTrace();
              }
163         }
            OrderedXactEntry entry = orderedTransactions.remove();
165         if(runtimeContext.validate(entry.getReadSet(), entry.getWriteSet()))
              runtimeContext.applyWriteSet(entry.getWriteSet(),false);
167       } finally {
            lock.unlock();
169       }
        }
171   }
    }
```

A replication protocol is created by implementing the interface `ReplicationProtocol` which is composed by several interfaces needed to interact with both the Group Communication Service (GCS) and the STM architecture components. The programmer also needs to implement some methods to bootstrap the system. The next paragraph describes these methods.

The class constructor is shown in the lines 11 to 16 and illustrates the Dependency Injection mechanisms. By tagging the constructor with a `@Inject` annotation, the programmer is indicating to the DI tool that when an Object of this type is created, the DI tool must provide an instance of an object that implement the `GroupCommunication` interface and an object that implements the `RuntimeContext` interface. The first contains all the needed information to join a group of replicas and the second contains the implementation of STM specific methods, needed to validate and apply transactions. the `boot()` method (lines 18 to 43) will be called by the framework upon the system bootstrap and the protocol must register all the needed listeners and join the group of replicas. This method must block until a pre-configured number of replicas successfully joined the same group. In this example, the method is unblocked in the `onMembershipChange()` method (lines 100 to 104).

The replication protocol receives three notifications from the STM layer: the `onBegin()`, the `onCommitting()` and the `onCommitted()`. In this protocol, the only relevant notification if the `onCommitting()` (lines 48 to 60). When this method is called, the protocol first checks if the transaction is valid at this point. If the transaction is not valid, the protocol returns an `ABORT` response to finish immediately the transaction without committing it. This avoids broadcasting the transaction when it is already doomed to abort. If the transaction is still valid at this point, its read-set, write-set, transaction ID and memory version is sent through Atomic Broadcast to the group of replicas and the STM layer is notified to block the execution of the thread until the transactions' outcome is known. This last step is done by returning a `WAIT` response to the STM layer.

Messages are received through two methods, namely `onMessage()` and onServiceEnsured(). These methods belong to the jGCS interface (Carvalho, Pereira, & Rodrigues 2006) and provide the notion of optimistic deliveries, which in this example is only used to unmarshal the message contents. The first method (lines 64 to 76) corresponds to the notification of the received message, without any ordering guarantees. When the message is ordered and uniform, a final notification is received in the `onServiceEnsured()` method (lines 78 to 90) with the unmarshaled message contents. This message is delivered to the applier thread using a shared Queue.

The applier thread is implemented in this example by the ApplierProcess class (lines 118 to 171). The applier thread is dedicated to validate and apply all the (local or remote) transactions that are received through Atomic Broadcast, respecting the reception order. This is done in the `processXacts()` method (lines 153 to 170).

6.5 Discussion

Using results obtained experimentally with a real prototype, this chapter shows that, for the same workload, different replication approaches lead to different performance results. This indicates that the system must be configured to use the appropriate STM and replication mechanisms that better fit particular workloads, leading to the second contribution of this chapter:

an architecture that facilitates the integration and execution of multiple replication techniques in a single, coherent, middleware infrastructure. This paves the way towards the development of autonomic mechanisms, able to select in runtime the most appropriate replication technique for the workload at hand.

There are several systems that also try to accommodate several STM algorithms. DSTM2 (Herlihy, Luchangco, & Moir 2006b) represents, to the best of our knowledge, the first generic framework proposed to simplify and homogenize the development and comparison of alternative *non-distributed* STM schemes. Specifically, the focus of the DSTM2 framework is on the comparison of different contention management algorithms (namely on the policy to be adopted by an STM upon detection of a conflict between transactions) in a single versioned STM. A more recent, also non-distributed, generic STM framework is Deuce (Korland, Shavit, & Felber 2009), which, unlike DSTM2, also allows to accommodate implementations of multi-versioned STMs. Unlike GenRSTM (and DSTM2), Deuce makes extensive usage of *byte-code* injection and dynamic *byte-code* rewriting in order to maximize transparency towards applications. Since Deuce was not tailored for distribution or replication, it lacks the necessary support for distribution, which would increase the complexity of developing, debugging and maintaining new STM replication protocols (which represents one of the main design goals of GenRSTM). Unlike DSTM2 and Deuce, GenRSTM targets replicated STMs distributed across a set of nodes. As a consequence, the GenRSTM includes a series of additional modules (i.e. the RM and GCS) providing flexible support for generic replication mechanisms. This framework also defines a set of reflective interfaces to allow the replication manager to access state and functionalities which classic STM APIs do not expose to programmers (e.g. methods to access the read-set and write-set of transactions, or to apply the write-set of remotely updated transactions).

The closer existing solution to this proposal is DiSTM (Kotselidis, Ansari, Jarvis, Luján, Kirkham, & Watson 2008), which is the only framework for distributed STMs we are aware of. With respect to GenRSTM there are a number of relevant differences. First, being based on the aforementioned DSTM2, DiSTM can locally support exclusively single-versioned STMs. Further, the focus of GenRSTM is on replicated STMs, whereas DiSTM provides support for distributed, but not replicated, STMs. Finally, by relying on a generic Group Communication Service (jGCS), the proposed framework allows to seamlessly integrate with a wide range of different communication paradigms.

GenRSTM is a framework that, in its current version, is able to integrate different replication schemes and STMs, being completely transparent to the application. GenRSTM was implemented in Java and is available for download as open source software. It allows the configuration of the system with an optimal setup for the given workload and network characteristics. This architecture simplifies the development and evaluation of alternative replication protocols, by encapsulating the three key components of a replicated STM that communicate through well defined interfaces. With the presented generic architecture new results can be integrated, serving also as a tool that helps the community to easily prototype and test new replication algorithms.

6.6 Summary

This chapter provided evidence that indicates that it is unlikely that a given configuration of a distributed replicated STM, based on a specific local STM support and an unique replication protocol, will provide good results in face of the heterogeneous workloads that such systems are likely to be subject. To address this limitation, this chapter introduced GenRSTM, an architecture that is able to accommodate several STM and replication algorithms. This architecture is materialized by a Java based framework that was also described in this chapter. With this framework, the system can be configured to use the best possible configuration for a specific application workload.

Notes

The results presented in this chapter were accomplished with a joint work with other members of the GSD research group, namely Luís Rodrigues, Paolo Romano and Maria Couceiro. Several of the results reported in this chapter have been achieved in cooperation with Maria Couceiro. The motivation for the proposed architecture was first presented in the paper “Generic Replication of Software Transactional Memory”, Doctoral Symposium of the 11^{th} International Middleware Conference, Bangalore, India, December 2010. The architecture was proposed in the short paper “A Generic Framework for Replicated Software Transactional Memories”, Proceedings of the 10^{th} IEEE International Symposium on Network Computing and Applications, Cambridge (MA), USA, August 2011.

7 Conclusions

This thesis has addressed the problem of building fault-tolerant distributed STM systems. A distributed STM provides the abstraction of a global address space, that can be accessed from threads in different nodes as a local STM. By allowing non-conflicting transactions to be executed in parallel in different nodes, distribution provides scalability to STMs. Replication keeps multiple copies of the data consistent, such that data is not lost if a node fails. Together, these features allow STMs to face the demanding scalability and dependability requirements imposed by enterprise systems.

The thesis has pinpointed a number of challenges in building such systems and contributed to identify a number of techniques to overcome those challenges. The exploration of these ideas resulted in two novel replication protocols tailored to STMs:

- The Asynchronous Lease-based Certification (ALC) protocol, a lease based replication protocol that increases the throughput of the system in face of workloads with data locality. The ALC protocol provides several key benefits. First of all, it has mechanisms to reduce the commit phase latency of the transactions that access a given set of data items. This ensures that no other replica will be allowed to validate any conflicting transaction, making it unnecessary to enforce distributed agreement on the global transactional serialization order. A lease based protocol takes advantage of this by limiting the use of Atomic Broadcast exclusively for establishing a global order for the lease ownership. Secondly, ALC shelter transactions from repeated abortions due to remote conflicts, ensuring the absence of remote conflicts on the subsequent re-execution of a transaction, provided that it deterministically accesses the same set of data items accessed during its first execution, as it is typically the case with realistic applications.
- The Speculative Certification (SCert) protocol, a speculative replication protocol that increases the performance of the system in scenarios with stable network latencies. The SCert protocol exploits early knowledge about message ordering in the underlying atomic

broadcast layer to propagate, in a speculative fashion, the updates of transactions before there is an agreement on the final serialization order. This speculative approach brings the two following key benefits. On one hand, it lowers the chances that transactions access stale snapshots, thus minimizing the probability of later incurring in an abort. On the other hand, it provides early conflict detection, thus reducing the amount of wasted computation and/or waiting time from transactions doomed to abort.

The results obtained with these protocols, and with protocols developed by other members of the Grupo de Sistemas Distribuídos at INESC-ID, show that it is unlikely that a single replication protocol can outperform all the other protocols, for all workloads that characterize STMs environments. Therefore, a contribution of this work is also a generic architecture that allows multiple replication protocols to coexist in a seamless manner in the same STM, more precisely:

- A generic and configurable architecture that can be tuned to optimize the performance of an STM system for different scenarios and workloads. This Java based framework integrates several STM mechanisms and all the replication protocols proposed in this thesis, as well as other existing replication protocols for comparison. All the protocols were evaluated using the presented framework with several heterogeneous STM based benchmarks.

7.1 Future Work

There are several possible optimizations to the proposed protocols that are deemed to be explored.

First, it would be interesting to identify techniques capable of effectively minimizing the frequency of rotation of leases among the replicas, so to maximize the performance gains achievable through the use of ALC. These include locality aware load balancing strategies, as well as mechanisms capable of adaptively adjusting the lease rotation mechanism based on the actual replicas (spatial/temporal) locality of reference.

Another technique that may improve the ALC protocol is to apply the key idea underlying the SCert protocol, namely speculatively propagating information on the data accessed by transactions prior to the completion of the group communication primitive used to disseminate this information. Merging both ideas into one single protocol can improve the system performance in some workloads.

However, the diversity of workloads that can be faced by a distributed STM make it unpractical to rely on a single protocol for optimal performance. The generic architecture proposed here opens the door to build adaptive solutions that can dynamically and automatically select the best replication protocol for a given deployment and workload, thus paving the way to the implementation of autonomic distributed dependable STMs that offer good performance in a wide range of scenarios. This is the main follow up of this thesis, which is already being addressed within our research group.

References

Abadi, M., T. Harris, & M. Mehrara (2009, February). Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, pp. 185–196. ACM.

Adl-Tabatabai, A.-R., C. Kozyrakis, & B. Saha (2006, December). Unlocking concurrency. *Queue 4*, 24–33.

Adve, S.V.; Hill, M. (1993, June). A unified formalization of four shared-memory models. *Transactions on Parallel and Distributed Systems 4*(6), 613–624.

Adya, A. (1999, March). *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D., MIT, Cambridge, MA, USA. Also as Technical Report MIT/LCS/TR-786.

Agrawal, D., G. Alonso, A. E. Abbadi, & I. Stanoi (1997). 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, pp. 496–503. Springer-Verlag.

Ahamad, M., G. Neiger, J. Burns, P. Kohli, & P. Hutto (1995). Causal memory: definitions, implementation, and programming. *Distributed Computing 9*, 37–49.

Amir, Y., C. Danilov, & J. Stanton (2000, June). A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 327–336.

Ansari, M., C. Kotselidis, I. Watson, C. Kirkham, M. Luján, & K. Jarvis (2008, June). Lee-TM: A non-trivial benchmark suite for transactional memory. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '08, Berlin, Heidelberg, pp. 196–207. Springer-Verlag.

Bartoli, A. & O. Babaoglu (1997, October). Selecting a "primary partition" in partitionable asynchronous distributed systems. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, SRDS '97, Washington, DC, USA, pp. 138–145. IEEE Computer Society.

Bernstein, P. A., V. Hadzilacos, & N. Goodman (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Bettina, K. & G. Alonso (1998, May). A suite of database replication protocols based on group communication primitives. In *Proceedings of the The 18th International Conference on Distributed Computing Systems*, ICDCS '98, Washington, DC, USA, pp. 156–163. IEEE Computer Society.

Bocchino, R. L., V. S. Adve, & B. L. Chamberlain (2008, February). Software transactional memory for large scale clusters. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, New York, NY, USA, pp. 247–258. ACM.

Cachopo, J. a. & A. Rito-Silva (2006, December). Versioned boxes as the basis for memory transactions. *Science of Computer Programming 63*, 172–185.

Carter, J. B., J. K. Bennett, & W. Zwaenepoel (1995, August). Techniques for reducing consistency-related communication in distributed shared-memory systems. *Transactions on Computer Systems 13*(3), 205–243.

Carvalho, N. (2010, November). Generic replication of software transactional memory. In *Proceedings of the 7th Middleware Doctoral Symposium*, MDS '10, New York, NY, USA, pp. 14–19. ACM.

Carvalho, N., J. Cachopo, L. Rodrigues, & A. Silva (2008, March). Versioned transactional shared memory for the FenixEDU web application. In *Proceedings of the Second Workshop on Dependable Distributed Data Management (in conjunction with Eurosys 2008)*, Glasgow, Scotland. ACM.

Carvalho, N., J. Pereira, & L. Rodrigues (2006, October). Towards a generic group communication service. In R. Meersman & Z. Tari (Eds.), *On the Move to Meaningful Internet Systems 2006: Distributed Objects and Applications*, Volume 4276 of *Lecture Notes in Computer Science*, Montpelier, France, pp. 1485–1502. Springer Berlin / Heidelberg.

Carvalho, N., P. Romano, & L. Rodrigues (2010, November). Asynchronous lease-based replication of software transactional memory. In *Proceedings of the ACM/IFIP/USENIX 11th Middleware Conference*, Bangalore, India, pp. 376–396.

Carvalho, N., P. Romano, & L. Rodrigues (2011a, August). A generic framework for replicated software transactional memories. In *Proceedings of the 10th International Symposium on Network Computing and Applications*, Cambridge, MA, USA. IEEE Computer Society.

Carvalho, N., P. Romano, & L. Rodrigues (2011b, June). SCert: Speculative certification in replicated software transactional memories. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, New York, NY, USA, pp. 10:1–10:13. ACM.

Cecchet, E., J. Marguerite, & W. Zwaenepole (2004). C-JDBC: flexible database clustering middleware. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, pp. 26–26. USENIX Association.

Chockler, G. V., I. Keidar, & R. Vitenberg (2001, December). Group communication specifications: a comprehensive study. *Computing Surveys 33*, 427–469.

Couceiro, M., P. Romano, N. Carvalho, & L. Rodrigues (2009). D^{2}STM: Dependable distributed software transactional memory. In *Proceedings of the 2009 15th Pacific Rim International Symposium on Dependable Computing*, PRDC '09, Washington, DC, USA, pp. 307–313. IEEE Computer Society.

Dice, D., O. Shalev, & N. Shavit (2006, September). Transactional locking II. In *Proceedings of the International Symposium on Distributed Computing*, pp. 194–208.

Dubois, M., C. Scheurich, & F. Briggs (1998). Memory access buffering in multiprocessors. In *25 Years of the International Symposia on Computer Architecture (selected papers)*, New York, NY, USA, pp. 320–328. ACM.

Duvvuri, V., P. Shenoy, & R. Tewari (2000, March). Adaptive leases: a strong consistency mechanism for the world wide web. In *Proceedings of the 19th Annual Joint Conference of the Computer and Communications Societies*, Volume 2, Tel Aviv , Israel, pp. 834–843. IEEE Computer Society.

Gamma, E., R. Helm, R. E. Johnson, & J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Garcia-Molina, H. & G. Wiederhold (1982, June). Read-only transactions in a distributed database. *ACM Transactions on Database Systems 7*, 209–234.

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, & J. Hennessy (1990, June).

Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, New York, NY, USA, pp. 15–26. ACM.

Goodman, J. R. (1989, March). Cache consistency and sequential consistency. Technical Report 61, University of Wisconsin-Madison.

Goodman, J. R. (1998). Using cache memory to reduce processor-memory traffic. In *25 Years of the International Symposia on Computer Architecture (selected papers)*, ISCA '98, New York, NY, USA, pp. 255–262. ACM.

Gray, C. & D. Cheriton (1989, November). Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *Operating Systems Review 23*, 202–210.

Gray, J., P. Helland, P. O'Neil, & D. Shasha (1996, June). The dangers of replication and a solution. *SIGMOD Record 25*, 173–182.

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

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

Guerraoui, R. & M. Kapalka (2009, February). How Live Can a Transactional Memory Be? Technical Report LPD-REPORT-2009-001, École Polytechnique Fédérale de Lausanne.

Guerraoui, R., M. Kapalka, & J. Vitek (2007). STMBench7: a benchmark for software transactional memory. In *Proceedings of the 2nd EuroSys European Conference on Computer Systems*, EuroSys '07, New York, NY, USA, pp. 315–324. ACM.

Guerraoui, R. & L. Rodrigues (2006). *Introduction to Reliable Distributed Programming.* Springer.

Gustavsson, S. & S. F. Andler (2002). Self-stabilization and eventual consistency in replicated real-time databases. In *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, New York, NY, USA, pp. 105–107. ACM.

Hansdah, R. & L. Patnaik (1986). Update serializability in locking. In G. Ausiello & P. Atzeni (Eds.), *ICDT '86*, Volume 243 of *Lecture Notes in Computer Science*, pp. 171–185. Springer

Berlin / Heidelberg.

Harmanci, D., V. Gramoli, P. Felber, & C. Fetzer (2010, October). Extensible transactional memory testbed. *Parallel Distributed Computing 70*, 1053–1067.

Harris, T., S. Marlow, S. Peyton-Jones, & M. Herlihy (2005, June). Composable memory transactions. In *Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, New York, NY, USA, pp. 48–60. ACM.

Harris, T., M. Plesko, A. Shinnar, & D. Tarditi (2006, June). Optimizing memory transactions. *SIGPLAN Notices 41*, 14–25.

Herlihy, M. (1991, January). Wait-free synchronization. *Transactions on Programming Languages and Systems 13*, 124–149.

Herlihy, M., V. Luchangco, & M. Moir (2003, May). Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, Providence, RI, USA, pp. 522–529. IEEE Computer Society.

Herlihy, M., V. Luchangco, & M. Moir (2006a, October). A flexible framework for implementing software transactional memory. In *Proceedings of the 21st Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, New York, NY, USA, pp. 253–262. ACM.

Herlihy, M., V. Luchangco, & M. Moir (2006b, October). A flexible framework for implementing software transactional memory. *SIGPLAN Notices 41*, 253–262.

Herlihy, M., V. Luchangco, M. Moir, & W. N. Scherer, III (2003, July). Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, PODC '03, New York, NY, USA, pp. 92–101. ACM.

Herlihy, M. & J. E. B. Moss (1993, May). Transactional memory: architectural support for lock-free data structures. *SIGARCH Computer Architecture News – Special Issue 21*, 289–300.

Herlihy, M. P. & J. M. Wing (1990, July). Linearizability: a correctness condition for concurrent objects. *Transactions on Programming Languages and Systems 12*, 463–492.

Imbs, D. & M. Raynal (2008, December). A lock-based STM protocol that satisfies opacity and progressiveness. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, Berlin, Heidelberg, pp. 226–245. Springer-Verlag.

Intel (2008). Cluster OpenMP. http://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers/.

IST (2009a). Fenix Framework. https://fenix-ashes.ist.utl.pt/trac/fenix-framework.

IST (2009b). FenixEDU. http://fenixedu.sourceforge.net.

Jiménez-Peris, R., M. Patiño Martínez, & G. Alonso (2002, October). Non-intrusive, parallel recovery of replicated data. In *Proceedings of the 21st Symposium on Reliable Distributed Systems*, SRDS '02, Washington, DC, USA, pp. 150–159. IEEE Computer Society.

Katsinis, C. & D. Hecht (2004, April). Fault-tolerant DSM on the SOME-bus multiprocessor architecture with message combining. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, USA. IEEE Computer Society.

Keleher, P., A. L. Cox, S. Dwarkadas, & W. Zwaenepoel (1994). TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, Berkeley, CA, USA, pp. 115–131. USENIX Association.

Keleher, P., A. L. Cox, & W. Zwaenepoel (1992, May). Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer architecture*, New York, NY, USA, pp. 13–21. ACM.

Kemme, B. & G. Alonso (2000, September). Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, San Francisco, CA, USA, pp. 134–143. Morgan Kaufmann Publishers Inc.

Kemme, B., F. Pedone, G. Alonso, & A. Schiper (1999, June). Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, Washington, DC, USA, pp. 424–431. IEEE Computer Society.

Kermarrec, A.-M., C. Morin, & M. Banâtre (1998, July). Design, implementation and evaluation of ICARE: an efficient recoverable DSM. *Software – Practice & Experience – Special issue on multiprocessor operating systems 28*, 981–1010.

Kiczales, G. (1991, October). Towards a new model of abstraction in software engineering. In *Proceedings of International Workshop on Object Orientation in Operating Systems*, pp. 127–128.

Korland, G., N. Shavit, & P. Felber (2009, May). Noninvasive Java concurrency with Deuce STM (poster). In *Proceedings of the Israeli Experimental Systems Conference*, Haifa, Israel.

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

Lamport, L. (1978, July). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*(7), 558–565.

Lamport, L. (1979, September). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers 28*, 690–691.

Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, & J. Hennessy (1990, May). The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, New York, NY, USA, pp. 148–159. ACM.

Li, K. (1988, August). IVY: A shared virtual memory system for parallel computing. In *Proceedings of International Conference on Parallel Processing*, pp. 94–101. Pennsylvania State University Press.

Maes, P. (1987, December). Concepts and experiments in computational reflection. *ACM SIGPLAN Notices 22*, 147–155.

Manassiev, K., M. Mihailescu, & C. Amza (2006, February). Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, New York, NY, USA, pp. 198–208. ACM.

Martin, M., C. Blundell, & E. Lewis (2006, July). Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters 5*, 17–17.

Miranda, H., A. Pinto, & L. Rodrigues (2001, April). Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st InternationalDST Conference* *on Distributed Computing Systems*, Phoenix, Arizona, pp. 707–710. IEEE Computer Society.

Morin, C. & I. Puaut (1997, September). A survey of recoverable distributed shared virtual memory systems. *Transactions on Parallel and Distributed Systems 8*(9), 959–969.

Palmieri, R., F. Quaglia, & P. Romano (2010, July). AGGRO: Boosting STM replication via aggressively optimistic transaction processing. In *Proceedings of the 9th International Symposium on Network Computing and Applications*, pp. 20–27.

Patiño Martínez, M., R. Jiménez-Peris, K. Bettina, & G. Alonso (2000). Scalable replication in database clusters. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, London, UK, UK, pp. 315–329. Springer-Verlag.

Pedone, F., R. Guerraoui, & A. Schiper (2003, July). The database state machine approach. *Distributed Parallel Databases 14*, 71–98.

Pedone, F. & A. Schiper (2003, January). Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science - Special Issue: Distributed Computing 291*, 79–101.

PostgreSQL Global Development Group (2011). PostgreSQL. http://www.postgresql.org.

Riegel, T., P. Felber, & C. Fetzer (2006, September). A lazy snapshot algorithm with eager validation. In *Proceedings of the International Symposium on Distributed Computing*, pp. 284–298.

Rodrigues, L., H. Miranda, R. Almeida, J. a. Martins, & P. Vicente (2002, October). The globdata fault-tolerant replicated distributed object database. In *Proceedings of the First EurAsian Conference on Information and Communication Technology*, EurAsia-ICT '02, London, UK, pp. 426–433. Springer-Verlag.

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

Schneider, F. B. (1993). *Replication management using the state-machine approach*, pp. 169–197. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Schöttner, M., S. Frenz, R. Göckelmann, & P. Schulthess (2004). Fault tolerance in a DSM cluster operating system. In U. Brinkschulte, J. Becker, D. Fey, K.-E. Groupietsch,

C. Hochberger, E. Maehle, & T. A. Runkler (Eds.), *ARCS Workshops*, Volume 41 of *LNI*, pp. 44–53. GI.

Shavit, N. & D. Touitou (1995). Software transactional memory. In *Proceedings of the 14th Annual Symposium on Principles of Distributed Computing*, PODC '95, New York, NY, USA, pp. 204–213. ACM.

Slony Development Group (2011). Slony-I. http://slony.info.

Terracotta (2009). Terracotta. http://www.terracotta.org.

Torrellas, J., H. S. Lam, & J. L. Hennessy (1994, June). False sharing and spatial locality in multiprocessor caches. *Transactions on Computers 43*, 651–663.

Transaction Processing Performance Council (2002). *TPC Benchmark^{TM} W, Standard Specification, Version 1.8.* Transaction Processing Performance Council.

Vicente, P. & L. Rodrigues (2002, October). An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of the 21st Symposium on Reliable Distributed Systems*, SRDS '02, Washington, DC, USA, pp. 92–101. IEEE Computer Society.

Vounckx, D., G. Deconinck, J. Vounckx, R. Lauwereins, & J. A. Peperstraete (1993, May). Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback. In *Proceedings of the IASTED International Conference on Modeling and Simulation*, pp. 262–265.