INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Cache Coherence in Distributed and Replicated Transactional Memory Systems

Maria Isabel Catarino Couceiro

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

Júri

Presidente:	Prof. Doutor Pedro Manuel Moreira Vaz Antunes de Sousa
Orientador:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Vogal:	Prof. Doutor João Manuel Santos Lourenço

September 2009

Agradecimentos

I would like to start by thanking my advisor, Prof. Luís Rodrigues, not only for this opportunity, but also for his guidance and advices during the elaboration of the thesis.

I also want to thank Paolo Romano and Nuno Carvalho, for all the help and support they provided throughout the development of this work.

Finally, I wish to thank my family and friends for their endless support, encouragement and patience.

This work was partially funded by the FCT research grant PTDC/EIA/72405/2006 in the context of the Pastramy project.

Lisboa, September 2009

Maria Isabel Catarino Couceiro

For my parents,

António and Margarida.

Resumo

Os sistemas de Memória Transaccional em Software (MTS) têm vindo a aumentar a sua relevância no desenvolvimento de aplicações concorrentes. No entanto, a construção deste tipo de sistemas recorrendo à replicação e à distribuição, com o objectivo de melhorar tanto o desempenho como a tolerância a faltas, ainda não se encontra devidamente explorada. Este trabalho aborda este problema através da análise, concretização e avaliação de técnicas que mantêm a coerência dos dados replicados num protótipo funcional ao qual se deu o nome de D^{2}STM. As transacções são processadas de forma autónoma por um único nó, evitando-se qualquer comunicação entre réplicas durante a sua execução. A coerência é assegurada no momento de confirmação por algoritmos distribuídos de certificação. Em particular, esta tese também apresenta o algoritmo denominado por BFC (*Bloom Filter Certification*), que explora os mecanismos de codificação dos filtros de Bloom para reduzir substancialmente o custo inerente à coordenação das várias réplicas (à custa de um aumento da probabilidade de uma transacção ser abortada, com um valor que pode ser ajustado pelo utilizador). Os algoritmos distribuídos de certificação foram concretizados e avalidados recorrendo a uma bancada experimental. Os resultados obtidos mostram que o BFC permite atingir ganhos consideráveis (aproximadamente 30% quando comparado com um esquema de certificação sem votação clássico), mesmo quando o aumento na percentagem de transacções abortadas é muito reduzido (por exemplo, 1%).

Abstract

Software Transactional Memory (STM) systems have emerged as a powerful paradigm to develop concurrent applications. At current date, however, the problem of how to build efficient distributed and replicated STMs to enhance both dependability and performance is still largely unexplored. This thesis addresses this problem by analyzing, implementing and evaluating techniques to maintain the consistency of the replicated STM data using a functional prototype named D^{2}STM. Transactions are autonomously processed on each node, avoiding any replica inter-communication during transaction execution. Consistency is enforced at transaction commit time by the implemented non-blocking distributed certification schemes. This thesis also introduces BFC (Bloom Filter Certification), which exploits a novel Bloom Filter-based encoding mechanism that permits to significantly reduce the overheads of replica coordination at the cost of a user tunable increase in the probability of transaction abort. Standard STM benchmarks, used evaluate the implemented distributed certification schemes, show that the BFC scheme permits to achieve remarkable performance gains (approximately 30% when compared to a classic non-voting certification scheme) even for negligible (e.g. 1%) increases of the transaction abort rate.

Palavras Chave
Keywords

Palavras Chave

Memória Transaccional em Software

Filtros de Bloom

Replicação

Coerência

Keywords

Software Transactional Memory

Bloom Filters

Replication

Coherence

Índice

List of Figures

List of Tables

089568 00000 n 0000044171 00000 n 0000039078 00000 n 0000241603 00000 n 0000244428 00000 n 0000089215 00000 n 0000089451 00000 n 0000089497 00000 n 0000093471 00000 n 0000095770 00000 n 0000097714 00000 n 0000100340 00000 n 0000093363 00000 n 0000089766 00000 n 0000095416 00000 n 0000095653 00000 n 0000095699 00000 n 0000097357 00000 n 0000097597 00000 n 0000097643 00000 n 0000099992 00000 n 0000100223 00000 n 0000100269 00000 n 0000104603 00000 n 0000106519 00000 n 0000109623 00000 n 0000104495 00000 n 0000100507 00000 n 0000106165 00000 n 0000106402 00000 n 0000106448 00000 n 0000109278 00000 n 0000109506 00000 n 0000109552 00000 n 0000114998 00000 n 0000114890 00000 n 0000109764 00000 n 0000115078 00000 n 0000115108 00000 n 0

1 Introduction

Software Transactional Memory (STM) systems have emerged as a powerful paradigm to develop concurrent applications (Herlihy, Luchangco, Moir, & William N. Scherer 2003; Harris & Fraser 2003; Dice, Shalev, & Shavit 2006). When using STMs, the programmer is not required to deal explicitly with concurrency control mechanisms. Instead, the programmer has only to identify the sequence of instructions, or transactions, that need to access and modify concurrent objects atomically. As a result, the reliability of the code increases and the software development time is shortened.

For scalability and fault-tolerance reasons, it is relevant to build distributed and replicated implementations of this paradigm. In such implementations, the STM is replicated in multiple nodes, which have to coordinate to ensure the consistency of the cached data. One example of this type of system is the FenixEDU system (Carvalho, Cachopo, Rodrigues, & Rito-Silva 2008), a web application in which the transactional semantics of an operation are supported by a STM (Cachopo & Rito-Silva 2006) running at each application server.

One of the goals of the project where this work is included is to build a more scalable and fault-tolerant version of the FenixEDU system. In this system, servers run an STM and use a logically centralized database to store the data and also as a synchronization mechanism to maintain their cache consistent. In the current architecture of the FenixEDU system, every time a transaction is started, the application server has to access the database to check if its cache is still up-to-date. Since caches only need to be updated when a remote transaction is committed, unnecessary accesses could be avoided by having the application servers exchange messages when a write transaction commits.

This work studies techniques that allow to maintain the coherence of the replicated STM data, identifies some limitations in the existing techniques, proposes some alternatives that may help in avoiding these limitations, and provides an implementation and evaluation of the proposed techniques.

	Read/Write set size	
	Average	Maximum
Read set of read-only transactions	5,844	63,746,562
Read set of write transactions	47,226	2,292,625
Write set of write transactions	35	32,340

Table 1.1: Average and maximum read set and write set sizes for each kind of transaction in the FenixEDU system.

1.1 Motivation

As it will be shown later in the thesis, one of the most promising techniques to build a distributed and replicated STM is based on the use of total order broadcast primitives (Défago, Schiper, & Urbán 2004) to disseminate control data and serialize conflicting transactions. Typically, the information exchanged in the payload of the total order broadcast includes the read and write sets of the transactions, i.e., the set of data items that have been read and written by the transaction during its execution at one replica. Therefore, this technique, called certification, avoids replica coordination during the execution phase, running transactions locally in an optimistic fashion.

Even though certification based replication schemes appear attractive to apply in the STM context, the overhead of schemes based on the total order broadcast primitive can be particularly detrimental in STM environments (Romano, Carvalho, & Rodrigues 2008). This is due to the fact that, unlike classical database systems, STMs neither incur in disk access latencies nor in the overhead of SQL statement parsing and plan optimization, which makes the execution time of typical STM transactions much shorter and lead to a corresponding amplification of the overhead of inter-replica coordination costs.

Table 1.1 shows the size of read and write sets of transactions in the FenixEDU system collected over a period of two weeks (Carvalho, Cachopo, Rodrigues, & Rito-Silva 2008). From this data it is clear that read sets can be extremely large. Thus, the large size of read sets can be identified as one possible obstacle to the implementation of efficient replicated STM systems.

To mitigate the problem above, this work proposes the use of Bloom filters (Bloom 1970) as a space-efficient technique to disseminate read sets and, as a consequence, to significantly reduce the overhead of the distributed certification scheme at the cost of a marginal, and user configurable, increase of the transaction abort probability.

1.2 Contributions

This work addresses the problem of implementing efficient distributed and replicated software transactional memory systems. More precisely, the thesis analyzes, implements and evaluates techniques to maintain the coherence of the replicated STM data. As a result, the thesis makes the following contributions:

- a novel non-voting certification scheme, named Bloom Filter Certification (BFC), that exploits a space-efficient Bloom Filter-based encoding to represent a transaction's read set, allowing to drastically reduce the overhead of the distributed certification phase at the cost of a reduced (but controlled) increase in the risk of transaction aborts;
- the identification of workload scenarios that benefit from the use of BFC.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- A prototype that integrates four distributed transaction certification processes, namely plain voting and non-voting certification schemes and two versions of Bloom Filter Certification(BFC). Both versions of BFC replace a transaction's read set by a Bloom filter in the replica synchronization message, but the last one also explores a voting certification scheme to achieve significantly compressed messages without incurring in a higher transaction abort rate.
- An experimental evaluation of the implemented certification protocols based on synthetic micro-benchmarks, as well as more complex STM benchmarks.

1.4 Research History

This work was performed in the context of the Pastramy[1] project (Persistent and highly Available Software TRAnsactional MemorY). One of this project's main goals is to build efficient distributed and replicated software transactional memory systems.

[1] `http://pastramy.gsd.inesc-id.pt`

From the very beginning, the main objective of this work was to design a new, more efficient, replica consistency protocol based on certification for a distributed and replicated STM. This new protocol, BFC, follows a non-voting approach, with the novelty being on the use of the Bloom filter technique to reduce the size of the messages exchanged by replicas. Additionally, in order to validate the results produced by the BFC, a standard version of the algorithm without any mechanism of message compression was implemented. Later on, a new protocol also using Bloom filters but this time combining a voting with a non-voting approach was designed with the objective of achieving higher message compression rates without increasing the transaction abort rate to equally high levels. This was preceded by the implementation of a standard voting algorithm, because of its many similarities with this second version of BFC.

In the end, this allowed us to compare all these four approaches using a variety of benchmarks and workloads, identifying their strengths and drawbacks, and determining the scenarios where each protocol performs better.

During my work, I benefited from the fruitful collaboration with the remaining members of the GSD team working on Pastramy, namely Paolo Romano and Nuno Carvalho.

Previous descriptions of this work were published in (Couceiro, Romano, Carvalho, & Rodrigues 2009a) and (Couceiro, Romano, Carvalho, & Rodrigues 2009b).

1.5 Structure of the Document

The rest of this document is organized as follows. Chapter 2 provides an introduction to the different technical areas related to this work. Chapter 3 introduces D^{2}STM, a Dependable Distributed Software Transactional Memory, describing in detail the replica consistency protocols implemented and Chapter 4 presents the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by summarizing its main points and future work.

2 Related Work

2.1 Introduction

Understanding the problems addressed in this thesis implies knowing the fundamentals of different areas in distributed systems.

This chapter starts by addressing Software Transactional Memory (STM) systems (Section 2.2), which are the basis for building a distributed STM (DSTM) system. To understand how STMs can be distributed, there is a brief survey about Distributed Shared Memory systems (Section 2.3), followed by a description of DSTM systems inspired by those algorithms (Section 2.4). Then, with the aim of replicating the STM we start by introducing some relevant communication and coordination abstractions, namely group communication primitives (Section 2.5). Those primitives offering total order can be used for communication between sites, ensuring consistency and serializability. Since databases and STM systems share the same key abstraction of atomic transaction, some mechanisms used in database replication (Section 2.6) may be applied in a replicated STM. Furthermore, replicated databases also share some goals with Distributed Memory Systems (and consequently with DSTM systems). Finally, Bloom filters (Section 2.7) are introduced as a mechanism that may be used to reduce the size of the messages exchanged by sites.

2.2 Software Transactional Memory

2.2.1 Transactional Memory

A concurrent program is one that uses multiple threads of control (also named just threads or processes) that execute concurrently and access shared objects. These objects are called concurrent objects and can be defined as data objects shared by multiple processes in a concurrent system (Marathe & Scott 2004). To ensure the consistency of concurrent objects, the access

to their internal data needs to be controlled by some form of concurrency control. Classical concurrency control relies on low-level mechanisms such as locks. Unfortunately, ensuring the correctness of a lock-based program is very difficult, as a single misplaced or missing lock may easily compromise the consistency of data.

Even if locks are correctly placed, lock-based synchronization still has several important drawbacks: it is prone to deadlocks and to priority inversion (when a high priority thread is blocked by lower priority threads), it is vulnerable to thread failure (a thread may fail without ever relinquish acquired locks), poor performance in face of preemption and page faults (a thread may be prevented from executing while still holding locks) or lock convoying (a livelock phenomena that may occur where there is high contend for some shared resource).

Transactional memory (Herlihy & Moss 1993) is an abstraction that addresses the problems above by preventing the programmer from dealing explicitly with concurrency control mechanisms. Instead, the programmer has only to identify the sequence of instructions, or transactions, that need to access and modify concurrent objects atomically. Transactional memory can be defined as a generic non-blocking synchronization construct that allows correct sequential objects to be converted automatically into correct concurrent objects (Marathe & Scott 2004).

Transactional memory borrows the notions of atomicity, consistency and isolation from database transactions. The abstraction relies on an underlying runtime system that is able to enforce some target consistency criteria for the concurrent execution (such as serializability (Bernstein, Shipman, & Wong 1979)). If the consistency criteria is met, the transaction is committed, otherwise its execution is aborted and the transaction is restarted. The ability to abort transactions eliminates the complexity and potential deadlocks of fine-grain locking protocols. The ability to execute non-conflicting transactions simultaneously may lead to high performance.

2.2.2 Hardware vs Software Transactional Memory

The transactional memory abstraction can be implemented with hardware support or entirely on software. An hardware implementation of transactional memory was proposed in Herlihy & Moss (1993). It is based on extensions to multiprocessor cache coherence protocols and provides support for a flexible transactional language for writing synchronization operations,

which can be written as a transaction. However, this solution is blocking, i.e., uses mutually exclusive critical sections to serialize access to concurrent objects. Recently, a significant effort has been put in developing a software version of the abstraction (Herlihy, Luchangco, Moir, & William N. Scherer 2003; Harris & Fraser 2003; Dice, Shalev, & Shavit 2006). Most of these implementations use non-blocking synchronization algorithms, which allow asynchronous and concurrent access to concurrent objects while guaranteeing consistent updates using atomic operations. Software Transactional Memory (Shavit & Touitou 1995) (STM) is an example of such algorithms.

This solution provides a non-blocking implementation of static transactions, a special form of transactions in which all the concurrent objects accessed by a transaction are known in advance. The system works as follows: a transaction updates a concurrent object only after a system wide declaration of its update intention, so that other transactions are aware that a particular concurrent object is going to be updated. The declaring transaction is called the owner of the object and it needs exclusive ownership in order to ensure atomic updates. One of the most significant limitations of this solution is that the system needs to have a previous knowledge of all the objects the transaction is going to access to guarantee ordered access. Consequently, it cannot access other concurrent objects during its execution rather than those predetermined. Recent work (Herlihy, Luchangco, Moir, & William N. Scherer 2003; Harris & Fraser 2003; Dice, Shalev, & Shavit 2006) has focused on providing software transactional memory algorithms for concurrent objects dynamically, since many applications allocate and use data structures dynamically.

2.2.3 Versioned Software Transactional Memory

Software transactional memory (STM) systems perform better when the number of transaction restarts is low compared to the overall number of transactions. This means that the number of conflicts should be minimized.

Versioned Software Transactional Memory (Cachopo & Rito-Silva 2006) is a STM designed with the aim of minimizing the number of aborts. For that purpose, the system maintains the data encapsulated in versioned boxes, which are boxes that may hold multiple versions of their contents. The use of per-transaction boxes, holding values that are private to each transaction, ensures that read-only transactions never conflict with other transactions. The per-transaction

box mechanism is implemented as a Java library (JVSTM[1]) and is currently being used in the FenixEDU project (Carvalho, Cachopo, Rodrigues, & Rito-Silva 2008) to reduce the conflicts on the collections used to implement the domain relationships. Section 3.5 presents a more detailed description of some of the JVSTM's internal mechanisms.

2.3 Distributed Shared Memory

Distributed Shared Memory (DSM) systems aim at combining the advantages of two architectures: shared memory systems (all processors have equal access to a single global physical memory) and distributed memory systems (multiple processing nodes that communicate by means of message passing). A DSM system implements a shared memory abstraction on top of message passing distributed memory systems. It inherits the ease of programming and portability from the shared memory programming paradigm and the cost-effectiveness and scalability from distributed memory systems (Protic, Tomasevic, & Milutinovic 1996).

2.3.1 Data Sharing

Data sharing can be provided at different levels and with different granularities. The most common approaches consist of supporting the sharing of a paged virtual address space (in which case the granularity is the hardware page size) or supporting the sharing of just specific variables of a parallel program (where the granularity can have a finer grain).

A classical example of the first approach is described in Li & Hudak (1989). In this system, processes share a unique paged virtual address space, divided into a set of fixed-size blocks (pages). Virtual to physical address translation is done by means of the standard memory management unit, which detects when a page is not loaded in memory and triggers a page fault exception. This exception results in the transference of the page to the machine where it occurred. A more recent description of this type of system can be found in Kontothanassis et al. (2005). Since whole pages are transferred, performance degrades due to the false sharing problem. False sharing (Bolosky & Scott 1993) happens when the shared data size is less than the page size and an apparent but not real conflict occurs between two copies of a page. In

[1] `http://web.ist.utl.pt/~joao.cachopo/jvstm`

other words, false sharing occurs when two or more processes modify non-overlapping parts of the same page.

The second approach solves the false sharing problem by sharing only the variables that need to be used by more than one process. Midway (Bershad, Zekauskas, & Sawdon 1993) implements this solution. At the programming language level, all shared data must be declared and explicitly associated with at least one synchronization object, also declared as an instance of Midway's data types, which include locks and barriers. The control of versions of synchronization objects is done using the associated timestamps, which are reset when data is modified (Protic, Tomasevic, & Milutinovic 1996).

2.3.2 Memory Consistency

It is possible to build a DSM system where a single copy of the data exists. When a node p needs to access a page that is the memory of node q, the page c is transferred from q to p and subsequently invalidated at q. Unfortunately, this approach is extremely inefficient, particularly in applications that have a majority of read operations. If p needs the page just for reading, it is generally preferable to keep a copy of the page at q and p so that multiple read operations can execute in parallel with no synchronization over the wire.

When data is not replicated, keeping it consistent is very straightforward, because accesses are sequenced according to the order in which they occur at the site where the data is held. Unfortunately this is no longer true when multiple copies exist, as the replica owners need to coordinate to ensure that all replicas are mutually consistent.

The behaviour of a (replicated) distributed shared memory system is defined by a consistency model. The stronger and most intuitive memory consistency model is the atomic model, which captures the behavior of a single copy (non-replicated) memory. In this model, the value returned by a read operation is always the same value written by the most recent write operation in the same address. A slightly weaker model is the sequential consistency (Lamport 1979; Morin & Puaut 1997), which ensures that all accesses to shared data are seen in the same order by every process, no matter where the data is located. In this model, the programmer uses the DSM as a shared multiprocessor. It is primarily used in page-based implementations (Li & Hudak 1989).

Unfortunately, the overhead of enforcing the previous memory consistency models is non-

negligible (Protic, Tomasevic, & Milutinovic 1996; Carter, Bennett, & Zwaenepoel 1995). Therefore, there was a significant amount of research in the quest of meaningful weaker memory consistency models that could be implemented in a more cost-effective manner (by requiring less synchronization and less data movements). Examples of such coherence semantics include weak, relaxed and entry consistency (Morin & Puaut 1997). Unfortunately, all these model require the programmer to be aware of the weak consistency of memory and obey to strict programming constraints, such as accessing shared variables within monitors.

2.3.3 Data Location

A key aspect in the design of a DSM system is how to locate the different replicas of a data item. Two main approaches can be used for this purpose: broadcast-based and directory-based algorithms (Li & Hudak 1989).

In the broadcast-based algorithms, when a copy of a shared data item must be located, a query message is broadcast in the network. This means that no information about the current location of shared data replicas needs to be kept. Each site manages precisely only those pages that it owns and when a page fault occurs, it sends a broadcast into the network to find the page's true owner. In spite of its simplicity, the performance of this approach can be poor since all sites have to process every broadcast message.

In the directory-based approach, information about the current location of shared data is kept in a global directory. For each data item there is a site, also referred to as the manager, that maintains the list of machines that own a copy of the page. Three types of directory organizations have been proposed for a page-based DSM system: central manager, fixed distributed manager and dynamic distributed manager.

In the central manager scheme, all pages have the same manager, which maintains the global directory. Whenever there is a page fault, the site where it occurred relies on the central manager to locate its owner. Therefore, as the number of sites and page faults increases, the central manager may become a bottleneck in the system. In order to avoid this situation, the managerial task may be distributed among individual sites. In the fixed distributed manager algorithm, the global directory is distributed over a set of sites and the association between a page and its manager is fixed. Even though performance improves when compared to the

centralized manager algorithm, it is difficult to find a fixed distribution function that fits the majority of applications. Finally, in the dynamic distributed manager algorithm, a manager site is dynamically associated with a page. The global directory entry related to a given page is stored in a single site, the page owner, and it may change during the execution of the application. When a site has a page fault, it sends a request to the page's probable owner indicated by its local directory. If that site is the true owner, it proceeds as the centralized algorithm. Else, it forwards the request to the probable owner indicated by its own local directory and does not need to send a reply to the requesting site. In this case, the performance does not degrade as more sites are added to the system, but rather as more sites contend for the same page.

2.4 Distributed Software Transactional Memory

Even though extensive work has been developed in the area of centralized STMs, i.e., for cache coherent shared memory systems, only recently has the design of distributed architectures for these systems started to raise interest. The most important difference between these two system architectures is that distributed STMs require expensive communication for remote accesses, typically provided by a software communication layer. This section briefly describes the works published so far (Kotselidis, Ansari, Jarvis, Lujan, Kirkham, & Watson 2008; Manassiev, Mihailescu, & Amza 2006; Bocchino, Adve, & Chamberlain 2008) with a special focus on the replica coherence protocols adopted by them.

2.4.1 Distributed Multiversioning

Analogously to JVSTM, the work in Manassiev, Mihailescu, & Amza (2006) implements a distributed multi-versioning scheme (DMV) by exploring the simultaneous presence of different versions of the same transactional dataset across the nodes. This allows read-only transactions to be executed in parallel with conflicting updating transactions by ensuring that the former are able to create a consistent snapshots of the dataset from older committed versions of the transactional data items. This way, several concurrent read only transactions that require different snapshots, but have disjoint read sets, can execute in parallel in the same node.

The following paragraphs describe the two consistency maintenance protocol implemented in DMV.

2.4.1.1 Update-Anywhere

Updates are only visible to other transactions upon commit. At commit time, each transaction creates a new version of the transactional memory, broadcasts the modifications made by itself to all the other nodes and, finally, it waits for their acknowledgements before committing the transaction locally.

Each update transaction obtains a cluster-wide unique token to enforce a consistent serialization order for this kind of transactions. This way, no two nodes attempt to commit conflicting transactions at the same time. Unfortunately, since committing a transaction imposes a two communication step synchronization phase (for updates propagation), the token acquisition phase can introduce considerable overhead by creating a bottleneck in the system and degrade performance.

Since each node may only keep one version of the data set at a time, the nodes in DMV explicitly delay applying (local or remote) updates to increase the chance of not having to invalidate a snapshot of currently active read-only transactions. Instead, they reply immediately after receiving a broadcast from other nodes in order not to introduce additional delay for committing update transactions. The modifications are applied in a lazy fashion when a local transaction needs them.

2.4.1.2 Scheduler-based Master-Update

In this scheme there is a scheduler that distributes the transactions on the cluster and is aware of their type and the versions they are supposed to read. This knowledge is used for scheduling the execution of update transactions on a master node and conflicting read-only transactions over a set of slave nodes.

The master node broadcasts the modifications resulting from update transactions to the set of slaves. The global serialization order is decided by the master's internal concurrency control. Similarly to the previous scheme, the slave nodes delay the application of the updates but reply to the master immediately. Each update on the master node creates a new version number which is communicated to the scheduler. The scheduler keeps track of the data versions that exist at each node and sends a read-only transaction to the slave with the smallest probability of conflicts.

2.4.2 DiSTM

The architecture used in DiSTM's (Kotselidis, Ansari, Jarvis, Lujan, Kirkham, & Watson 2008) implementation relies on a central master, in charge of coordinating the execution in the cluster. DiSTM has two core components: a transactional execution engine, DSTM2 (Herlihy, Luchangco, & Moir 2006), and a remote communication system, based on the ProActive framework (Baduel, Baude, Caromel, Contes, Huet, Morel, & Quilici 2006).

In DSTM2, all transactions are executed speculatively. When a transaction updates an object, instead of directly modifying the actual object, uses a cloned version of it, which is kept private until the transaction commits. Before a transaction commits, a validation phase is executed in order to detect and resolve conflicts. The transaction's read and write sets, containing the identifiers of the objects read/written during its execution, are broadcast to the remaining nodes, where they are validated against the objects contained in the read and write sets of the transactions currently executing there (the consistency protocol will be described in detail in future paragraphs). If the transaction can commit, its changes are made public, i.e., the shared objects are replaced with their respective modified cloned objects.

The key concept of the ProActive framework is the notion of active object. Each active object has its own thread of execution and can be distributed over the network. This means that remote copies of an active object can be created and active objects are remotely accessible via method invocation.

The following paragraphs briefly describe the Transactional Memory Coherence protocols implemented in this system.

2.4.2.1 Transactional Coherence and Consistency

DiSTM implements an adaptation of the decentralized Transactional Coherence and Consistency (Hammond, Wong, Chen, Carlstrom, Davis, Hertzberg, Prabhu, Wijaya, Kozyrakis, & Olukotun 2004) validation protocol. To maintain coherence, transactions acquire a global serialization number, called a ticket, from the master node before they start the remote validation, which corresponds to the broadcast of their read and write sets. When a conflict occurs, the transaction which attempted to remotely validate its read/write sets first "wins".

The coherence of the system is ensured by a master-centric, eager approach. After a transaction makes its changes visible locally, it updates the global dataset kept at the master node. In turn, the master node eagerly updates all the cached datasets on the rest of the nodes of the cluster. Upon updating the cached datasets, the transactions that did not read the most up-to-date values of the cached dataset are invalidated.

2.4.2.2 Lease-based Transactional Memory Coherence Protocols

2.4.2.2.1 Serialization Lease The role of the lease is to serialize the transactions' commits in the cluster, avoiding the expensive broadcast of transactions read and write sets for validation purposes. Upon commit, each transaction that passes the local validation stage tries to acquire the lease from the master node. If another transaction has the lease, the transaction that is acquiring the lease blocks and waits for its turn to commit, after adding itself to an ordered queue kept at the master node. Else, it acquires the lease. When the lease owner commits, it updates the global data at the master node and releases the lease. The master node then updates the cached datasets of the worker nodes. Any conflicting local transactions are aborted at this stage. Afterwards, if the next transaction in the queue was not aborted meanwhile, the master node assigns the lease to it.

Even though the cost of the messages broadcast is minimized when compared to the TCC protocol, the master node becomes a bottleneck for acquiring and releasing the leases. Besides, transactions waiting to be assigned the lease are blocked.

2.4.2.2.2 Multiple Leases In this protocol, multiple leases are assigned for transactions that attempt to commit. After a transaction passes the local validation phase, it tries to acquire a lease from the master node. The master node performs yet another validation phase, where the transaction is validated against each transaction that currently owns a lease. If there is no conflict, the transaction acquires the lease and commits. Else, it aborts and restarts. Upon successful commit, the transaction updates the global data at the master node and the master node updates the cached data at the worker node.

When compared to the serialization lease protocol where only one transaction could commit at a time, here transactions can commit concurrently. However, there is an extra validation step in the master node.

2.4.3 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 STM. This is achieved by assigning to each data item a home node, which maintains 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. The application is responsible for the caching and replication scheme, taking explicitly into account the issues related to data fetching and distribution, with an obvious increase in the complexity of the application development. Processors are treated as a flat set, which means that there is no difference between processors within a node and processors across nodes. Consequently, the availability of shared memory between multiple cores/processors on each replica is not exploited to speed up intra-node communication. Finally, contrasting with the two systems presented previously, Cluster-STM does not employ a multi-versioning local concurrency control to maximize the performance of read-only transactions, and is constrained to run only a single thread for each processor.

2.5 Group Communication

Group communication (Chockler, Keidar, & Vitenberg 2001; Powell 1996) is a way to provide multi-point to multi-point communication, by organizing processes in groups. The objective is that each process in the group receives copies of the messages sent to the group, often with delivery guarantees. The guarantees include agreement on the set of messages that every process of the group should receive and on their delivery order. Group communication is a powerful paradigm that can be used to maintain the consistency of multiples replicas of data, in order to build a replicated distributed software transactional memory system.

Here the focus will be on view-synchronous group communication systems (Birman & van Renesse 1994), which provide membership and reliable multicast services. The membership service maintains a list of the currently active and connected processes in a group, which is called view. The reliable multicast service delivers messages to the current view members: if a correct process sends a message m, then m is delivered by all correct processes in that view; if the sender is faulty, m may be delivered by either all correct processes in that view or by none of them.

The three most commonly provided types of ordering disciplines for reliable group communication services are FIFO, Causal and Total Order broadcast (Chockler, Keidar, & Vitenberg 2001). The FIFO service type guarantees that messages from the same sender arrive in the order in which they were sent at every process that receives them. This service can serve as a building block for higher level services. Causal order extends the FIFO order by ensuring that if two messages m and m' are sent and m causally precedes the broadcast of m' (according to Lamport's "happened-before" relation (Lamport 1978)) then every process that delivers both messages, delivers m before m'. Total Order broadcast (also known as Atomic broadcast) extends the Causal service by ensuring that all messages are delivered in the same order, as discussed below in more detail.

2.5.1 Total Order Broadcast

Agreement is a class of problems in distributed systems in which processes have to reach a common decision. Total Order broadcast (Défago, Schiper, & Urbán 2004) belongs to this class of problems: it is a reliable broadcast problem which must also ensure that all delivered messages are delivered by all processes in the same order.

Total Order broadcast can be defined in terms of two primitives, *TO-broadcast(m)* and *TO-deliver(m)*, where m is some message, and ensures the following properties:

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

The two first properties are liveness properties (Défago, Schiper, & Urbán 2004) (at any point in time, no matter what has happened up to that point, the property will eventually hold),

while the last two are safety properties (Défago, Schiper, & Urbán 2004) (if at any point in time the property does not hold, no matter what happens later, the property cannot eventually hold).

Uniform properties apply to both correct processes and to faulty ones. Since enforcing uniformity can be expensive in terms of performance, Agreement and Total Order properties can be defined as non-uniform:

- *(Regular) Agreement:* if a *correct* process TO-delivers a message m, then all correct processes eventually TO-deliver m.
- *(Regular) Total Order:* if two *correct* processes p and q both TO-deliver messages m and m', then p TO-delivers m before m' if and only if q TO-delivers m before m'.

Non-uniform properties may lead to inconsistencies at the application level if the application is not prepared to take the necessary corrective measures during failure recovery.

2.5.2 Implementing Total Order

According to (Défago, Schiper, & Urbán 2004), there are five different classes of algorithms to order messages, depending on the entity which generates the necessary information to define the order of the messages, as follows:

- Sequencer (process involved in ordering messages)
  - Fixed Sequencer: one process is elected as the sequencer and is responsible for ordering messages. There are three variants to this protocol, depending on the number of unicasts/broadcasts needed to sequence a message.
  - Moving Sequencer: based on the same principle as fixed sequencer algorithms. However, the role of sequencer can be transferred between several processes, in order to distribute the load among them. When a sender wants to broadcast a message m, it sends m to the sequencers. A token message, carrying a sequence number and a list of all sequenced messages, circulates among them. When a sequencer receives the token, it assigns a sequence number to all unsequenced messages and sends them to the destinations.

- Sender (process from which a message originates)
  - Privilege-Based: senders can broadcast messages only when they are granted the privilege to do so. When one wants to broadcast a message, it must wait for the token carrying the sequence number for the next message to broadcast. Upon reception, the sender assigns a sequence number to its messages, sends them to the destinations and updates the token.
  - Communication History: processes can broadcast messages at any time, since total order is ensured by delaying the delivery of messages, which carry a timestamp. The destinations decide when to deliver the messages based on their timestamp. A message is delivered when it will no longer violate total order.
- Destinations (processes to which a message is destined)
  - Destination Agreement: the delivery order results from an agreement between destination processes. This agreement can be on a message sequence number, on a message set or on the acceptance of a proposed message order.

2.5.3 Optimistic Total Order Broadcast

Total Order is very important for building distributed fault-tolerant applications but, unfortunately, its implementation can be very expensive. This is due to the number of communication steps and messages exchanged that are required to implement it, resulting in a significant delay before a message can be delivered (Rodrigues, Mocito, & Carvalho 2006). Optimistic Total Order aims at minimizing the effects of this latency, offering an additional primitive, called *TO-opt-deliver(m)*: the order by which a process TO-opt-delivers messages is an early estimate of the order by which it will TO-deliver the same messages (Rodrigues, Mocito, & Carvalho 2006). Based on this estimate, the application may (optimistically) perform a number of actions in parallel with the remaining communication steps of the total order protocol, which can be later committed when the final order is determined. However, sometimes the order by which messages are TO-opt-delivered may differ from the order by which they are TO-delivered. Consequently, aborting the steps executed optimistically may cancel the performance gains obtained when the optimistic order is correct.

2.6 Database Replication

The problem of keeping mutually consistent multiple replicas of transactional data items has been widely studied in the context of replicated database systems.

In Gray, Helland, O'Neil, & Shasha (1996), a classification based on two criteria is used to analyze existing database replication algorithms: the transaction location, that states in which replicas update transactions can be executed, and the synchronization point, that states at which stage of the transaction execution the inter-replica coordination is performed.

Regarding the transaction location criteria, two approaches can be identified: primary copy and update anywhere. In the primary copy approach, the primary replica executes all update transactions and propagates the write operations to the other replicas (secondaries). This approach simplifies conflict detection but is less flexible. On the other hand, the update anywhere approach allows both update and read-only transactions to be executed at any replica, which results in a more complex concurrency control mechanism but allows for non-conflicting updates to execute in parallel on different replicas. Note that, in any case, read-only transactions can always be submitted to any replica.

Regarding the synchronization point criteria, there are two possible approaches: eager and lazy. The eager replication technique requires coordination for the updates of a transaction to take place before it commits. Lazy replication algorithms asynchronously propagate replica updates to other nodes after a transaction commits.

Replication algorithms can also be classified based on whether the data is fully or partially replicated. In the former case, all replicas have a full copy of the database, whereas in the latter, each data item of the database has a physical copy on only a subset of sites.

In this thesis the focus is on variants of eager, update-anywhere protocols for fully replicated databases, as we aim at maximizing the concurrency of update transactions and still offer strong consistency. The following paragraphs contain a brief description of four different algorithms, namely: the Database State Machine approach (Pedone, Guerraoui, & Schiper 2003), Postgres-R (Kemme & Alonso 2000), SI-Rep (Lin, Kemme, no Martnez, & Jiménez-Peris 2005) and Optimistic Active Replication (Felber & Schiper 2001). These algorithms exemplify different features, such as certification type (local *versus* global), architectural alternatives (kernel-based *versus* middleware layer) and concurrency control mechanisms (shadow copies, snapshot iso-

lation, etc), to name but a few, that are useful to understand some fundamental replication mechanisms that could also be used in distributed STM systems.

2.6.1 The Database State Machine Approach

2.6.1.1 Overview

The Replicated State Machine Approach (Schneider 1993) is a classical algorithm to achieve fault-tolerance. This algorithm consists of running multiple replicas of a deterministic component. Replica coordination is achieved by ensuring that all replicas receive and process the same set of inputs in exactly the same order. For that purpose, state-machine commands are broadcast using a reliable totally ordered primitive, as described in Section 2.5.

The Database State Machine Approach (Pedone, Guerraoui, & Schiper 2003) (DBSM) applies this idea to replicated databases, by making the commit of a transaction a command to the replicated state-machine. More precisely, an update transaction is first executed locally at one replica. When the transaction is ready to commit, its read set and write set are broadcast to all replicas. Totally ordered broadcast is used, which means that all sites receive the same sequence of requests in the same order.

When the transaction information is delivered in total order, all replicas execute a global deterministic certification procedure. The purpose of the certification is to confirm that the transaction commit respects the target database consistency criteria (typically, one-copy serializability). If the transaction passes the certification test, its write-state is applied deterministically at each replica. A transaction passes the certification test if does not conflict with any concurrent already committed transactions. Transaction t_a and committed transaction t_b conflict if t_b does not precede t_a[2] and t_a's read set contains at least one data item from t_b's write set.

Thus, in the DBSM approach, replicas are not required to perform any coordination at each statement execution; all coordination is deferred to the commit phase. Due to this reason, the DBSM is said to use a deferred update technique. Note that if two different replicas execute

[2]Transaction t_b precedes transaction t_a if either the two execute at the same replica and t_b enters the committed state before t_a, or they execute at different replicas and t_b commits before t_a enters the committing state at a given replica.

concurrently two conflicting transactions, these conflicts will only be detected during the certification phase, when one of the transactions reaches the commit stage. This may lead to high transaction abort rates.

In order to reduce the abort rates, the certification test is modified based on the observation that the order in which transactions are committed does not need to be the same order in which the transactions are delivered to be certified. In the Reordering Certification test, a transaction is put in a list of committed transactions whose write locks have been granted but whose updates have not been applied to the database yet. The transaction is placed in a position before which no transaction conflicts with it, and after which no transaction either precedes it or reads any data item updated by it, thus increasing its probability of being committed. However, since the transactions in this list have granted write locks, its size needs to be carefully configured, or else it will introduce data contention, which would increase transaction abort levels.

2.6.1.2 Algorithm

During its execution, a transaction passes through three well-defined states: executing, committing and committed/aborted.

1. Executing: Read and write operations are locally executed at the replica where the transaction was initiated. When the commit is requested by the client, it moves to the committing state.

2. Committing: If a transaction is read-only, it is committed immediately. Otherwise, its read set and write set, as well as the updates performed by the transaction are sent to all replicas. Eventually, every replica certifies the transaction and all its updates are applied to the database.

3. Committed/Aborted: If the transaction is serializable with all previously committed transactions, its updates will be applied to the database. Local transactions in the first state holding read or write locks that conflict with its writes are aborted. The client receives the outcome for the transaction as soon as the local replica can determine whether it will be committed or aborted.

2.6.1.3 Discussion

When compared to immediate update techniques, the deferred update technique offers many advantages. The most relevant one is the fact that, by gathering and propagating multiple updates together, the number of messages exchanged between replicas is reduced. Since the transaction read and write sets are exchanged at commit time, the certification test is straightforward, as explained previously. However, the coordination messages may be excessively long and take a significant amount of time to be transmitted.

The resulting distributed algorithm runs an optimistic concurrency control scheme. Therefore, when the number of conflicts is high, it may lead to high abort rates. Clever re-ordering of ready to commit transactions may mitigate this problem.

2.6.2 Postgres-R

2.6.2.1 Overview

Postgres-R (Kemme & Alonso 2000) is a variant of the DBSM approach that avoids the need to disseminate the transaction read set during the commit phase. As the name implies, it was first implemented for the Postgres[3] database engine.

As is the case with the DBSM, transactions are executed locally at a single replica (that will be called the designated replica). While executing locally, update transactions perform all write operations on private (shadow) copies of the data. Shadow copies are used to check consistency constraints, fire triggers and capture write-read dependencies. Again, only when the transaction is ready to commit, a coordination phase is initiated. However, unlike the DBSM approach, in Postgres-R only the write set of the transaction is broadcast using the total order primitive.

As in DBSM, the total order defined by the atomic broadcast provides the basis for a serial execution of all certification procedures. However, since only the designated replica is aware of the transactions read set, only this replica can perform the certification test. The designated replica is then responsible for informing the remaining replicas of the outcome of the transaction, using a second broadcast, which does not need to be totally ordered.

[3] `http://www.postgresql.org`

2.6.2.2 Algorithm

In this replication protocol, a transaction is executed in four phases.

1. Local Read: Both read and write operations are performed locally, although write operations are executed on shadow copies. Before the execution of each operation, the proper lock is acquired.
2. Send: If the transaction is read-only, it is immediately committed. Otherwise, its write set is multicast to all sites.
3. Lock: When a write set is delivered, all the locks needed to execute each operation are requested in one atomic step.
   (a) For each operation on an item in the write set:
      i. Conflict test: if the local conflicting transaction is in its local read or send phase, multicast the decision message abort.
      ii. Grant the lock to the transaction if there is no lock on the item. Else, enqueue the lock request.
   (b) If the write set belongs to a local transaction, multicast the decision message commit.
4. Write: Updates are applied to the database when the corresponding write lock is granted. If the transaction is local, it can commit as soon as all updates have been applied. Else, it will terminate according to the corresponding decision message.

2.6.2.3 Discussion

The advantage of Postgres-R over DBSM is that the size of the exchanged messages can be significantly smaller, given that only the write set, which is typically smaller than the read set, needs to be broadcast. On the other hand, the certification of transactions can only be performed by the designated replica, because remote replicas do not possess the necessary information to test them for conflicts. This contrasts with the DBSM, in which a transaction is globally certified based on its read and write sets. Since the certification test is deterministic, in DBSM replicas need not exchange additional messages to inform each other that a transaction is to be committed or aborted. On the other hand, in Postgres-R the designated replica has to notify all other replicas of the outcome of every (local) transaction.

2.6.3 SI-Rep

2.6.3.1 Overview

SI-Rep (Lin, Kemme, no Martnez, & Jiménez-Peris 2005) is a database replication system that is implemented at the middleware level, i.e., as an intermediate layer between clients and the underlying DBMS, hiding DBMS internals from both user and the application. Thus, in opposition to DBSM and Postgres-R, which required changes to the database engine, SI-Rep can be used with legacy database engines. Its name comes from the fact that SI-Rep has been designed to provide snapshot isolation (Berenson, Bernstein, Gray, Melton, O'Neil, & O'Neil 1995) as the target consistency criteria. When using snapshot isolation (SI), transactions read from a snapshot of the data that was committed at the time the transaction was initiated, instead of reading data from the most up-to-date version. As a result, SI does not require the use of read locks and only detects conflicts between write operations: when such conflicts exist, only one of the conflicting transactions is allowed to commit.

With the purpose of reducing the validation overhead, a transaction is validated in two steps: local validation at the local middleware replica and global validation at all replicas. Local validation starts when the write set is retrieved, i.e., when the records updated by the transaction are exported from the local database replica to the local middleware replica at the end of its execution (before committing). The transaction is validated against the set of write sets that have been delivered but not yet executed and committed at the replica. It passes the validation test if it does not update data items present in any of the delivered write sets. On the other hand, during global validation a replica validates a transaction's write set against all the write sets delivered between the moment it was multicast by its local replica until it was delivered.

2.6.3.2 Algorithm

1. When a transaction is submitted to a middleware replica, it is first executed at the local database (DB) replica. Read and write operations are forwarded to the DB replica, which reads from a snapshot and writes new object versions.

2. As soon as the commit is requested, the middleware retrieves the write set from the DB replica. If it is empty, the transaction is read only and is committed locally. Otherwise,

the middleware multicasts it using Total Order broadcast to all other middleware replicas. Before the write set is multicasted, the middleware replica performs the local step of the validation process.

3. Each middleware replica has to perform validation for all write sets in delivery order. If the global validation succeeds, the middleware replicas apply the remote write sets in a lazy fashion and commit the transaction. Else, the transaction is aborted at the local replica.

2.6.3.3 Discussion

Middleware solutions have the advantage of simplifying the development as the internals of the database are either inaccessible, or too complex and difficult to change. In addition, this kind of solution can be maintained independently of the database systems. However, the main challenge is to coordinate replica control with concurrency control, as middleware has no access to the information residing on the database kernel.

As with the DBSM approach, this solution also has a global validation step. However, while the former approach performs validation in only one global step, the latter solution needs both local and global steps, because the local one does not consider concurrent remote transactions whose write sets are sent at the same time and by sending only write sets, there is not enough information for replicas to decide whether a transaction commits or aborts. Since total order broadcast is used to order messages, replicas need not notify each other on the transaction outcome because they will all validate a transaction against the same set of write sets and validation is a deterministic process.

2.6.4 Optimistic Active Replication

2.6.4.1 Overview

The Optimistic Active Replication (Felber & Schiper 2001) is another example of the Replicated State Machine approach (Schneider 1993). Unlike the DBSM approach, in which a transaction is executed at the local replica and only at commit time a command is issued to the replicated state machine, in this case the client is separated from the database and sends each

operation request to all replicas using a totally ordered reliable multicast primitive. All replicas handle the request and send back a response to the client.

The key aspect of this approach is that, although each request needs to be sent to the replicas using a Total Order broadcast, the replication algorithm attempts to avoid the full cost of this primitive (in particular, its high latency). Instead, the algorithm is prepared to use a lighter version of Total Order broadcast that can provide an estimate of the final total order using less communication steps, in the absence of failures. Thus, the protocol is based on the assumption that failures are infrequent and is optimized for failure-free runs. Even though its optimism may lead to inconsistencies among replicas and, as a result, the client may receive different replies, the algorithm ensures that the client will never choose an inconsistent reply.

The Total Order broadcast protocol used by this algorithm is a particular case of the Optimistic Total Order primitives that were introduced in Section 2.5. As any Optimistic Total Order primitive, it is able to provide an early estimate of the final total order, which may or may not be the final order of the request. The protocol is based on electing a sequencer. This sequencer is responsible for assigning a sequence number to each message, which corresponds to the optimistic total order. If the sequencer does not fail, every replica will endorse this sequence number, and the order becomes definitive. If the sequencer fails before a majority of replicas has endorsed the optimistic sequence number, a new sequencer is elected and may assign a different sequence number to that message. So, in good runs, every replica gets the optimistic total order in just two communications steps: one required to send the message from the client to the replicas and one required to send the sequence number from the sequencer to the remaining replicas.

Having this particular Optimistic Total Order protocol in mind, the replication algorithm works as follows. Each replica processes the request optimistically as soon as it receives the sequence number from the sequencer and sends the result back to the client. If the client receives a majority of replies that processed the request in the same order, it can assume that the order has become definitive and accept the result.

When the sequencer is suspected of having failed, the remaining replicas have to reach an agreement regarding the order of the requests already processed optimistically and those whose order had not been estimated yet. So, a consensus protocol has too be ran to determine the final order of clients' requests. After that, a new sequencer is chosen and replicas return to the

optimistic mode. If the resulting sequence is different from the optimistic order, a primitive is used to notify the replicas that a message has been optimistically delivered in the wrong order and the effects induced by its processing must be undone.

2.6.4.2 Algorithm

1. Client-side Algorithm

   The client multicast its request to all servers and waits for a quorum of replies. This set of replies is complete when its total weight equals the majority weight. Then, the client chooses the reply with the greatest weight.

2. Server-side Algorithm

   The algorithm proceeds in a sequence of epochs, each divided in two phases, optimistic and conservative, and consists of four tasks.

   - Phase 1 (Optimistic)
     - Task 1a: The sequencer checks for requests received but not yet ordered and orders them in a sequence, which is sent to all replicas.
     - Task 1b: When a sequence is delivered, the replica processes each request in the sequence, generates the reply and sends it back to the client. The weight of the reply consists of the process itself plus the sequencer, unless the process is the sequencer.
     - Task 1c: If a replica suspects the sequencer to have failed, it multicast a message to notify the other replicas to proceed to phase 2.
   - Phase 2 (Conservative)
     - Task 2: When a phase 2 message is delivered, replicas run a consensus protocol to order the requests. If the conservative order is different than the optimistic one, those requests' effects are undone before delivering all replies using a primitive providing total order semantics. After that, the replicas proceed to the next epoch. The new epoch starts with another replica acting as the sequencer.

2.6.4.3 Discussion

A striking difference between this solution and those previously presented is that clients send each operation to all replicas and they all process it. There is no need for any sort of certification test since all replicas execute the same operations in the same order. This contrasts with what was described before, since a whole transaction is executed at a local replica and only at commit time it is broadcast to the remaining replicas; afterwards it is certified and committed if possible. In addition, instead of being passive, clients have an active role in this protocol since they have the responsibility to select the correct reply from the set they receive from all replicas. While the previous solutions rely on Total Order broadcast to ensure consistency among replicas, the Optimistic Active Replication only requires a reliable multicast channel for communication between clients and replicas, a FIFO channel for replicas to exchange messages and a sequencer to order messages sent by the clients. As a consequence, this protocol performs better than the others when the replica acting as the sequencer does not fail, since the overhead of Total Order broadcast is simply the ordering of the requests received from the client, which is done optimistically. However, if the sequencer is suspected of having failed, the remaining replicas have to run a consensus protocol to determine the correct order to process requests, which results in worse performance results when compared to the previously described solutions.

2.7 Bloom Filters

A Bloom filter (Bloom 1970) is a simple space-efficient randomized data structure for representing a set in order to support membership queries (i.e., queries that ask: "*Is element x in set S?*"). Its main drawback is that the test may return true for elements that are not actually in the set (false positives). Nevertheless, the test will never return false for elements that are in the set.

The Bloom filter representation of a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements is described in Broder & Mitzenmacher (2003) as an array of m bits, initially all set to 0. A Bloom filter uses k independent hash functions $h_1, \ldots, h_k$ with range $\{1, \ldots, m\}$. If the set S is not empty, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. A location can be set to 1 multiple times, but only the first change has an effect. In order to check if an item y is in S, all $h_i(y)$ must be set to 1. If so, y is assumed to be in S, with some known probability of error.

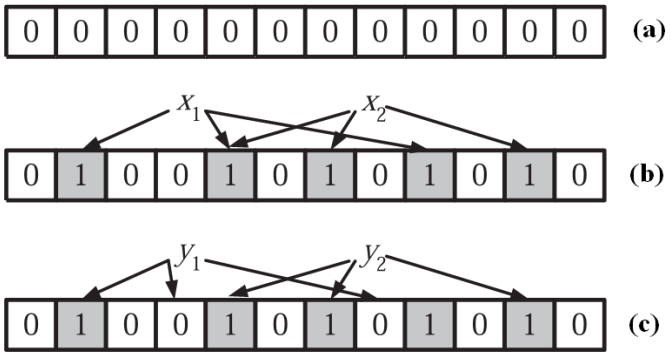


Figure 2.1: Example of a Bloom filter.

Else, y is not a member of S. Figure 2.1 shows how a bloom filter is built and used (Broder & Mitzenmacher 2003). In *(a)*, the filter is an array of 12 bits with all positions set to 0. In *(b)*, it represents the set $S = \{x_1, x_2\}$ with each element hashed three times, and the resulting bits set to 1. To check if elements y_1 and y_2 are in the filter, each must be hashed three times, as shown in *(c)*. Element y_1 is not in S since one of the bits is set to 0, while y_2 is probably in S because all bits are set to 1.

The probability of a false positive f for a single query to a Bloom Filter depends on the number of bits used per item m/n and the number of hash functions k according to the following equation:

$$f = (1 - e^{-kn/m})^k \tag{2.1}$$

where the optimal number k of hash functions that minimizes the false positive probability f given m and n can be shown to be equal to:

$$k = \lceil ln\ 2 \cdot m/n \rceil \tag{2.2}$$

To sum up, the main tradeoffs of using Bloom filters are:

- number of hash functions (k);
- size of the filter (m);
- false positive rate.

In spite of reducing substantially the size of a set, the use of Bloom filters is also associated with some disadvantages that may cause performance to degrade if parameters m and k are not correctly chosen. If the consequences of false positives are too prejudicial for a system's performance, filters need to be larger. Moreover, the number of hash functions is a major influence in the computational overhead.

2.7.1 Applications

Since their introduction, Bloom filters have seen various uses (Broder & Mitzenmacher 2003). Databases use them for differential files, which contain a batch of database records to be updated, and to speed up semi-join operations, by having the databases exchange Bloom filters instead of lists of records, reducing the overall communication between hosts.

There is also a significant range of network applications that rely on Bloom filters to improve their efficiency. In order to reduce message traffic, proxies in distributed caching (Fan, Cao, Almeida, & Broder 2000) do not transfer lists of URL with the exact contents of the cache, but instead periodically broadcast Bloom filters that represent the contents of their cache. Additionally, Bloom filters are also used in resource routing, since they allow probabilistic algorithms for locating resources, and packet routing, as a mean to speed up or simplify protocols. Finally, Bloom filters provide a useful tool for measurement infrastructures used to create data summaries in routers or other network devices.

Summary

This chapter introduced some fundamental concepts which will be central to the discussion in the next chapter. A presentation of the main features of both STM and DSM systems preceded the brief description of three distributed STM systems. Some fundamental concepts of group communication were also described, followed by a short survey on some of the most representative database replication algorithms which rely on the total order broadcast primitive to serialize conflicting transactions. Finally, Bloom Filters were introduced as a technique to reduce the size of the messages exchanged by replicas.

The next chapter introduces D^{2}STM and, most importantly, the distributed certification protocols used to maintain the consistency among replicas.

3 D²STM

3.1 Introduction

This chapter introduces D^{2}STM, a Dependable Distributed Software Transactional Memory that allows programmers to leverage on the computing resources available in a cluster environment, using a conventional STM interface, and transparently ensuring non-blocking and strong consistency guarantees even in the case of failures. A description of the system and its main components can be found in Sections 3.2 to 3.7.

D^{2}STM is built on top of JVSTM, an efficient STM library that supports multi-version concurrency control and, as a result, offers excellent performance for read-only transactions. It takes full advantage of the JVSTM's multi-versioning scheme, sheltering read-only transactions from the possibility of aborts due both to local or remote conflicts. Section 3.5 reports how the existing JVSTM library was integrated in D^{2}STM.

Replica consistency is achieved through a distributed certification process which, in turn, relies on the properties of the Atomic broadcast to serialize conflicting transactions. Two new protocols taking advantage of the Bloom filters' properties to reduce the size of the messages exchanged by replicas, as Section 3.9 presents, were designed. These protocols can be seem as an optimization of existing voting and non-voting protocols used in the context of database replication. The implementation of the four protocols is described in Section 3.8.

3.2 System Model

We consider a classical asynchronous distributed system model (Guerraoui & Rodrigues 2006) 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. We assume that a majority of processes is correct and that the system ensures a sufficient synchrony level (e.g. the availability of a $\Diamond S$

failure detector) to permit the implementation of atomic broadcast primitives. A more detailled description of the communication primitives is provided in Section 3.7.

3.3 System Properties

Recent works (Martin, Blundell, & Lewis 2006; Guerraoui & Kapalka 2008) have started to formalize the guarantees that implementations of transactional memory should provide.

D²STM preserves the weak atomicity (Martin, Blundell, & Lewis 2006) and opacity (Guerraoui & Kapalka 2008) properties of the underlying JVSTM. The former property implies that atomicity is guaranteed only for conflicting pairs of transactional accesses; conflicts between transactional and non-transactional accesses are not protected. Weak atomicity is less composable than strong atomicity (protecting all pairs where at least one is a transactional access). It also raises subtle problems, e.g., granular lost updates. However, the runtime overhead of strong atomicity can be prohibitively high in the absence of hardware support (Martin, Blundell, & Lewis 2006). Opacity, on the other hand, can be informally viewed as an extension of the classical database serializability property with the additional requirement that even non-committed transactions are prevented from accessing inconsistent states.

Finally, concerning the consistency criterion for the state of the replicated (JV)STM instances, D²STM guarantees 1-copy serializability of reads and writes to transactional data (Bernstein, Hadzilacos, & Goodman 1987), which ensures that transaction execution history across the whole set of replicas is equivalent to a serial transaction execution history on a not replicated (JV)STM.

3.4 System Architecture

3.4.1 Overview

The components of a node of the D²STM implementation, depicted in Figure 3.1, are structured into four main logical layers.

The bottom layer is a Group Communication Service (GCS) (Défago, Schiper, & Urbán 2004) which provides two services, as described in Section 3.7. Our implementation uses a generic

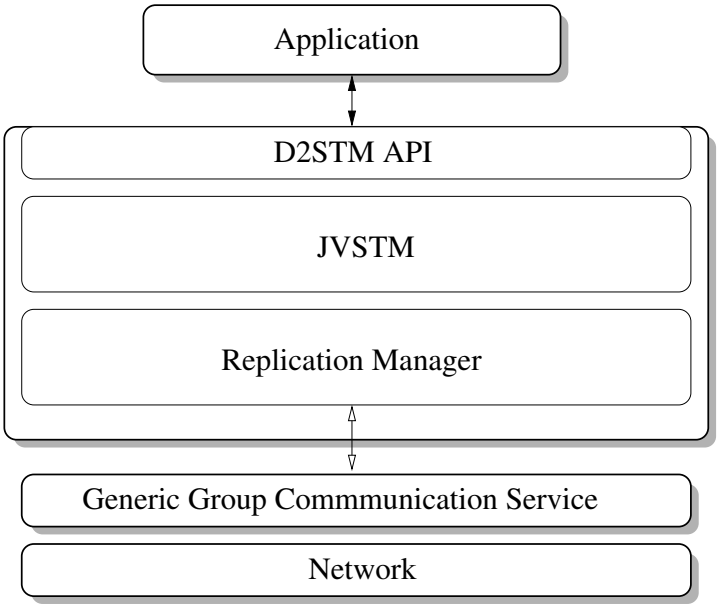


Figure 3.1: Components of a D^{2}STM replica.

GCS interface, called jGCS (Carvalho, Pereira, & Rodrigues 2006), that supports multiple implementations of a GCS. All the experiments described in Section 4 have been performed using the Appia GCS (Miranda, Pinto, & Rodrigues 2001)). Appia implements view-synchronous communication (Birman & van Renesse 1994), which allows to have dynamic membership, change the group view when failures occur, and remove or add new members to a group on the fly. However, this thesis does not address the problem of replica integration, a problem orthogonal to the focus of our study.

The core component of D^{2}STM is represented by the Replication Manager and implements the distributed coordination protocols required for ensuring replica consistency (i.e. 1-copy serializability). It interfaces, on one side, with the GCS layer and, on the other side, with a local instance of a Software Transactional Memory, more precisely JVSTM (Cachopo & Rito-Silva 2006). A detailed discussion of the integration between the replication manager and JVSTM, along with a summary of the most relevant JVSTM internal mechanisms, is provided in Section 3.5. The replica manager is further described in Section 3.6.

Finally, the top layer of D^{2}STM is a wrapper that intercepts the application level calls for transaction demarcation (i.e. to begin, commit or abort transactions), not interfering at all with the application accesses (read/write) to the objects which are managed directly by the underlying JVSTM layer. This approach allows D^{2}STM to transparently extend the classic

Method	Description
`begin()`	Begins a transaction.
`commit(`*Transaction*`)`	Commits a transaction.
`abort(`*Transaction*`)`	Aborts a transaction.
`createVBox()`	Creates a VBox with a unique identifier.
`VBox.get()`	Retrieves a value from the VBox.
`VBox.put(`*object*`)`	Stores a value in the VBox .

Table 3.1: D²STM interface with the application level.

STM programming model, while requiring only minor modifications to pre-existing JVSTM applications.

3.4.2 Interactions Among Components

Each of the four D²STM components presented in Figure 3.1 interact with the neighboring layers through well defined interfaces, as described in this section.

The D²STM API provides the application layer with methods to initialize and finish a transaction, by either committing or aborting it, as shown in Table 3.1. Additionally, methods to create, read and add new values to VBoxes, introduced in Section 2.2.3 and further described in Section 3.5, are also available in this interface. A more detailed description of these last two operations is presented in the next Section. The application level is informed of a transaction's outcome as soon as the validation process ends and the transaction is finished accordingly.

Whenever a transaction is ready to commit, JVSTM notifies the Replication Manager so that the latter can begin the distributed certification process.

On the other hand, the JVSTM interface with the Replication Manager is composed of the methods presented in Table 3.2. These methods mainly consist of retrieving information concerning the transaction's internals, such as its read and write sets, as well as its timestamp, essential to the distributed certification process. Furthermore, the *finishCommit* and the *applyRemoteTransaction* methods allow the Replication Manager to order a local or remote transaction to be committed (or aborted) after determining its outcome. This interface also provides a method to locally validate a transaction using its read set, namely, *validate*. Some of these methods will be further described in Section 3.5.

Finally, the group communication layer provides the Replication Manager the means to

Method	Description
`getReadSet(`*Transaction*`)`	Retrieves the transaction's read set.
`getWriteSet(`*Transaction*`)`	Retrieves the transaction's write set.
`getSnapshotID(`*Transaction*`)`	Retrieves the transaction's timestamp
`applyRemoteTransaction(`*WriteSet*`)`	Writes in memory a remote transaction's write set.
`finishCommit(`*boolean*`)`	Finishes the transaction by either committing (i.e., writing its updates in memory) or aborting it and notifies the application.
`validate(`*Transaction*`)`	Retrieves a boolean indicating the result of the transaction's local validation, which consists of verifying if the boxes read have been updated.

Table 3.2: JVSTM interface with the Replication Manager.

communicate with the other replicas. Its interface consists of methods to create services with user specified guarantees, enabling the Replication Manager to broadcast and receive messages through the corresponding communication channels.

3.5 Integration with JVSTM

JVSTM (Cachopo & Rito-Silva 2006) implements a multi-version scheme which is based on the abstraction of a *versioned box* (VBox) to hold the mutable state of a concurrent program. 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. To this end, JVSTM maintains an 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.

Besides its timestamp, each transaction also stores the set of boxes read and a set with the mapping between VBoxes and the new values written during its execution. A read operation consists of reading objects from VBoxes with version numbers lower or equal to the transaction's number. On the other hand, a write operation does not actually update a VBox, but instead maps the box to the new value.

This means that JVSTM relies on an optimistic approach which buffers transactions' writes and detects conflicts only at commit time, by checking whether any of the VBoxes read by a committing transaction T was updated by some other transaction T' with a larger timestamp value. In this case, T is aborted. Otherwise, T's *commitTimestamp* is increased, its *snapshotID* is set to the new value of *commitTimestamp* and the new values of all the VBoxes it updated are atomically stored within the VBoxes, tagged with the *snapshotID* value. Read-only transactions are never aborted since all the VBoxes read are in the same state as when the transaction started its execution.

To minimize performance overheads, the D^{2}STM's replica coordination protocols are tightly integrated with the JVSTM's transaction timestamping mechanisms.

JVSTM library already provides an interface to begin and commit/abort transactions. However, in order to enable a transaction to be remotely validated, the Replication Manager needs to disseminate information originally only accessible to the replica where it was executed, as well as be able to locally determine if a transaction is valid and commit both local and remote transactions. Consequently, the integration of JVSTM within the D^{2}STM required the implementation of four main (non-intrusive) modifications to JVSTM, extending its original API in order to allow the Replication Manager layer to:

1. extract information concerning internals of the transaction execution, i.e., its read set, write set, and *snapshotID* timestamp. In the remaining, we refer to the methods providing the aforementioned services for a transaction T_x, respectively, as *getReadset(Transaction T_x)*, *getWriteset(Transaction T_x)* and *getSnapshotID(Transaction T_x)*;

2. explicitly trigger the transaction validation procedure (method *validate(Transaction T_x)*), 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;

3. atomically apply, through the *applyRemoteTransaction(Writeset WS)* method, the write set WS of a remotely executed transaction (i.e. atomically updating the VBoxes of the local JVSTM with the new values written by a remote transaction) and simultaneously increasing the JVSTM's *commitTimestamp*;

4. permit cluster wide unique identification of the VBoxes updated by (remote) transactions, as well as of any object, possibly dynamically generated within a (remote) transaction,

whose reference could be stored within a VBox. This is achieved by tagging each JVSTM VBox (and each object, mutable or immutable, assigned to a VBox within a Transaction) 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 and to enable its independent generation at each replica. The current implementation of D^{2}STM relies on a widely recognized international standard, namely the ISO/IEC 11578:1996[1], which uses a 128 bits long encoding scheme[2] that includes the identifier of the generating node and a local timestamp based on a 100-nanosecond intervals.

The integration with JVSTM is not a contribution of this thesis; it has been designed and implemented by the other elements in the Pastramy team. Its details are included in this document for self-containment.

3.6 Replication Manager

The Replication Manager is a central component in D^{2}STM's architecture, in charge of coordinating the distributed replica consistency protocols that ensure replica consistency. When a transaction finishes its local execution, the Replication Manager fetches information from JVSTM in order to build a replica synchronization message. The message is then atomically broadcast to all replicas. Depending on the replication scheme, this message may contain different information.

As it will be seen, in some cases the message includes enough information for all replicas to validate the corresponding transaction, culminating on committing or aborting. In this case, the replica where the transaction has been executed only has to commit/abort the transaction and remote replicas have to apply or discard the write set accordingly. In other replication schemes, only the replica where the transaction has been executed has the required information to decide on the outcome of the transaction. In case, a remote replica needs to wait for a second message containing that transaction's outcome.

Four replica consistency protocols were implemented, namely plain voting and non-voting

[1] Also ITU-T Rec. X.667 - ISO/IEC 9834-8:2005, and integrated within the official Java library since version 1.5.

[2] The standard Leach-Salz variant layout encoding was used.

certification and two versions of Bloom Filter Certification (BFC), one implementing a non-voting algorithm and the other combining both voting and non-voting approaches. Section 3.8 provides a more detailed description of these protocols.

This component is also responsible for managing the timestamps of all the locally active transactions as a part of the distributed garbage collection mechanism, as described in Section 3.10.

3.7 Group Communication

The group communication service is responsible for maintaining up-to-date information regarding the membership of the group of replicas (including failure detection), and providing the required communication support for the coordination among them.

Two services are provided by the GCS component: a totallly ordered reliable broadcast service (ABcast) and a non-ordered reliable broadcast FIFO service (FIFO). The GCS interface consists of four primitives: AB-send(m)/FIFO-send(m) and AB-deliver(m)/FIFO-deliver(m). When a process executes AB-send(m) we say it "ABcasts" m and when a a process executes AB-deliver(m) we say it "ABdelivers" m. Similarly, when a process executes FIFO-send(m) we say it "FIFOcasts" m and when a a process executes FIFO-deliver(m) we say it "FIFOdelivers" m.

The properties of the FIFO service are as follows:

- *Validity:* if a correct process broadcasts a message m, then it eventually delivers m.
- *Uniform Agreement:* if a process delivers m, then all correct processes eventually deliver m.
- *Uniform Integrity:* for any message m, every process delivers m at most once, and only if m was previously broadcast by its sender
- *FIFO order*: If a correct process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.
- *Causal order*: If the broadcast of a message m causally precedes the broadcast of a message m', then no correct process delivers m' unless it has previously delivered m.

In addition to these properties, the ABcast service also has the following property:

- *Uniform Total Order:* if processes p and q both deliver messages m and m', then p delivers m before m' only if q delivers m before m'.

We recall, from Section 2.5.1, that a regular version of these services can be defined, by using the following properties instead of the uniform versions:

- *(Regular) Agreement:* if a *correct* process delivers a message m, then all correct processes eventually deliver m.
- *(Regular) Total Order:* if two *correct* processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

As noted before, all communication services are provided by the Appia system (Miranda, Pinto, & Rodrigues 2001), a layered communication support framework. The Appia system (Miranda, Pinto, & Rodrigues 2001) allows the usage of either the uniform or the regular version of the reliability property in both services (Défago, Schiper, & Urbán 2004). The Replication Manager uses one or more services in each certification protocol, considering the guarantees each type of message exchanged requires from the communication channel. So, the Total Order (or Atomic) broadcast is used by the Replication Manager to disseminate both the read sets (or their Bloom filter representation) and write sets or just the write sets of transactions that are ready to commit, depending on the certification protocol. This way, all replicas validate and commit or abort every transaction in the same order. On the other hand, the non-ordered uniform Reliable broadcast service is used by voting certification protocols to notify the other replicas about the outcome of transactions.

3.8 Replica Consistency Algorithms

In D^{2}STM, replica consistency is achieved through a distributed certification procedure which leverages on the properties of an Atomic Broadcast primitive. Unlike classic eager replication schemes (based on fine-grained distributed locking and atomic commit), that suffer from large communication overheads and fall prey of distributed deadlocks (Gray, Helland, O'Neil,

& Shasha 1996), certification based schemes avoid any onerous replica coordination during the execution phase, running transactions locally in an optimistic fashion.

The consistency of replicas (typically, 1-Copy serializability) is ensured at commit-time, via a distributed certification phase that uses a single Atomic Broadcast to enforce agreement on a common transaction serialization order, avoiding distributed deadlocks, and providing non-blocking guarantees in the presence of (a minority of) replica failures.

Furthermore, unlike classic read-one/write-all approaches that require the full execution of update transactions at all replicas (Bernstein, Hadzilacos, & Goodman 1987), only one replica executes an update transaction, whereas the remaining replicas are only required to validate the transaction and to apply the resulting updates. This allows to achieve high scalability levels even in the presence of write-dominated workloads, as long as the transaction conflict rate remains moderate (Pedone, Guerraoui, & Schiper 2003).

Certification based approaches can be classified into voting and non-voting schemes (Romano, Carvalho, & Rodrigues 2008). In the latter approach, all replicas have enough information about a transaction that allows them to validate it. In voting schemes, on the other hand, only the replica where a transaction was executed can validate it. Therefore, these schemes incur the overhead of an additional message informing the remaining replicas of the transaction's outcome.

This section describes several implementations of these schemes in D^{2}STM, namely plain non-voting and voting certification schemes and two versions of Bloom filter certification (BFC), one implementing the non-voting approach and the final one combining both the voting and the non-voting schemes to achieve a very high compression factor without incurring in equally high transaction abort rates.

3.8.1 Non-Voting Certification Schemes

Non-voting certification schemes require every replica to send both the read and the write sets for each transaction it executes (locally). These sets are broadcast using the total order primitive. Consequently, every site has enough information to validate every transaction. Therefore, since the certification is a deterministic process, transactions can be validated globally without the need for any other messages to be exchanged. If the read set is valid, i.e., the versions of the objects read by the transaction are still up-to-date, the transaction can commit.

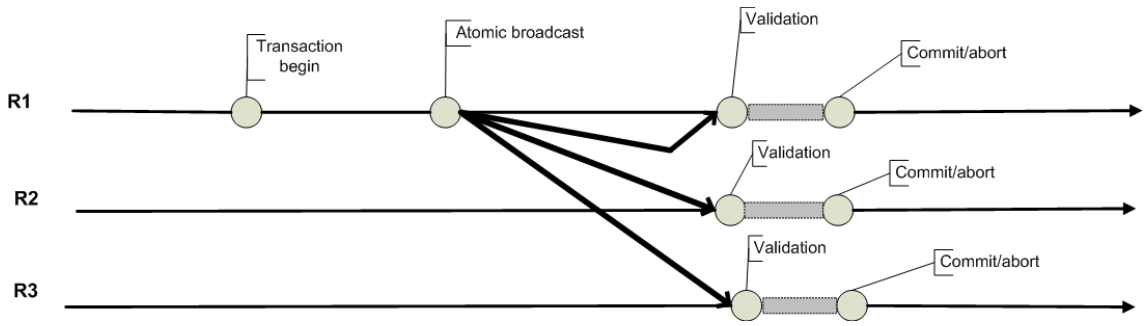


Figure 3.2: Execution of a transaction with a non-voting certification scheme.

Otherwise, it is aborted. This process is illustrated by Figure 3.2, which shows the execution of a transaction in a system with three replicas, from its beginning until it is committed/aborted.

Two of such protocols were implemented: a plain non-voting certification protocol, as described by Pedone, Guerraoui, and Schiper (2003) in the context of database replication, and a more efficient version using Bloom filters to represent the transactions' read set, named Bloom filter certification (BFC).

3.8.1.1 Plain Non-Voting Certification

Figures 3.3 and 3.4 show the plain non-voting certification algorithm pseudo-code, which will now be described in detail. The procedures to commit a transaction are depicted in Figure 3.3. Read-only transactions are executed locally, and committed without incurring in any additional overhead (lines 8-10). 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 (due to both local or remote conflicts).

A committing transaction with a non-null write set (i.e. it has updated some VBox), is first locally validated to detect any local conflicts (lines 12-14). 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 ABcasts the read set(i.e., the set of identifiers of all the VBoxes read by the transaction) along with the transaction's write set and its snapshotID (line 15). In addition, the message exchanged also contains the timestamp of the oldest transaction currently active in that replica. The transaction's timestamp (snapshotID) is necessary for the validation procedure, while the timestamp of the oldest active transaction is used in the distributed garbage collecting mechanism (described

```
1: Set ActiveXacts, CommittedXacts, AbortedXacts=∅;

2: Transaction begin()
3:   Transaction T_x=JVSTM.begin();
4:   ActiveXacts=ActiveXacts∪T_x;
5:   return T_x;

6: boolean commit(Transaction T_x)
7:   // Read-only transactions are processed locally
8:   if (getWriteset(T_x)=∅)
9:       ActiveXacts=ActiveXacts\T_x;
10:      return true;
11:  // Update transactions are first locally validated
12:  if (¬validate(T_x))
13:      ActiveXacts=ActiveXacts\T_x;
14:      return false;
15:  AB-send[T_x, getSnapshotID(T_x), getReadset(T_x), getWriteset(T_x), min_{T_y∈ActiveXacts}( getSnapshotID(T_y) ) ];
16:  // The xact's outcome is determined upon AB-delivery
17:  wait T_x ∈ ( AbortedXacts ∪ CommittedXacts )
18:  ActiveXacts=ActiveXacts\T_x;
19:  if (T_x ∈ AbortedXacts)
20:      AbortedXacts=AbortedXacts\T_x;
21:      return false;
22:  else return true;
```

Figure 3.3: Pseudo-code of the plain non-voting certification algorithm, part 1: commit.

in Section 3.10).

As in classical non-voting certification protocols, update transactions are validated upon their AB-delivery (Figure 3.4). At this stage, it is checked whether T_x's read set contains any item updated by transactions with a *snapshotID* timestamp larger than that of T_x's (lines 25-27). If no match is found, then T_x can be safely committed (lines 32-36), otherwise it is aborted (lines 29-30). Committing a transaction T_x consists of the following steps. If T_x is a local transaction, it just suffices to request the local JVSTM to commit it. If, on the other hand, T_x is a remote transaction, its write set is atomically applied using the *applyRemoteTransaction(*WS_{T_x}*)* method.

The garbage collection of the set containing the UIDs of the objects updated by committed transactions, described in Section 3.10, is performed after the validation phase (line 37).

In the validation process described above, T_x's read set is checked against the write sets of the transactions committed during its execution to determine if an object read during its execution was modified by another update transaction. Instead, an early approach to this process consisted of checking whether each read set entry's version number had been updated

```
23: upon AB-deliver[Transaction T_x, int snapshotID, ReadSet RS, WriteSet WS, int oldestActiveXact] from p_j do
24:  // Validate Transaction
25:  ∀T_y ∈ CommittedXacts s.t. getSnapshotID(T_k)>snapshotID do
26:      ∀ < UID, · >∈ getWriteset(T_y) do
27:          if (RS.contains(UID))
28:              // Xact failed validation and is aborted
29:              AbortedXacts=AbortedXacts∪{T_x}
30:              return;
31:  // Xact passed validation: commit xact
32:  CommittedXacts=CommittedXacts∪T_x;
33:  if (isLocal(T_x))
34:      JVSTM.commit(T_x);
35:  else
36:      applyRemoteTransactionWS(WS);
37:  GCCommittedXacts(oldestActiveXact, j); //as described in Section 3.10
```

Figure 3.4: Pseudo-code of the plain non-voting certification algorithm, part 2: validation.

to a more recent value than the transaction's snapshotID. If this condition was verified, the object had been modified by an already committed concurrent transaction and the validating transaction was to be aborted.

Since the value of each read set entry's version number is read directly from memory, additional information on past transactions is not necessary and, therefore, the distributed garbage collection scheme is irrelevant in this approach.

However, this solution proved to be less efficient in workloads dominated by read sets significantly larger than write sets, which are typical of common applications (Carvalho, Cachopo, Rodrigues, & Rito-Silva 2008). This can be explained by the fact that, even in situations where the probability of concurrent transactions is high, a transaction's read set size can still be larger than already committed concurrent transactions' write sets. Moreover, since the performance this scheme is to be compared with the BFC's, both validation processes should be as close as possible so that the differences that may occur are due to message compression and not other factors.

3.8.1.2 Non-Voting Bloom Filter Certification

Bloom filters, introduced in Section 2.7, are data structures that support membership queries and may be used to reduce the size of the messages exchanged during the distributed certification phase of a transaction at the cost of a user-tunable increase in the abort rate. Read sets do

```
1: Set ActiveXacts, CommittedXacts, AbortedXacts=∅;
2: int avgBFQueries=initialAvgBFQueries;

3: Transaction begin()
4:   Transaction T_x=JVSTM.begin();
5:   ActiveXacts=ActiveXacts∪T_x;
6:   return T_x;

7: boolean commit(Transaction T_x)
8:   // Read-only transactions are processed locally
9:   if (getWriteset(T_x)=∅)
10:        ActiveXacts=ActiveXacts\T_x;
11:        return true;
12:  // Update transactions are first locally validated
13:  if (¬validate(T_x))
14:        ActiveXacts=ActiveXacts\T_x;
15:        return false;
16:  int BFSize=estimateBFSize(avgBFQueries);
17:  BloomFilter BF=new BloomFilter(BFSize);
18:  ∀UID ∈ getReadset(T_x) BF.add(UID);
19:  AB-send[T_x, getSnapshotID(T_x), BF, getWriteset(T_x), min_{T_y∈ActiveXacts}( getSnapshotID(T_y) ) ];
20:  // The xact's outcome is determined upon AB-delivery
21:  wait T_x ∈ ( AbortedXacts ∪ CommittedXacts )
22:  ActiveXacts=ActiveXacts\T_x;
23:  if (T_x ∈ AbortedXacts)
24:        AbortedXacts=AbortedXacts\T_x;
25:        return false;
26:  else return true;
```

Figure 3.5: Pseudo-code of the (non-voting) Bloom filter certification algorithm, part 1: commit

not need to convey data values, only the UIDs of the objects read need to be exchanged, which makes them appropriate to be encoded by a Bloom filter. Therefore, the BFC leverages on this technique to reduce the size of the messages exchanged between the replicas by replacing the transactions' read set with a Bloom filter, which is also used to validate the transaction.

The BFC algorithm, presented in Figures 3.5 and 3.6, is very similar to the plain non-voting certification, apart from three modifications highlighted in the pseudo-code. Firstly, if a transaction passes the local validation phase, its read set is encoded in a Bloom filter, which replaces it in the ABcast the Replication Manager sends to all replicas (Figure 3.5, lines 16-19). The second modification is that, instead of checking the transaction's read set against the write sets of concurrent transactions previously committed to validate it, this verification is done analogously using the Bloom filter (Figure 3.6, lines 38-46). Finally, in order to correctly estimate the size of a Bloom filter, replicas keep track of the number of queries performed

```
27: upon AB-deliver[Transaction T_x, int snapshotID, BloomFilter BF, WriteSet WS, int oldestActiveXact] from p_j do
28:  if(¬ValidateUsingBF(T.snapshotID, T.Bloomfilter))
29:      AbortedXacts=AbortedXacts∪{T_x}
30:      return;
31:  // Xact passed validation: commit xact
32:  CommittedXacts=CommittedXacts∪T_x;
33:  if (isLocal(T_x))
34:      JVSTM.commit(T_x);
35:  else
36:      applyRemoteTransactionWS(WS);
37:  GCCommittedXacts(oldestActiveXact, j); //as described in Section 3.10

38:  boolean ValidateXactUsingBF(int snapshotID, BloomFilter BF) //validate transaction based on bloom filter
39:   int BFQueries=0;
40:   ∀T_y ∈ CommittedXacts s.t. getSnapshotID(T_y)>snapshotID do
41:       ∀ < UID, · >∈ getWriteset(T_y) do
42:           BFQueries++;
43:           if (BF.contains(UID)))
44:               return false; //the transaction failed validation
45:   avgBFQueries=updateAvg(BFQueries,recComXacts);
46:   return true ;
```

Figure 3.6: Pseudo-code of the (non-voting) Bloom filter certification algorithm, part 2: validation.

against each BF during the validation phase (Figure 3.6, lines 42 and 45). 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, which we denote as *maxAbortRate*. The logic for sizing of the Bloom Filter is encapsulated by the *estimateBFSize()* primitive (Figure 3.5, line 16), and will be detailed in Section 3.9.1.

The messages exchanged between the replicas consist of the transaction's id, its timestamp (snapshotID), write set, the Bloom filter representing its read set, the number of hash functions used to encode each element in the filter (as explained in Section 3.9) and the timestamp of the oldest transaction currently active in that replica(Figure 3.5, line 19).

The correctness of the BFC scheme can be (informally) proved by observing that i) replicas validate all write transactions in the same order (the one determined by the AB-deliver primitive), and that, ii) the validation procedure, despite being subject to false positives, is deterministic given that all replicas rely on the same set of hash functions to query for the presence/determine the encoding of data items in the Bloom filter. Hence, as already highlighted,

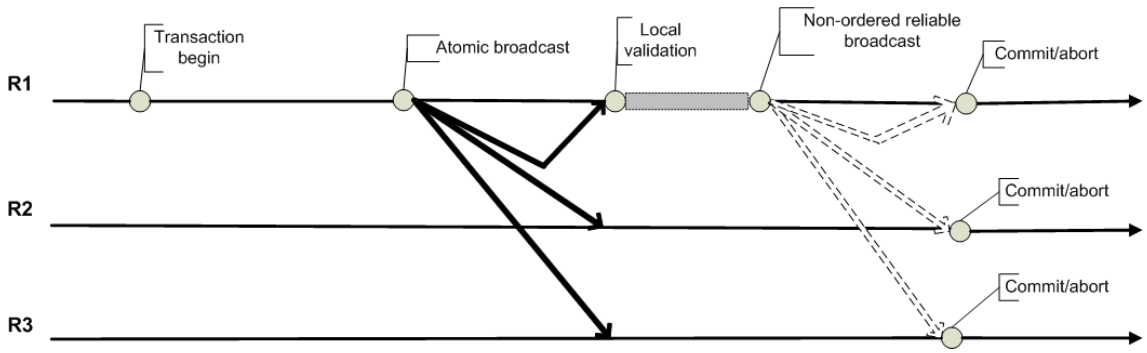


Figure 3.7: Execution of a transaction with a voting certification scheme.

the occurrence of false positives results in an increase of the transaction abort rate, but can never lead to inconsistencies of the replicas' states.

3.8.2 Voting Certification Schemes

In Voting Certification Protocols, only the write set is broadcast to the other replicas. This means that only the replica where the transaction was executed has all the data needed to determine its outcome. If the local replica detects a conflict, it broadcast a message warning the other replicas to discard the write set. Else, the broadcast message will inform the other replicas they can apply the write set. These notification messages are not required to be broadcast using the total order primitive. This process is depicted in Figure 3.7, which illustrates the execution of a transaction in a system with three replicas.

The implemented protocol described next was first introduced in Kemme & Alonso (2000) in the context of database replication, which is followed by a different version of the BFC protocol presented earlier and combines both voting and non-voting approaches to achieve a higher compression factor while avoiding false positives.

3.8.2.1 Plain Voting Certification

Figures 3.8 and 3.9 depict the pseudo-code of the plain voting certification protocol implemented in D²STM. The most striking difference between this algorithm and those presented previously is that a transaction is only validated at the replica where it was executed, which means that the remaining replicas simply have to apply its write set. So, replicas do not keep information about previously committed transactions and, consequently, there is no need for a

```
1: Set AbortedXacts, CommittedXacts=∅;
2: Queue OrderedXacts, UnorderedXacts=∅; //contain the triple <Transaction, WriteSet, boolean>

3: Transaction begin()
4:   Transaction T_x=JVSTM.begin();
5:   return T_x;

6: boolean commit(Transaction T_x)
7:   // Read-only transactions are processed locally
8:   if (getWriteset(T_x)=∅)
9:        ActiveXacts=ActiveXacts\T_x;
10:       return true;
11:  // Update transactions are first locally validated
12:  if (¬validate(T_x))
13:       return false;
14:  AB-send[T_x, getWriteset(T_x)) ];
15:  wait T_x ∈ ( AbortedXacts ∪ CommittedXacts )
16:  ActiveXacts=ActiveXacts\T_x;
17:  if (T_x ∈ AbortedXacts)
18:       AbortedXacts=AbortedXacts\T_x;
19:       return false;
20:  CommittedXacts=CommittedXacts\T_x;
21:  return true;
```

Figure 3.8: Pseudo-code of the plain voting certification algorithm, part 1: commit.

distributed garbage collecting scheme. On the other hand, instead of a single message conveying the transaction's read and write sets, this approach requires replicas to exchange two messages in order to complete the transaction's validation process, one with the transaction's id and write set and the other with the transaction's id and the result of the local validation, which may introduce an additional overhead in the distributed certification process.

Each replica keeps two queues (Figure 3.8, line 2): one for unordered transactions (*UnorderedTxs*), which stores the decision messages delivered by the FIFO service, more precisely a transaction's identifier and a boolean indicating its outcome, and another for ordered transactions (*OrderedTxs*), which contains transactions delivered by the ABcast service. Each entry in the latter queue is composed of a transaction's id, its write set and also a boolean indicating its outcome (this last information is delivered through the FIFO service, as mentioned before). A transaction is only committed if it is in the *OrderedTxs* queue and the replica has all the data it needs to commit (or abort) it (which, corresponds to the write set and the result of the local validation). This way all replicas commit (or abort) all transactions in the same sequence, since they all add transactions to the *OrderedTxs* queue in the same order.

```
22: upon AB-deliver[Transaction T_x, WriteSet WS] from p_j do
23:  if(T_x ∈ UnorderedXacts)
24:       boolean result = getResultFromSet(T_x, UnorderedTransactions);
25:       enqueue(OrderedXacts, < T_x, WS, result >);
26:       remove(UnorderedXacts, T_x);
27:  else enqueue(OrderedXacts, < T_x, WS, null >);
28:  ProcessOrderedXact( );

29: upon FIFO-deliver[Transaction T_x, boolean result] from p_j do
30:  if(T_x ∈ OrderedXacts)
31:       setResultInSet(T_x, OrderedXacts, result);
32:  else enqueue(UnorderedXacts, < T_x, null, result >);
33:  ProcessOrderedXact( );

34: void ProcessOrderedXact( )
35:  if(size(OrderedXacts)<1)
36:       return;
37:  <Transaction, WriteSet, boolean> T = peek(OrderedXacts); //Retrieves but does not remove first item from the queue
38:  if(¬isLocal(T.Transaction) ∧¬hasWriteSetAndResult(T))
39:       return;
40:  if(hasWriteSetAndResult(T))
41:       if(T.Result)
42:            if(isLocal(T.Transaction))
43:                 JVSTM.commit(T.Transaction);
44:            else applyRemoteTransactionWS(T.WriteSet);
45:            CommittedXacts=CommittedXacts∪{T_x};
46:       else AbortedXacts=AbortedXacts∪{T_x};
47:       dequeue(OrderedXacts);
48:       return;
49:  if(isLocal(T.Transaction))
50:       boolean result = validate(T.Transaction); //local validation
51:       FIFO-send[T.Transaction, result];
```

Figure 3.9: Pseudo-code of the plain voting certification algorithm, part 2: message delivery and validation.

The first part of the algorithm is similar to the already presented non-voting schemes, diverging only in the validation procedure. When an update transaction T_x finishes its local execution and passes the local validation phase, the Replication Manager AB-casts only its write set (Figure 3.8, line 14). Upon receiving this message (Figure 3.9, lines 22-28), T_x is added to the *OrderedTxs* queue. Additionally, if it was already in the *UnorderedTxs* queue, the corresponding result is also associated with its entry in the *OrderedTxs* queue. When a message is received from the FIFO service (Figure 3.9, lines 29-35), it is added to the *UnorderedTxs* queue, unless that transaction's write set had already been delivered through an AB-cast. In that case, the result is associated with the corresponding transaction's entry in the *OrderedTxs* queue.

Whenever a message arrives, whether through the ABCast service or the FIFO service, the

```
1: Set ActiveXacts, CommittedXacts, AbortedXacts, WaitingForResultXacts=∅;
2: int avgBFQueries=initialAvgBFQueries;
3: Queue OrderedXacts, UnorderedXacts=∅; //contain the structure <Transaction, BloomFilter, WriteSet, int, boolean>
4: Set GCInfo = ∅; //contains the structure <Transaction, int, int>

5: upon AB-deliver[Transaction T_x, int snapshotID, BloomFilter BF, WriteSet WS, int oldestActiveXact] from p_j do
6:   if(T_x ∈ UnorderedXacts)
7:       boolean result = getResult(T_x, UnorderedTransactions);
8:       enqueue(OrderedXacts, < T_x, BE, WS, snapshotID, result >);
9:       remove(UnorderedXacts, T_x);
10:  else enqueue(OrderedXacts, < T_x, BF, WS, snapshotID, null >);
11:  GCInfo = GCInfo∪ < T_x, oldestActiveXact, j>;
12:  ProcessOrderedXact( );

13: upon FIFO-deliver[Transaction T_x, boolean result] from p_j do
14:  if(T_x ∈ OrderedXacts)
15:      setResult(T_x, OrderedXacts, result);
16:  else enqueue(UnorderedXacts, < T_x, null, null, null, result >);
17:  WaitingForResultXact=WaitingForResultXact\T_x;
18:  ProcessOrderedXact( );
```

Figure 3.10: Pseudo-code of the (voting and non-voting) Bloom filter certification algorithm, part 1: message delivery.

first transaction in the *OrderedTxs* queue is processed if it is either local (i.e., was executed in that replica) or both its write set and the corresponding result have already been delivered (Figure 3.9, lines 39-42). In the first situation, the replica locally validates the transaction and broadcasts a message with the result of this validation, using the FIFO service (Figure 3.9, lines 49-51). In the second situation, the replica has already received the write set and the result of that transaction's validation and will either commit or abort it, if it is the local replica, otherwise it will apply or discard the transaction's write set (Figure 3.9, lines 40-48).

3.8.2.2 Voting Bloom Filter Certification

This version of the BFC protocol combines both the voting and the non-voting certification approaches so that a higher message compression factor can be used in the Bloom filters while avoiding an equal increase in the transaction abort rate. The compression achieved by the non-voting BFC is limited to the *maxAbortRate* pre-defined by the user and higher rates may not be tolerated by abort-sensitive applications. The main idea regarding the combination of the two certification approaches is to achieve higher compression rates without incurring the risk of obtaining an excessive number of aborted transactions.

```
19: void ProcessOrderedXact[]
20:  if(size(OrderedXacts)<1 ∨ peek(OrderedXacts).Transaction ∈ WaitingForResultXacts)
21:       return;
22:  <Transaction, BloomFilter, WriteSet, int, boolean> T = peek(OrderedXacts); //First item is not removed
23:  if(hasResult(T))
24:       if(T.Result)
25:            if(isLocal(T.Transaction))
26:                 JVSTM.commit(T.Transaction);
27:            else applyRemoteTransactionWS(T.WriteSet);
28:            CommittedXacts=CommittedXacts∪{T_x};
29:       else AbortedXacts=AbortedXacts∪{T_x};
30:       GCXacts(T.Transaction);
31:  else //validate Xact using its Bloom filter
32:       if(¬ValidateUsingBF(T.snapshotID, T.Bloomfilter))
33:            if(isLocal(T.Transaction))
34:                 boolean result = validate(T.Transaction); //local validation
35:                 FIFO-send[T.Transaction, result]; //local replica broadcasts the result of validation with the read set
36:            WaitingForResultXacts=WaitingForResultXact∪{T.Transaction}; //the replicas wait for the result
37:            return;
38:       //Xact passed the validation and can commit
39:       if (isLocal(T.Transaction))
40:            JVSTM.commit(T.Transaction);
41:       else
42:            applyRemoteTransactionWS(T.WS);
43:       CommittedXacts=CommittedXacts∪{T.Transaction};
44:       GCXacts(T.Transaction);

45: void GCXacts[Transaction T]
46:  <Transaction, int, int> info = get(GCInfo, T);
47:  GCCommittedXacts(info.oldestActiveXact, info.j); //as described in Section 3.10
48:  GCInfo = GCInfo \ info;
49:  dequeue(OrderedXacts);
```

Figure 3.11: Pseudo-code of the (voting and non-voting) Bloom filter certification algorithm, part 2: validation.

All replicas first validate a transaction using the bloom filter representing its read set. If the validation fails (most likely due to a false positive), the replica where it was executed uses its local data to determine the transaction's outcome and sends this information to the remaining replicas, which then apply or discard the transaction's write set. Otherwise, the transaction is committed (or its write set is applied) without the need for exchanging additional messages. Figures 3.10 and 3.11 show the pseudo-code for this algorithm, highlighting the most significant differences between it and the one presented in Section 3.8.2.1.

This algorithm differs from the plain voting certification in three main aspects. Firstly, the ABcast message contains a bloom filter encoding the transaction's read set, as well as its snap-

shotID and write set (and also the timestamp of the oldest active transaction in that replica), so all replicas have enough information to validate all transactions, as shown in Figure 3.5, lines 16-19. This brings us to the second difference, which consists of the fact that, instead of waiting for the local replica to disclose the outcome of a transaction, all replicas first validate all transactions using the Bloom filter, as already described in the first version of BFC (Section 3.8.1.2) and depicted in Figure 3.11 in line 32. If the transaction is successfully validated, it is committed (or, in remote replicas, its write set is applied) (Figure 3.11, lines 39-44). Else, they wait for the replica where the transaction was executed to broadcast a message containing a boolean indicating the outcome of the transaction through the FIFO service (Figure 3.11, lines 35 and 36). This replica uses the transaction's read set for its validation in order to avoid the Bloom filter's false positives (Figure 3.11, line 34). The third and final difference is that, since replicas use data from transactions committed in the past in the validation process, there is the need for the distributed garbage collection mechanism described in Section 3.10 (Figure 3.11, lines 30, 44 and 45-49).

Analogously to the plain voting certification, transactions are processed in the order they were added to the *OrderedXacts* queue, which now stores a bloom filter and the transaction's snapshotID, in addition to its write set (Figure 3.10, line 3). Processing a new transaction starts only when the previous one is either committed or aborted in that replica.

Even though this algorithm in some cases needs an additional message when compared to the non-voting BFC, the transaction abort rate is not affected by Bloom filter's false positives since transactions that fail the first validation have the process repeated but using their actual read set rather than the filter that represents it.

Figures 3.2 and 3.7 illustrate this algorithm in a system with three replicas. The first shows the case when the remote validation (using the bloom filter) indicates the transaction should commit, while the second corresponds to the situation when the transaction does not pass the remote validation (probably due to a false positive) and the local replica needs to send a second message with the correct outcome.

3.9 Bloom Filters: Management and Implementation

As already discussed in Section 3.8.1.2 and Section 3.8.2.2, when describing, respectively the non-voting and voting BFC schemes, the occurrence of a false positive during the querying of the Bloom Filters in the validation phase can negatively impact the performance of the two protocols. In detail, a false positive indication from a Bloom Filter can induce the unnecessary abort of a transaction in the non-voting BFC scheme, or require the execution of an additional voting phase in the voting BFC scheme. Therefore, in order to maximize the protocols' performances, it is essential to be able to exert control over the probability of incurring in false positives during the validation phase.

In addition, as building the filter introduces an additional computational overhead associated with its creation and decoding heavily influenced by the hash function chosen, it is necessary to select algorithms that are fast but, at the same time, provide a low collision rate.

A description of how these two factors are taken into account in D^{2}STM is presented in this subsection.

3.9.1 Controlling the False Positive Rate

The probability of incurring in a false positive during the validation phase of a transaction T_x depends on the number of queries that will be performed on its Bloom Filter, which is equal to the number of items updated by transactions concurrent with T_x. The first implemented solution to build Bloom filters did not consider the fact that, when validating a transaction, the Bloom filter's false positive rate varies according to the number of distinct queries performed against it. The filters were built with a fixed number of bits per element and hash functions, estimated based only on the equations 2.1 and 2.2.

The results of this approach were, however, very unpredictable (usually a higher abort rate than the one previously determined), which eventually lead us to consider a more dynamic approach to build the filters depending on the size of the concurrent transactions' write sets.

Determining the size of the Bloom Filter for a committing transactions, so to guarantee that a target *maxAbortRate* is never exceeded, would require to know *exactly* the number q of distinct queries that will have to be performed against the Bloom Filter during the transaction

validation procedure (i.e. once it is ABdelivered). However, at the time in which T_x enters the commit phase, it is not possible to exactly foresee neither how many transactions will commit before T_x is ABdelivered, nor what will be the size of the write sets of each of these transactions. On the other hand, any error in estimating q does not compromise safety, but may only lead to (positive or negative) deviations from the target *maxAbortRate* threshold.

The BFC uses a simple and lightweight heuristic, which exploits the fact that each replica can keep track of the number of queries performed to the Bloom filter of any locally ABdelivered transaction. In detail, we rely on the moving average across the number of queries performed during the validation phase of the last *recComXacts* transactions as an estimator of q (as seen in Figure 3.6). Once q is estimated, we can then determine the number m of bits in the Bloom Filter by considering that the false positives for any distinct query are independent and identically distributed events which generate a Bernoullian process (Bertsekas & Tsitsiklis 2002). At the light of this observation, the probability of aborting a transaction because of a false positive in the Bloom Filter-based validation procedure, *maxAbortRate*, can be expressed as:

$$maxAbortRate = 1 - (1 - f)^q$$

which, combined with Equations 2.1 and 2.2, allows us to estimate m as:

$$m = \left\lceil - n \, \frac{log_2(1 - (1 - maxAbortRate)^{\frac{1}{q}})}{ln \; 2} \right\rceil$$

Note that, when implementing the bit array using the *BitSet* class of the Java library, the actual size of the Bloom filter may not always exactly correspond to the estimated number of bits per element multiplied by the number of elements in the set (n). This is due to the fact that the array is built with blocks of 64 bits. Consequently, filters can, in some occasions, be larger than m.

The reduction of the amount of information exchanged, achievable by the BFC scheme, is clearly highlighted by the graph in Figure 3.12, which shows the BFC's compression factor (defined as the ratio between the number of bits for encoding a transaction's read set with the ISO/IEC 11578:1996 standard UID encoding, and with BFC) as a function of the target *maxAbortRate* parameter and of the number q of queries performed during the validation phase.

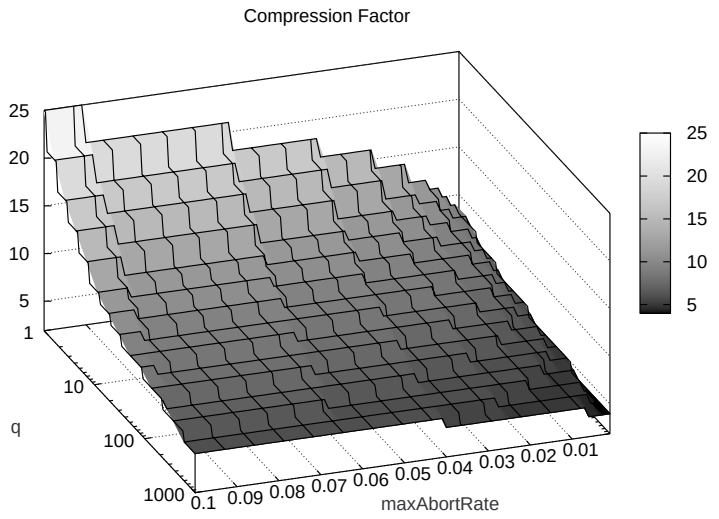


Figure 3.12: Compression Factor achieved by BFC.

The plotted data shows that, even for marginal increases of the transaction abort probability in the range of [1%-2%], BFC achieves a [5x-12x] compression factor, and that the compression factor extends up to 25x in the case of 10% probability of transaction aborts induced by a false positive of the Bloom Filter.

Seeing that the number of hash functions k used in the filter depends directly on the number of bits per element (2.2) which, in turn, is determined when the filter is built, the messages exchanged for replica synchronization purposes need to convey k along with the filter. This way, all replicas have all the information necessary to query the filter with local data.

3.9.2 Hash Functions

In order to speed up the Bloom Filter construction (more precisely the insertion of items within the Bloom Filter), the BFC exploits a recently proposed optimization by Kirsch & Mitzenmacher (2006) which generates the $k = \lceil ln\ 2 \cdot m/n \rceil$ hash values required for encoding a data item within the Bloom Filter via a plain (and very efficient) linear combination of the output of only two independent hash functions. In detail, two hash functions $h_1(x)$ and $h_2(x)$ are used to simulate additional hash functions of the form $g_i(x) = h_1(x) + ih_2(x)$, $1 \leq i \leq k$, where k is the number of hash functions needed to represent an element in the filter, as illustrated by the pseudo-code in in Figure 3.13.

```
int[ ] determineFilterPositions[UID objectID, int seed, int filterSize]
    int filterPositions[k]={0,. . .,0};
    int hash1=hash(objectID, seed);
    int hash2=hash(objectID, hash1);

    for ( i = 0; i < k; i = i + 1)
        filterPositions[i] = |(hash1 + i∗hash2) % filterSize|;

    return filterPositions;
```

Figure 3.13: Pseudo-code of the algorithm used to determine the positions of an item in a Bloom filter.

Many hashing algorithms have been proposed to be used in this context. One of the most popular examples is MD5[3] (Broder & Mitzenmacher 2003; Mitzenmacher 2002; Fan, Cao, Almeida, & Broder 2000). However, this hashing algorithm proved to be very inefficient due to its focus on security.

The hashing algorithm that showed the best performance while matching the analytically forecast false positive probability turned out to be MurmurHash2[4], a simple, multiplicative hash function whose excellent performances have been also confirmed by some recent benchmarking results[5].

3.10 Distributed Garbage Collection

As already described, the validation phase of a transaction requires the availability of the write sets of committed concurrent transactions to check if the validating transaction has read any objects updated since it started its execution. To this end, the Replication Manager locally buffers the UIDs of the VBoxes updated by any committed transaction in the *CommittedXacts* set immediately after the validation phase. However, this data structure will very rapidly reach a considerable size and eventually store unnecessary information, since a transaction can only be invalidated by those committed during its execution period.

In order to avoid an unbounded growth of the *CommittedXacts* set, we rely on a distributed garbage collection scheme (analogous to the one employed by Perez-Sorrosal et al. (2007)), in

[3] `http://www.ietf.org/rfc/rfc1321.txt?number=1321`
[4] `http://murmurhash.googlepages.com`
[5] `http://www.strchr.com/hash_functions`

```
int oldestActiveXacts[n]={0,...,0};
Set CommittedXacts=∅;

void GCCommittedXacts[int oldestActiveXact, int replicaId]
    int oldGlobalOldestXact= min_{i∈[1,n]}(oldestActiveXacts[i]);
    if (oldestActiveXact<oldestActiveXacts[replicaId])
        oldestActiveXacts[replicaId]=oldestActiveXact;

        if (oldGlobalOldestXact != min_{i∈[1,n]}(oldestActiveXacts[i]))
            ∀T_k ∈ CommittedXacts s.t. getSnapshotID(T_k)≤ min_{i∈[1,n]}(oldestActiveXacts[i]) do
                CommittedXact=CommittedXact\T_k;
```

Figure 3.14: Pseudo-code of the Distributed Garbage Collection algorithm.

which each replica exchanges (as a piggyback to the AB-casted transaction validation message) the minimum *snapshotID* of all the locally active transactions. This way, all replicas are aware of the oldest active transaction in every replica, which enables them to determine the oldest timestamp among those of all transactions currently active in the system. This information is used to garbage collect the *CommitXacts* set by removing the data associated with any committed update transactions whose execution is no longer relevant to the validation process. In other words, replicas only keep the UIDs of objects belonging to write sets of committed update transactions that can still invalidate currently active transactions.

Figure 3.14 depicts the pseudo-code for distributed garbage collection operation. All replicas store an array with the timestamp of the oldest transaction running in each replica. After a transaction's validation process ends, the array entry corresponding to the replica where this transaction was originally executed is updated with the value indicated in the corresponding replica synchronization message. If, as a consequence, the global oldest transaction timestamp changed, the information belonging to transactions with a timestamp smaller than this value is discarded.

3.11 Failure Handling

Even though this is not a central aspect of this thesis, for self-containment purposes it is import to describe how the system behaves in the presence of a fault from the replica consistency perspective. Three different situations can occur, but what they all have in common is that either all replicas certify a transaction or none does.

Two cases are common to all protocols, differing in the moment a replica fails. The first case is when the local replica fails *before* the ABcast of a transaction's information. In this case, since no information is exchanged, the other replicas will never be aware this transaction and, hence, the certification procedure for transactions executing at the replica when it fails will never be triggered in the remaining replicas. On the other hand, should a replica fail *during* the ABcast, the Atomic broadcast properties ensure that either all replicas receive this message or none does. Besides, the uniformity property guarantees that, if one replica receives the ABcast, then all replicas will.

In the voting protocols, which require two messages to complete the validation phase, the non-failing replicas need to discard any transaction information received if the local replica fails between sending the ABcast and the vote. Once more, the view synchrony property ensures that either all replicas receive the vote before receiving the new view or none does.

Summary

This chapter presented the main components of D^{2}STM's architecture, with the focus being on the Replication Manager and the Distributed Coordination Protocols. It started with an introduction of the system model and the components of a D^{2}STM node. The integration of the existing JVSTM library within the D^{2}STM was also described. The replica consistency algorithms, namely the plain non-voting and voting certification and the two versions of BFC, which rely on Atomic Broadcast to serialize conflicting transactions, were introduced. This was followed by a description of how two of the most important trade-offs in building Bloom filters, specifically its size and hashing algorithm, were determined so that the user-tunable false positive rate would match the actual results.

In the following chapter, the certification schemes presented are evaluated using both synthetic workloads and more realistic and complex benchmarks.

4 Evaluation

4.1 Introduction

This chapter reports the results of an experimental study aimed at evaluating and comparing the performance of the protocols presented in the previous chapter in a real distributed STM system, namely using the D^{2}STM prototype, in face of a variety of synthetic workloads with different levels of complexity. It begins by describing the experimental settings and the criteria used in the evaluation. Then, Section 4.4 describes the results obtained with two micro-benchmarks, Bank and Red Black Tree benchmark. The STMBench7 benchmark and its results are presented in Section 4.5. Finally, Section 4.6 presents a brief discussion of the results obtained.

4.2 Experimental Settings

All the experiments presented here were performed in a cluster of 8 nodes, each one equipped with an Intel QuadCore Q6600 at 2.40GHz with 8 GB of RAM running Linux 2.6.27.7. The nodes are interconnected via a private Gigabit Ethernet.

The replica consistency protocols, described in the previous chapter, were implemented in the D^{2}STM prototype, also introduced in the same chapter. The Atomic Broadcast implementation used is based on a classic sequencer-based algorithm (Guerraoui & Rodrigues 2006; Défago, Schiper, & Urbán 2004), also described in Section 2.5.2. Both the Atomic broadcast and the non-ordered Reliable broadcast implementations used in these experiments have the uniform property (Défago, Schiper, & Urbán 2004).

4.3 Evaluation Criteria

Two main criteria were used to evaluate the replica consistency protocols, namely the system throughput and the transaction execution time. The first consists of the number of committed transactions per second (considering the transactions committed in all replicas). Protocol p_1 is said to perform better than protocol p_2 if p_1's throughput is higher than p_2's. The latter criteria corresponds to the amount of time a successfully committed transaction takes to finish its execution, which includes the local execution, the distributed certification process and the commit operation. Protocol p_1 is said to perform better than protocol p_2 if the average time a successful transaction takes to finish its execution using p_1 is lower than when using p_2.

In addition, the transaction abort rate was considered to experimentally validate the analytical model used to estimate a Bloom filter's size according to a user-determined false positive rate.

Finally, the number of objects read by an update transaction, i.e., its read set, is also important when evaluating the performance of the replica consistency protocols, since the size of the message conveying it heavily influences the throughput of the system, as shown by the results presented in this chapter.

4.4 Micro-benchmarks

4.4.1 Bank Benchmark

4.4.1.1 Description

The Bank benchmark, a synthetic workload originally used for evaluating DSTM2 (Herlihy, Luchangco, & Moir 2006), is composed by reading operations, that access an account's balance, and update operations, that transfer a given amount of money from one account to another.

Three different configurations were used in this benchmark, each serving a different purpose:

- Configuration A: validate the analytical model introduced in Section 3.8.1.2 for determining the Bloom Filter's size as a function of a target *maxAbortRate* factor.

- Configuration B: submit the implemented protocols to a variety of read set sizes and compare their performance.

- Configuration C: simulate the average workload of the FenixEDU (Carvalho, Cachopo, Rodrigues, & Rito-Silva 2008) system (regarding read and write set sizes).

Configuration A The benchmark was adapted so that the STM at each replica is initialized with a vector of $numThreads \cdot numMachines \cdot 10.000$ items. Each thread $i \in [0, numThreads - 1]$ executing on replica $j \in [0, numMachines - 1]$ accesses a distinct fragment (of indexes $[(i + j \cdot numThreads) \cdot 10.000, (1 + i + j \cdot numThreads) \cdot 10.000 - 1]$) of 10.000 elements of the array. A transaction consists of reading all these elements and randomly performing a number of write operations uniformly distributed in the range [50-100]. Given that the fragments of the array accessed by different threads never overlap, this ensures that any transaction abort is only due to false positives in the Bloom Filter based validation.

Configuration B Again, the items read and updated by the threads do not overlap. However, several tests were run for each implemented protocol, varying the number of items in the read set with the following values: 10, 100, 1.000, 5.000, 10.000, 30.000, 50.000 and 100.000; and updating only two elements. These tests were then run first in two replicas with one thread each and also in eight replicas with four threads each. This way, it is possible to observe the behaviour of each protocol in the presence of opposite network loads and a wide variety of read set sizes and with none or very few conflicts (due to the false positives of the non-voting version of the BFC) so that the performance differences are only due to the number and size of the messages exchanged between replicas. In both this configuration and in the following, C, the false positive rate is set to 1% for the non-voting BFC and 10% for the voting BFC.

Configuration C Finally, this configuration is also similar to A, but each thread reads between 40.000 and 50.000 elements of the array and updates between 30 and 40 to loosely simulate the average workload of the FenixEDU, as seen in Table 1.1. This experiment was also run in two opposite network load situations: low (two replicas with one thread each) and high (eight replicas with four threads each).

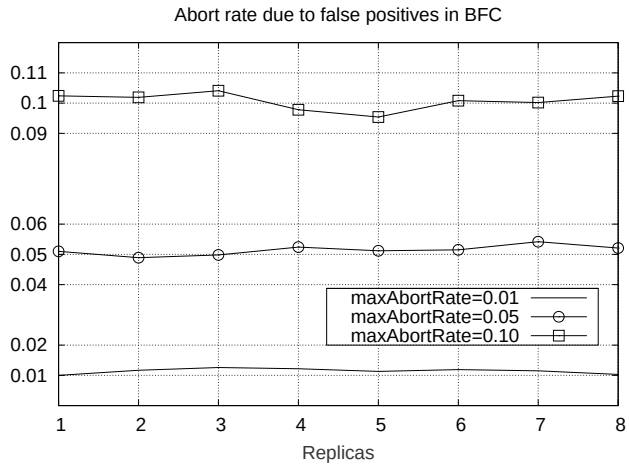
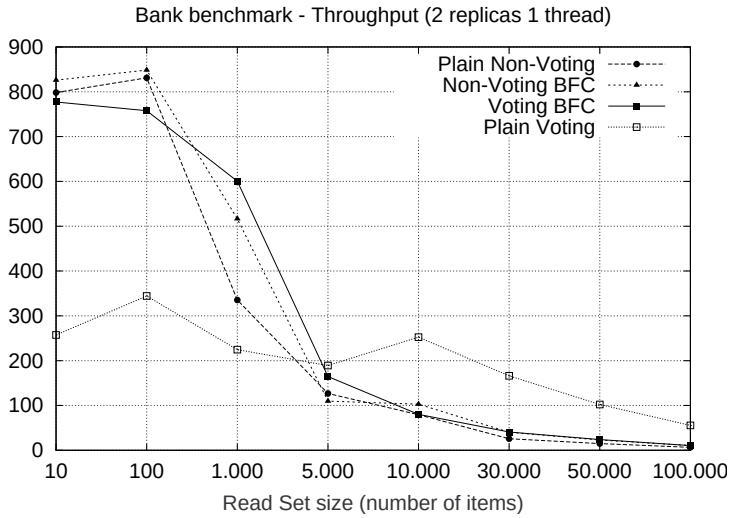


Figure 4.1: Transaction abort rate due to false positives in the Bloom Filter-based validation.

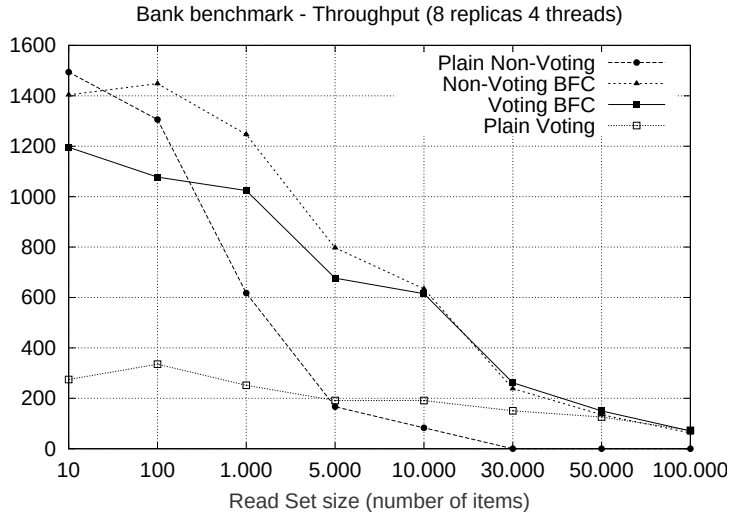
4.4.1.2 Results

Configuration A The plot in Figure 4.1 shows the percentage of aborted transactions when using the BFC scheme with a target *maxAbortRate* of 1%, 5%, 10% as the number of active replicas is varied from 1 to 8 (with 4 threads executing on each replica), highlighting the tight matching between the analytical forecast and the experimental results in presence of heterogeneous load conditions. In other words, when the only reason for a transaction not to commit is a false positive, the transaction abort rate matches the *maxAbortRate* parameter defined by the user.

Configuration B The plots in Figure 4.2 depict the throughput (commits per second) of the replica consistency protocols under different workloads and opposite network traffic scenarios. Figure 4.2(a) shows that, as the message size increases, the throughput of the four protocols tends to converge, with the exception of the plain voting scheme, which more stable, and eventually surpasses the other three. In this scheme, the size of the messages is constant, regardless of the number of items read set size (since the write sets were configured to always consist of only two items). On the other hand, for the non-voting schemes, the size of the messages grows proportionally to the size of the read sets, which means that their throughput is bound to decrease as the number of items read during a transaction increases. Additionally, since this scenario does not create heavy network traffic, the voting schemes, while needing two messages to complete the validation stage, tend to perform better for larger read sets due to the small



(a) 2 replicas and 1 thread



(b) 8 replicas and 4 threads

Figure 4.2: Bank Benchmark - Configuration B: throughput (commits/second).

Protocol	2 replicas 1 thread	8 replicas 4 threads
Plain Non-voting	17	-
Non-voting BFC	24	131
Voting BFC	26	141
Plain Voting	107	129

Table 4.1: Bank Benchmark - Configuration C: throughput (commits/second).

size of the messages and overall reduction of traffic in the network.
Figure 4.2(b) shows that, with a more congested network, both BFC versions obtain better performances. This can be explained by the fact that these protocols, besides compressing the messages exchanged, only need one message to complete the validation stage in 100% and 90% of the transactions, respectively for the non-voting and the voting versions. Conversely, the uncompressed messages of the plain non-voting protocol quickly saturate the network and for read sets larger than 10.000 items the system collapsed, hence the 0 throughput depicted in the plot. The plain voting protocol's throughput is once more constant throughout the test, however, unlike the previous situation, it never surpasses neither versions of BFC because the two messages needed to validate of a transaction take more time to be delivered in the presence of more traffic in the network. Similarly to the previous scenario, as the size of the read set increases, all protocols start converging to the same throughput, which can be due to both the large messages that congest the network and the fact that transactions take longer to execute.

Configuration C Table 4.1 shows the throughput of the replica consistency protocols under a workload with variable read and write set sizes similar to those observed in FenixEDU (Table 1.1). These results confirm the ones obtained in the previous test and indicate that, when the network traffic is not high, the plain voting protocol achieves the best results and, in the opposite situation, the voting BFC has the best performance, closely followed by the non-voting version and the plain voting protocol.

4.4.2 Red Black Tree Benchmark

4.4.2.1 Description

The Red Black tree, more complex than the previously presented benchmark, is again obtained by adapting the implementation originally used for evaluating DSTM2 (Herlihy,

Luchangco, & Moir 2006). It consists of transactions that insert, search, and remove elements from a red black tree data structure, as the name indicates. More precisely, we consider a mix of three different transactions: i) a read-only transaction, performing a sequence of searches, ii) a write transaction performing a sequence of searches and insertions, and iii) a write transaction performing a sequence of searches and removals.

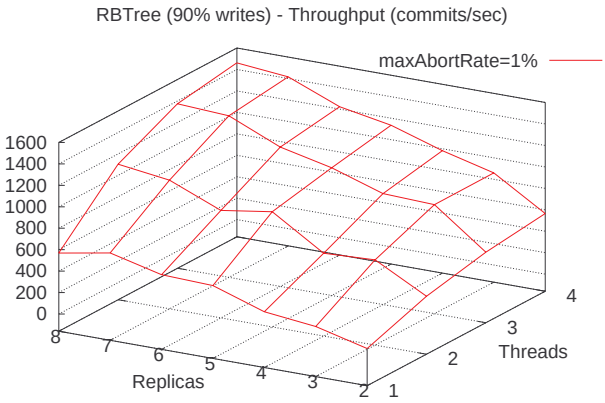
In detail, the tree is pre-populated with 50.000 (randomly determined) integer values in the range [-100.000, 100.000]. Read-only transactions consist of 200 range queries, each one spanning 5 tree's entries around a randomly chosen integer value. The insertion, resp. removal, write transactions perform first of all 20 range queries, where each query range spans 50 tree's entries, which are aimed at identifying at least a value v which is absent, resp. present, in the tree. If the sequence of range queries fail to identify any such element, the tree is sequentially scanned starting from a randomly chosen value as long as v is found or the maximum value the tree can store, namely 100.000 is reached (though this case is in practice extremely rare). Finally, if v was found, it is inserted in, resp. removed from, the tree. Note that this logic is aimed at ensuring that the insertion/removal transactions actually perform an update of the tree without, in the case of insertions, introducing duplicate keys. Also, the initial size of the data structure is sufficiently large to yield a light contention level (the abort rate varies between 1% and 4%).

4.4.2.2 Results

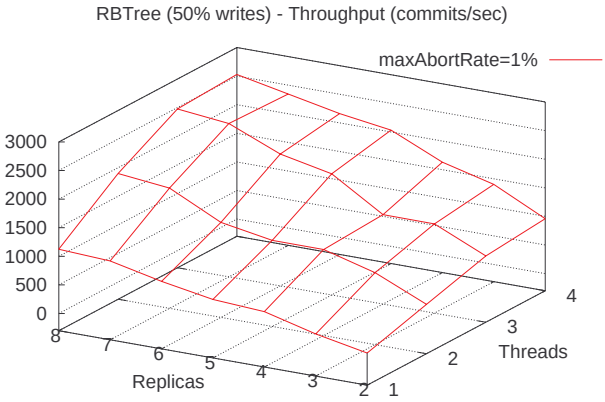
Figure 4.3 depicts the throughput of the system (i.e. number of committed transactions per second) for the three considered workloads when using BFC with the *maxAbortRate* parameter set to 1%. Each plot shows the system throughput for different combinations of number of replicas and number of server threads in each replica. The number of replicas is varied from 2 to 8 and the number of threads in each replica is varied from 1 to 4.

Only the performance of the non-voting replica consistency protocols is reported because the average size of a transaction's read set is significantly smaller than the threshold above which the voting protocols start obtaining the best results (550 items *versus* 30.000 items), as reported earlier by the Bank benchmark.

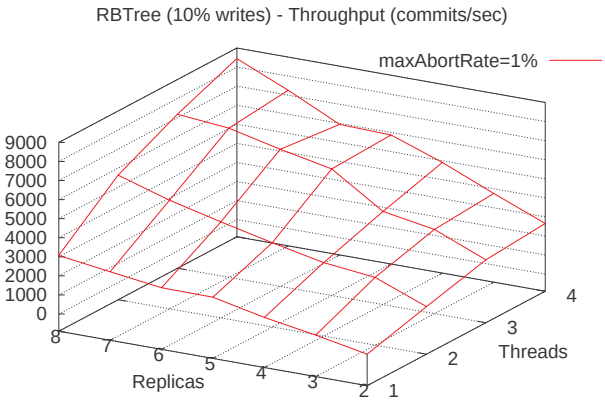
One interesting aspect of these results is that one can observe linear speedups when the number of replicas increases, even in the scenario where 90% of the transactions are write



(a) 90 % writes



(b) 50 % writes



(c) 10 % writes

Figure 4.3: Red Black Tree - Throughput ($maxAbortRate$=1%)

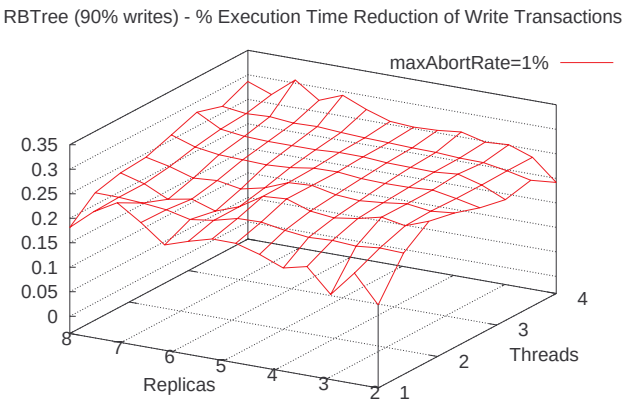


Figure 4.4: Red Black Tree - Reduction of the Execution Time of Write Transactions (*maxAbortRate*=1%)

transactions (Figure 4.3(a)). The latter is, naturally, the scenario with worse performance, given that almost all transactions require the write set to be AB-casted and applied everywhere. Still, even in this case, the throughput of the system is doubled when moving from 2 to 6 replicas. As expectable, when the percentage of update transactions is smaller, the system's performance remarkably improves. For instance, for 10% updates (Figure 4.3(c)) a configuration with 8 replicas and 4 threads achieves a throughput above 8000 tps (against the 1600 tps for the 90% update case). Also, when considering the workload with 10% updates, the configuration with 8 replicas and 4 threads per replica almost triplicates the performance of the same system with only 2 replicas (more precisely, the throughput grows from 3000 tps to more than 8000 tps).

Figure 4.4 shows the improvement in the execution time of write transactions that is obtained by the use of Bloom Filters for the scenario with 90% write transactions with respect to the standard non-voting certification algorithm requiring to atomically broadcast the whole transaction's read set described in Section 3.8.1.1. As it can be observed in the plot, the BFC's optimizations reduce the execution time of write transactions up to approximately 37% in scenarios with a large number of replicas and threads. This is due to the 10x compression of the messages achieved thanks to the Bloom Filter encoding and to the corresponding reduction of the ABcast latency, which represents a dominant component of the whole transaction's execution time. Note that since the cost of multicast grows with the number of replicas, the reduction also grows proportionally.

Parameter	Value
NumAtomicPerComp	100
NumConnPerAtomic	3
DocumentSize	20000
ManualSize	1000000
NumCompPerModule	250
NumAssmPerAssm	3
NumAssmLevels	7
NumCompPerAssm	3
NumModules	1

Table 4.2: Parameters used to build the initial data structure of STMBench7 benchmark.

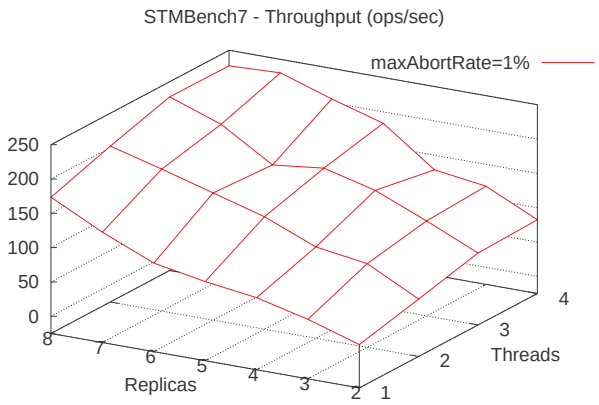
4.5 STMBench7 Benchmark

4.5.1 Description

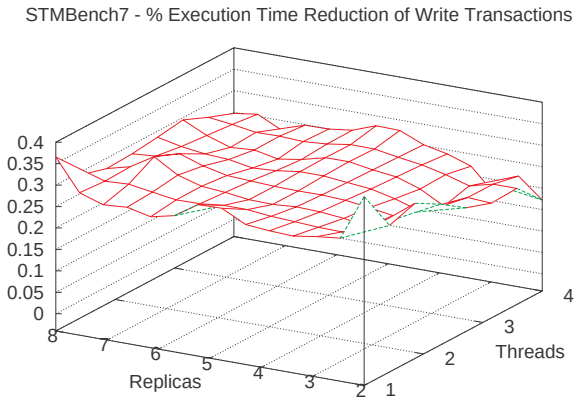
The STMBench7 benchmark (Guerraoui, Kapalka, & Vitek 2007) is a synthetic benchmark whose workloads aim at representing realistic, complex, object-oriented applications. It features a number of operations with different levels of complexity, from very short, read-only operations to very long ones that modify large parts of the data structure, which manipulate an object-graph with millions of objects heavily interconnected and three types of workloads (read dominated, read-write and write dominated). This benchmark can generate very demanding workloads which include, for instance, heavy-weight write transactions performing long traversals of the object graph generating very large read sets.

Among the three possible workloads, "read dominated with long traversals" was the one selected because of its resemblance to the FenixEDU (Carvalho, Cachopo, Rodrigues, & Rito-Silva 2008) own workload (a majority of read transactions and update transactions with large read sets).

In order to avoid the excessive growth of the size of the messages exchanged when using the plain non-voting certification algorithm, we found necessary to reduce the size of some of the benchmark's data structures with respect to their default configuration. The exact settings of the benchmark's scale parameters is reported in Table 4.2 in order to ensure reproducibility of the experiments.



(a) Throughput



(b) % Execution Time Reduction

Figure 4.5: STMBench7 - read dominated with long traversals ($maxAbortRate$=1%)

4.5.2 Results

Figure 4.5 depicts the performance of the system using the "read dominated with long traversals" workload. As before, each plot shows the system throughput for a different combination of number of replicas (from 2 to 8) and threads per replica (from 1 to 4).

Again, only the results obtained with non-voting schemes are reported for this benchmark since the average size of a transaction's read set is not significant for the voting protocols, in spite of the rare occurrence of update transactions with read sets as large as 307.430 items.

The speedup results are consistent with the results obtained with the Red Black tree benchmark. Looking at the throughput numbers in Figure 4.5(a), one can also observe linear speedups with the increase in the number of replicas. For instance, by moving from 2 to 8 replicas, the system performance increases by a factor 4x independently of the number of threads.

Figure 4.5(b) highlights the performance gains achievable thanks to the usage of Bloom filters with respect to the plain non-voting certification scheme described in Section 3.8.1.1. The reduction of execution time for write transactions (namely the only ones to require a distributed certification) reported fluctuates in the range from around 20% to around 40%. These gains were achieved, in this case, thanks to the 3x message compression factor permitted by the use of Bloom Filters.

4.6 Discussion

All benchmarks showed that the plain non-voting protocol will most likely be outperformed by both versions of the BFC. This is particularly evident in Red Black Tree and STMBench7 benchmarks, which depict reductions in the transaction execution time that can go up to 40% when using the latter certification protocols.

The Bank benchmark, in particular, configurations B and C, compares the performance of all the implemented replica consistency protocols. In all the tested scenarios, no protocol clearly dominated the rest. Instead, the performance of each protocol was heavily influenced by the size of the messages and the congestion level of the network. One very good example of this is the fact that, in a low traffic scenario, for the same number of replicas and threads, the plain voting scheme achieved the best results with messages carrying more than 10.000 items in the

read set, while both versions of the BFC's throughput surpassed the other protocols' when the messages exchanged between replicas were shorter.

An interesting finding highlighted by the experimental analysis is that, in realistic settings, the non-voting BFC scheme achieves significant performance gains even for a negligible (i.e. 1%) additional increase of the transaction's abort rate. This makes the BFC scheme viable, in practice, even in abort-sensitive applications. However, if this increase is not tolerable, with the voting version of the BFC it is possible to get nearly the same performance of the non-voting BFC (and in some situations, even surpass it) while avoiding the drawbacks of false positives thanks to the additional voting step triggered when a transaction fails the validation with Bloom filters.

Summary

This chapter presented the experimental study based on standard STM benchmarks with synthetic workloads of different complexity. It began with Bank benchmark, which served two main purposes, namely, validating the analytical model used to estimate the size of the Bloom filters and comparing the implemented protocols by varying of size of the read set. Next, the performance of the non-voting BFC with workloads with different percentages of write transactions was assessed using the Red Black Tree benchmark. STMBench7 was used to evaluate the performance of the non-voting BFC scheme in the presence of a complex workload. Finally, the results obtained were briefly discussed.

The next chapter presents the final remarks regarding the implemented protocols and the results obtained in the experimental study, as well as the future work.

5 Conclusions

5.1 Conclusions

Software Transactional Memory systems prevent the programmer from dealing explicitly with concurrency control mechanisms, which is why they have become a powerful mechanism to develop concurrent programs. However, building efficient distributed and replicated STMs to enhance both fault-tolerance and performance is still a largely unexplored problem. This thesis addressed this problem by analyzing, implementing and evaluating techniques to maintain the coherence of the replicated STM data.

To this end, the thesis introduced BFC, a novel distributed certification based protocol which relies on the properties of an Atomic Broadcast primitive to serialize conflicting transactions and uses Bloom filters as a technique to reduce the size of the messages exchanged during the validation phase by compressing the read sets. Two versions of this scheme were implemented, one purely non-voting and the other combining non-voting and voting techniques to achieve a higher message compression. Similarly to the classic non-voting certification protocols, the non-voting BFC uses a single ABcast to complete the validation phase of a transaction. In addition, it achieves a significant message compression rate even with a low increase in the number of aborted transactions. However, the false positives caused by using Bloom filters result in an increase in the abort rate and limit the achievable compression. On the other hand, in the voting version of the BFC, the occurrence of any false positive is detected. This avoids incurring in an increase of the transaction's abort rate, making it particularly attractive for abort sensitive applications. On the down side, this protocol may require a second message in order to complete a transaction's validation phase.

With the objective of validating the results obtained in the experimental evaluation of BFC, the non-voting and voting transaction validation schemes introduced in the context of database replication in (Pedone, Guerraoui, & Schiper 2003) and (Kemme & Alonso 2000), respectively,

were also implemented.

From the experimental results, it is clear that the BFC achieves an overall better performance than the plain non-voting protocol in all the experiments performed. This is due to the compression of the read sets which, in turn, reduces the network latency in message delivery, shortening the duration of the validation phase and, consequently, the transaction's execution time.

On the other hand, the experimental evaluation also showed that, for workloads with write sets significantly smaller than read sets, if the network is not congested, voting schemes can achieve better performance than non-voting ones, despite the two messages required to complete the validation phase. In a situation with more network traffic, the advantage achieved by the shorter messages is no longer critical, since the validation phase takes longer to complete due to the increased network latency.

In conclusion, the Bloom Filter Certification scheme implemented in D^{2}STM provides fault-tolerance, makes it possible to use additional replicas to improve the throughput of the system (mainly in the presence of read dominated workloads) and, last but not the least, permits to use (faster) certification approaches in the presence of diverse workloads.

5.2 Future Work

An interesting direction for future work is to experimentally evaluate the performance of the protocols presented in this thesis with different workloads. This would allow to further determine for the voting protocols in particular, both the BFC and the plain one, on one hand, how they are affected by different network traffic conditions and, on the other, how their throughput is influenced by, for the first scheme, different false positive rates and, for the latter, different write set sizes.

The experimental results for the two versions of BFC were very similar. This can be explained by the fact that our experimental testbed consisted of a cluster equipped with Gigabit Ethernet and the messages exchanged by the replicas during the tests may not have exhausted the available bandwidth. So, further experiments in scenarios with scarce bandwidth should be performed in order to determine how the system's throughput is affected by both versions of the BFC configured with a range of different values for the false positive rate.

One possible optimization for the voting version of the BFC would be to initiate the validation process for transactions whose read set does not intersect with the write set of those waiting for a message from the local replica when the remote validation failed in order to speed up this process for non-conflicting transactions. This would eventually allow to take advantage of the protocol's higher compression factor and, consequently, improve its throughput.

Finally, the results of the experimental study suggest that D^2STM should take advantage of the various certification schemes in order to improve its overall performance according to the system's workload. By transforming it into an adaptative system, the Replica Manager would be able to switch between certification protocols according to the conditions of the network and sizes of the read and write sets, leveraging on each protocol's features in order to achieve the best possible performance.

References

Baduel, L., F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, & R. Quilici (2006). *Grid Computing: Software Environments and Tools*, Chapter Programming, Composing, Deploying for the Grid (chapter 9), pp. 205–229. Springer.

Berenson, H., P. Bernstein, J. Gray, J. Melton, E. O'Neil, & P. O'Neil (1995). A critique of ansi sql isolation levels. *SIGMOD Rec. 24*(2), 1–10.

Bernstein, P., D. Shipman, & W. Wong (1979). Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering 5*(3), 203–216.

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

Bershad, B. N., M. J. Zekauskas, & W. A. Sawdon (1993). The midway distributed shared memory system. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA.

Bertsekas, D. P. & J. N. Tsitsiklis (2002, June). *Introduction to Probability*. Athena Scientific.

Birman, K. & R. van Renesse (1994, March). *Reliable Distributed Computing With the ISIS Toolkit*. IEEE CS Press.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*(7), 422–426.

Bocchino, R. L., V. S. Adve, & B. L. Chamberlain (2008). Software transactional memory for large scale clusters. In *Proceedings of the Symposium on Principles and practice of parallel programming*, New York, NY, USA, pp. 247–258. ACM.

Bolosky, W. J. & M. L. Scott (1993). False sharing and its effect on shared memory performance. In *Sedms'93: USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems*, Berkeley, CA, USA. USENIX Association.

Broder, A. & M. Mitzenmacher (2003). Network Applications of Bloom Filters: A Survey. *Internet Mathematics 1*(4), 485–509.

Cachopo, J. & A. Rito-Silva (2006). Versioned boxes as the basis for memory transactions. *Science Computer Programmming 63*(2), 172–185.

Carter, J. B., J. K. Bennett, & W. Zwaenepoel (1995). Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Trans. Comput. Syst. 13*(3), 205–243.

Carvalho, N., J. Cachopo, L. Rodrigues, & A. Rito-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 *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA)*, Montpellier, France.

Chockler, G. V., I. Keidar, & R. Vitenberg (2001). Group communication specifications: a comprehensive study. *ACM Comput. Surv. 33*(4), 427–469.

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

Couceiro, M., P. Romano, N. Carvalho, & L. Rodrigues (2009b). D2STM: Memória transaccional em software distribuída e confiável. In *Inforum - Simposium de Informática*.

Défago, X., A. Schiper, & P. Urbán (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv. 36*(4), 372–421.

Dice, D., O. Shalev, & N. Shavit (2006). Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pp. 194–208.

Fan, L., P. Cao, J. Almeida, & A. Z. Broder (2000). Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw. 8*(3), 281–293.

Felber, P. & A. Schiper (2001, April). Optimistic active replication. In *Proceedings of 21st International Conference on Distributed Computing Systems*, Phoenix, Arizona, USA. IEEE Computer Society.

Gray, J., P. Helland, P. O'Neil, & D. Shasha (1996). The dangers of replication and a solution.

In *Proc. of the 25th Conference on the Management of Data (SIGMOD)*, New York, NY, USA, pp. 173–182. ACM.

Guerraoui, R. & M. Kapalka (2008). On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, NY, USA, pp. 175–184. ACM.

Guerraoui, R., M. Kapalka, & J. Vitek (2007). Stmbench7: a benchmark for software transactional memory. *SIGOPS Operating Systems Review 41*(3), 315–324.

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

Hammond, L., V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, & K. Olukotun (2004). Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News 32*(2), 102.

Harris, T. & K. Fraser (2003). Language support for lightweight transactions. *SIGPLAN Not. 38*(11), 388–402.

Herlihy, M., V. Luchangco, & M. Moir (2006). A flexible framework for implementing software transactional memory. *SIGPLAN Not. 41*(10), 253–262.

Herlihy, M., V. Luchangco, M. Moir, & I. William N. Scherer (2003). Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, New York, NY, USA, pp. 92–101. ACM.

Herlihy, M. & J. E. B. Moss (1993). Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, New York, NY, USA, pp. 289–300. 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 Very Large Data Base Conference*, Cairo, Egypt. ACM.

Kirsch, A. & M. Mitzenmacher (2006). Less hashing, same performance: building a better bloom filter. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, London, UK, pp. 456–467. Springer-Verlag.

Kontothanassis, L., R. Stets, G. Hunt, U. Rencuzogullari, G. Altekar, S. Dwarkadas, & M. L.

Scott (2005). Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks. *ACM Trans. Comput. Syst. 23*(3), 301–335.

Kotselidis, C., M. Ansari, K. Jarvis, M. Lujan, C. Kirkham, & I. Watson (2008, Sept.). DiSTM: A software transactional memory framework for clusters. In *Proc. of the International Conference on Parallel Processing (ICPP)*, pp. 51–58.

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

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers 28*(9), 690–691.

Li, K. & P. Hudak (1989). Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst. 7*(4), 321–359.

Lin, Y., B. Kemme, M. P. no Martnez, & R. Jiménez-Peris (2005, June). Middleware based Data Replication providing Snapshot Isolation. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, Baltimore, Maryland, USA. ACM.

Manassiev, K., M. Mihailescu, & C. Amza (2006). Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the Symposium on Principles and practice of parallel programming*, New York, NY, USA, pp. 198–208. ACM.

Marathe, V. J. & M. L. Scott (2004, Jun). A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept.

Martin, M., C. Blundell, & E. Lewis (2006). Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett. 5*(2), 17.

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

Mitzenmacher, M. (2002). Compressed bloom filters. *IEEE/ACM Trans. Netw. 10*(5), 604–612.

Morin, C. & I. Puaut (1997). A Survey of Recoverable Distributed Shared Virtual Memory Systems. *IEEE Transactions on Parallel and Distributed Systems 8*(9), 959–969.

Pedone, F., R. Guerraoui, & A. Schiper (2003). The Database State Machine Approach. *Distributed and Parallel Databases 14*(1), 71–98.

Perez-Sorrosal, F., M. Patino-Martinez, R. Jimenez-Peris, & B. Kemme (2007). Consistent and scalable cache replication for multi-tier j2ee applications. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, New York, NY, USA, pp. 328–347. Springer-Verlag New York, Inc.

Powell, D. (1996). Group communication. *Communications of the ACM 39*(4), 50–53.

Protic, J., M. Tomasevic, & V. Milutinovic (1996). Distributed Shared Memory: Concepts and Systems. *IEEE Parallel Distrib. Technol. 4*(2), 63–79.

Rodrigues, L., J. Mocito, & N. Carvalho (2006). From spontaneous total order to uniform total order: different degrees of optimistic delivery. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, New York, NY, USA, pp. 723–727. ACM.

Romano, P., N. Carvalho, & L. Rodrigues (2008, September). Towards distributed software transactional memory systems. In *Proceedings of the Workshop on Large-Scale Distributed Systems and Middleware (LADIS 2008)*, Watson Research Labs, Yorktown Heights (NY), USA. (invited paper).

Schneider, F. B. (1993). *Replication management using the state-machine approach.* ACM Press/Addison-Wesley Publishing Co.

Shavit, N. & D. Touitou (1995). Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, New York, NY, USA, pp. 204–213. ACM.