Cache Coherence in Distributed and Replicated Transactional Memory Systems

Maria Isabel Catarino Couceiro
maria.couceiro@ist.utl.pt

Instituto Superior Técnico

(Advisor: Professor Luís Rodrigues)

Abstract. Software Transactional Memory (STM) systems have emerged as a powerful paradigm to develop concurrent applications. Implementing distributed and replicated versions of this paradigm is important for scalability and fault-tolerance reasons. In such implementations, the STM is replicated in multiple nodes, which have to coordinate to ensure the consistency of the cached data. This work studies techniques that allow to maintain the coherence of the STM replicated data, identifies some limitations in the existing techniques, proposes some alternatives that may help to mitigate these limitations, and discusses how these alternatives can be implemented and evaluated.

1 Introduction

Software Transactional Memory (STM) systems have emerged as a powerful paradigm to develop concurrent applications [1,2,3]. 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 [4], a web application in which the transactional semantics of an operation are supported by a STM running at each application server.

This work is performed in the context of the Pastramy[1] project (Persistent and highly Available Software TRAnsactional MemorY). One of the goals of this project is to build a more scalable and fault-tolerant version of the FenixEDU system. In this system, servers run a 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

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

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 discusses how these alternatives can be implemented and evaluated.

The rest of this report is organized as follows. Section 2 motivates and describes the goals of this work. Section 3 provides an introduction to the different technical areas related to it. Section 4 drafts the architecture of the solution to be implemented as future work and Section 5 describes how this solution will be evaluated. The scheduling of future work is presented in Section 6. Finally, Section 7 concludes this report by summarizing its main points.

2 Goals

This work aims at contributing to the implementation of efficient distributed and replicated software transactional memory systems. More precisely:

> *Goals:* This works aims at analyzing, designing, and evaluating techniques to maintain the coherence of the replicated STM data.

The work started with a survey of related work (to be presented in Section 3). From this survey it becomes clear that one of the most promising techniques is based on the use of total order broadcast primitives [5] 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.

Table 1 shows the size of read and write sets of transactions in the FenixEDU system collected over a period of two weeks [4]. 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.

Table 1. Average and maximum read set and write set sizes for each kind of transaction in the FenixEDUv2 system. [4]

	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

To mitigate the problem above, this report proposes the use of Bloom filters [6] as a space-efficient technique to disseminate read sets. Different replica consistency algorithms, with and without the use of Bloom filters, will be implemented and their performance analyzed. So, the objective is to achieve the following results:

> *Expected results:* This work will: i) implement a set of consistency algorithms for replicated STMs; ii) provide an experimental comparative evaluation of these algorithms and; iii) provide a quantified assessment of the advantages and disadvantages to use Bloom filters as a way to represent read sets in this class of applications.

3 Related Work

Understanding the problems this project poses implies knowing the fundamentals of different areas in distributed systems.

This section starts by addressing Software Transactional Memory (STM) systems (Subsection 3.1), which are the basis for building a distributed STM (DSTM) system. To understand how STM can be distributed, this report briefly surveys Distributed Shared Memory systems (Subsection 3.2) and then describes a DSTM system inspired by those algorithms (Subsection 3.3). Then, with the aim of replicating the STM we start by introducing some relevant communication and coordination abstractions, namely group communication primitives (Subsection 3.4). 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 (Subsection 3.5) 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 (Subsection 3.6) are introduced as a mechanism that may be used to reduce the size of the messages exchanged among sites.

3.1 Software Transactional Memory Systems

Transactional Memory A concurrent program is one that uses multiple threads of control (named threads or processes) that execute concurrently and access shared objects. These objects are called concurrent objects [7]. 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 [8] 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 [7].

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 [9]). 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.

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 [8]. 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. Examples include [1,2,3]. 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 [10] (STM) is an example of such algorithms.

This solution provides a non-blocking implementation of static transactions, a special form of transactions in which 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 [1,2,3] has focused on providing software transactional memory algorithms for concurrent objects dynamically, since many applications allocate and use data structures dynamically.

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 [11] is a STM designed with the aim of minimizing the number of aborts. For that purpose, the system maintains 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. Furthermore, conflicts are avoided by delaying computations until commit time. Conflicts that cannot be avoided, may be resolved by re-executing the conflicting part of the transaction. The per-transaction box mechanism is implemented as a Java library (JVSTM[2]) and is currently being used in the FenixEDU project [4] to reduce the conflicts on the collections used to implement the domain relationships.

3.2 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 [12].

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 [13]. 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 [14]. Since whole pages are transfered, performance degrades due to the false sharing problem. False

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

sharing [15] 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 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 [16] 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 [12].

Memory Consistency It is possible to build a DSM system were 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 transfered 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 [17,18], 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 [13].

Unfortunately, the overhead of enforcing the previous memory consistency models is non-negligible [12,19]. 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 [18]. 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.

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 [13].

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. 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.

3.3 Distributed Software Transactional Memory

One way to build a distributed software transaction memory is to use techniques inspired by distributed shared memory systems. An example of one of such systems is DiSTM [20], a STM framework for clusters. The architecture used in this implementation relies on a central master, in charge of coordinating the execution of the cluster. DiSTM has two core components: a transactional execution engine, DSTM2 [21], and a remote communication system, based on the ProActive framework [22].

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. Each active object has one entry point, the root, which is accessible from anywhere. Each node has a number of active objects serving several requests. The main thread on the master node is an active object that serves requests from the worker nodes. A worker node has two active objects: one that coordinates the execution on the node and is responsible for updating the worker node's datasets upon a transaction's commit, and another that accepts transactions from other nodes wanting to be validated against the transactions running on that node.

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

Transactional Coherence and Consistency DiSTM implements an adaptation of the decentralized Transactional Coherence and Consistency [23] 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.

Lease-based Transactional Memory Coherence Protocols

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.

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.

3.4 Group Communication

Group communication [24,25] 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 [26], 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 [24]. 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 [27]) 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.

Total Order Broadcast Agreement is a class of problems in distributed systems in which processes have to reach a common decision. Total Order broadcast [5] 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', the p TO-delivers m before m' only if q TO-delivers m before m'.

The two first properties are liveness properties [5] (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 [5] (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.

Implementing Total Order According to [5], 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.

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 [28]. 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 [28]. 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 not compensate the performance gains.

3.5 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 [29], 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 report the focus is on variants of eager, update-anywhere protocols for fully replicated databases, as we would like to maximize 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 [30], Postgres-R [31], SI-Rep [32] and Optimistic Active Replication [33]. 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 isolation, etc), to name but a few, that are useful to understand some fundamental replication mechanisms that could also be used in distributed STM systems.

The Database State Machine Approach

Overview. The Replicated State Machine Approach [34] 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 3.4.

The Database State Machine Approach [30] (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[3] 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 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.

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.

[3] 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.

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.

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.

Postgres-R

Overview. Postgres-R [31] 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[4] 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 transaction´s 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.

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

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.

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.

SI-Rep

Overview. SI-Rep [32] 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 [35] 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.

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.

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.

Optimistic Active Replication

Overview. The Optimistic Active Replication [33] is another example of the Replicated State Machine approach [34]. 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 3.4. 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.

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.

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.

3.6 Bloom Filters

A Bloom filter [6] 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 [36] 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. Else, y is not a member of S. Figure 1 shows how a bloom filter is built and used.

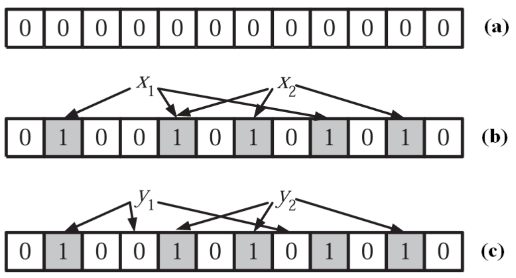

Fig. 1. Example of a Bloom filter [36]. In *(a)*, the filter is an array of 12 bits with all positions set to 0. In *(b)*, it represents the set $S = \{x_1, x_2\}$ with each element hashed three times, and the resulting bits set to 1. To check if elements y_1 and y_2 are in the filter, each must be hashed 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 false positive rate depends on the number of hash functions (k), the size of the array (m) and the number of elements in S (n). After all elements of S are hashed into the Bloom filter, the probability that a specific bit is 0 is $p = \left(1 - \frac{1}{m}\right)^{kn}$. If h_k return optimally balanced random values, the probability that a test will return a false positive result is approximately $(1-p)^k$.

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.

Counting Bloom filters. The counting Bloom filter [37] aims to avoid the problem of insertion and deletion of elements in a filter. Each entry is not a single bit but rather a small counter (it tries to answer the question "*How many occurrences of the element x are in set S?*"). Every time an element is inserted, the corresponding counters are incremented, and when it is deleted, the counters are decremented.

Compressed Bloom filters. Some application use Bloom filters to exchange data across a network, making the size of the transmitted filter relevant. A compressed Bloom filter [38] may lead to significant bandwidth savings at the cost of higher memory requirements (as result of larger uncompressed filters) and some additional computation time to compress the filter that is sent across the network. This results in a new tradeoff: transmission size *versus* uncompressed size.

Applications Since their introduction, Bloom filters have seen various uses [36].

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 [37] 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.

3.7 Summary of the Related Work

Since the goal of this project is to build a distributed software transactional memory system, this section started by introducing two fundamental concepts: transactional memory and transaction. The latter abstraction is used by both software transactional memory systems and database systems. Given the similarity among the two systems, some basic mechanisms of database replication, such as certification, were also introduced. In modern replicated database systems, communication between sites is done by means of a total order broadcast primitive, which ensures data consistency and serializability of clients requests. The design and implementation of these communication primitives was also briefly addressed.

Distributed shared memory systems share similar goals with replicated and distributed database systems as they both attempt to minimize the access time to shared data that is to be kept consistent. In addition, both systems share the same set of design choices [12]:

1. when and where to allocate physical copies of logical data to minimize access latency;
2. when and how to update them to maintain an acceptable degree of mutual consistency, while minimizing coherence management overhead.

However, the algorithms used in these two systems can display significant differences, as databases must provide for persistent data, reliability and atomicity.

This section also introduces a mechanism that may be used to reduce the size of the messages exchanged by sites. Bloom filters are data structures that support membership queries and allow for trading off the space compression factor for the possible existence of false positives.

An alternative approach to build a distributed software transactional memory system different from the one that will be presented in Section 4 as the basis of this work was also presented. While the former is a master-centric approach with the master node responsible for maintaining data consistency among the worker nodes, the latter is decentralized and relies on a total order primitive to serialize transactions.

4 Solution's Architecture

4.1 Overview

At this stage, the envisaged architecture to build a distributed replicated STM to support the execution of the FenixEDU system is the one described in [4]. In this architecture, illustrated in Figure 2, each of the application servers is composed of the following components:

– Request Processor (RP): responsible for receiving the requests and starting the VSTM transactions.
– VSTM: Transactions are executed locally and optimistically by a Versioned Software Transactional Memory (introduced in Section 3.1), as described in the following paragraph.
– The Certification Manager (Cert): certifies the transactions ready to commit to check for conflicts with concurrent transactions. The two certification protocols used in this architecture will be further described in Section 4.2.
– Group Communication (GC): responsible for maintaining up-to-date information regarding the membership of the group of application servers (including failure detection), and providing the required communication support for the coordination among servers. A more detailed description of the services this module provides is presented later.

– Persistent Store (PS): responsible for persisting the updates on stable storage for later recovery.
– Cache Manager (CM): responsible for maintaining a consistent cache of regularly used objects.

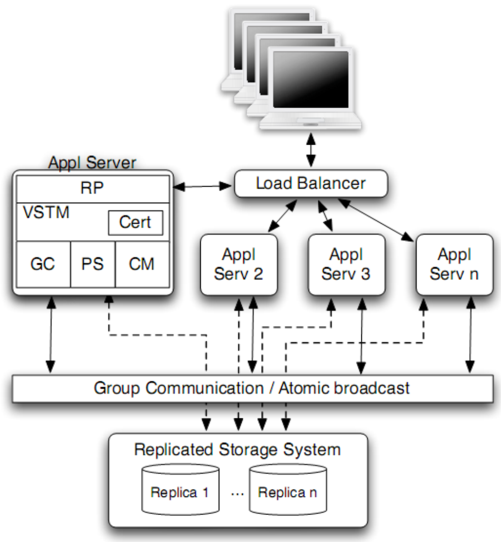

Fig. 2. Proposed Architecture of a distributed replicated STM for the FenixEDU system [4].

A transaction is started whenever the application server receives a request from a client, via a load balancer and it may require the retrieval of some objects from the storage system. The information kept by each transaction consists of its identifier (a version number used to read the right values from the versioned boxes and to tag the values written during its execution), the set of boxes read during its execution and the set of values written by it [11]. Upon commit, depending on the certification protocol used, either only the transaction's write set or both its read and write sets are broadcast. The transaction is then validated and committed or aborted in all application servers. Consistency is ensured by the ordering the total order primitive imposes.

Two services are provided by the Group Communication module: a Total Order broadcast service and a non-ordered Reliable broadcast service. The Total Order broadcast is used by the VSTM to disseminate both the read and write sets or just the write sets of transactions that are ready to commit, depending on the certification protocol. The non-ordered Reliable broadcast service is used when a local certification protocol is employed by an application server to notify the other application servers about the outcome of transactions. This is achieved by using the Appia system [39], a layered communication support framework.

My work will consist of designing, implementing and evaluating different replica consistency algorithms for the Certification Manager module.

4.2 Replica Consistency Algorithms

It is clear from the architecture above that the replica consistency algorithms based on total order primitives are considered to be the most promising to build an efficient replicated STM. Therefore, two certification based protocols will be employed: local certification and global certification. Both these algorithms were already introduced in Section 3.5.

Global certification, used in the Database State Machine approach, requires every database site 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. So, as the name implies and since 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. Otherwise, it is aborted.

On the other hand, in local certification, used by the Postgres-R algorithm, only the write set is broadcast to the other replicas. This means that only the replica where the transaction was executed can determine its outcome. If the local replica detects a conflict, it broadcast a message warning the other replicas to discard the write set. Else, it can multicast the decision message commit so that the other replicas can apply the write set to the databases. These notification messages are not required to be broadcast using the total order primitive.

As noted before, a variant of the global certification algorithm will also be built using Bloom filters as a technique to reduce the size of the messages exchanged. The expected use of this technique is detailed in the next subsection.

4.3 Reducing Message Size

From the analysis of Table 1 (presented in Section 2), it is clear that read sets are considerably larger than write sets. Furthermore, read sets do not need to convey data values; only the object identifier and the number of the version read need to be exchanged, which is more appropriate to be represented in a filter. Therefore, we will experiment to use Bloom filter to represent read sets. This will result in the reduction of the size of the messages exchanged between sites, which will consist of the write set and a smaller representation of the read set.

When a transaction finishes its execution and is ready to have its read and write sets sent to the other sites, a filter representing the compressed read set is built (this representation consists of an array of bits, which are either set to 0 or 1, according to the results of the hash functions applied to each entry of the read set). Since different transactions have read sets of different dimensions, the size of the filter has to be adjusted accordingly. The number of hash functions needs also to be set according to the size of the filter, so that the false positives rate is kept at a constant level throughout the execution. The message composed of the filter and the transaction's write set is then broadcast using the total order primitive.

When a site receives this message, the compressed read set needs to be translated into meaningful information so that the certification of the transaction can take place. So, before the certification test is performed, the (probable) entries of the read set are determined using the transmitted filter, resulting in a read set with all the entries the original had plus the result of some false matches (consequence of the false positives problem).

4.4 Challenges

The most relevant drawback associated with the use of bloom filters is the existence of false positives. Since the read set specifies the set of objects (and versions) read during the execution of a transaction, if there is an indication that there are more objects in the filter representing the read set than in the actual read set, the probability of conflict increases during the global certification stage, and so does the transaction abort rate.

However, as the false positive rate is constant and known, its value may be adjusted so that consequences of false hits during certification do not surpass the performance gains of reducing the size of the messages broadcast.

Moreover, there may also be some additional computational overhead associated with the creation and decoding of the filters. This overhead is proportional to the number of hash functions used to represent an entry in the filter. This means that this number needs to be configured in order not to greatly affect the performance, but still keeping the false positives rate at a reasonable level.

Finally, the variable dimension of read sets must be taken into consideration when defining the size of the corresponding filter, since it directly influences both the false positives rate and the size of the transmitted message.

5 Methodology for Evaluating the Work

This work will be integrated in a running system, the Pastramy prototype. Therefore, the performance of the different consistency algorithms will be determined by experimental evaluations.

5.1 Metrics

The most important metrics to evaluate the performance of the algorithms are:

- the number of transactions delivered at a site;
- the number of aborted transactions.

These metrics allow us to compare the system's performance with and without the blooms filters and to assess the influence of the false positives in the certification test. Furthermore, it will be also possible to determine how each of the following parameters affect the false positives rate and the performance of the total order primitive:

– number of hash functions;
– size of the filter (considering the size of the read set).

As a result, we expect to get insights on how to configure Bloom filters for optimal performance and on which scenarios, if any, their use allows significant improvement on the performance of replicated STMs.

5.2 Benchmarks

The performance of web applications is usually evaluated using benchmarks such as TPC-App [40], which simulates the activities of a business-to-business transactional application server. This benchmark can approximate the typical usage of the FenixEDU system, since it falls into this category of applications. However, it is interesting to assess how the proposed solutions perform under diverse workloads. Thus, alternative benchmarks will be used. Examples of benchmarks that can be used to evaluate the performance of the implemented algorithms include:

– LeeBenchmark [41,42]: based on Lee's circuit routing algorithm, a classic algorithm for laying routes on a circuit board. Each thread attempts to lay a route on the mainboard. Conflicts occur when two threads try to write the same cell in the circuit. Transactions are significant in size, yet they exhibit very regular access patterns - every transaction first reads a large number of locations (searching for suitable paths) and then updates a small number of them (setting up a path).
– DSTM2 [21]: provides a transactional programming interface for the Java programming language enabling the comparison of different (STM) implementations using the same transactional application code. The library uses transactional factories (obstruction-free and the shadow factories) to transform sequential (unsynchronized) classes into atomic (transactionally synchronized) ones.
– STMBench7 [43,42]: is a synthetic benchmark whose workloads aim at representing realistic, complex, object-oriented applications. It exhibits a large variety of operations (from very short, read-only operations to very long ones that modify large parts of the data structure) and workloads (from those consisting mostly of read-only transactions to write-dominated ones). When compared to other benchmarks, its data structures are larger and the transactions are longer and access a larger number of objects.

It is highly probable that any of the examples presented above may have to be modified in order to support the distributed architecture of the system.

6 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3: Perform the complete experimental evaluation of the results.
- May 4- May 23, 2009: Write a paper describing the project.
- May 24 - June 15: Finish the writing of the dissertation.
- June 15, 2009: Deliver the MSc dissertation.

7 Conclusions

This work addresses the implementation of efficient distributed and replicated software transactional systems. These systems emerge from the convergence of multiple previous research areas, namely: (centralized) software transactional memory systems, replicated databases and distributed shared memory. The key issues in these areas were briefly surveyed.

From this survey it was possible to conclude that the most promising techniques to maintain the consistency of replicated STMs are the certification based protocols, based on totally ordered broadcast primitives. However, the large size of a transaction's read set may be a significant impairment to the performance of such algorithms. To address this problem, Bloom filters were suggested to encode read sets in a space-efficient manner.

The report is concluded with a brief description of the architecture that will be used to implement, experimentally compare, and validate the proposed replica consistency protocols.

Acknowledgments

I am grateful to the Pastramy team, namely to Nuno Carvalho and Paolo Romano, for their support and comments during the preparation of this report.

References

1. Herlihy, M., Luchangco, V., Moir, M., William N. Scherer, I.: 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, ACM (2003) 92–101
2. Harris, T., Fraser, K.: Language support for lightweight transactions. SIGPLAN Not. **38**(11) (2003) 388–402
3. Fraser, K.: Practical lock freedom. PhD thesis, Cambridge University Computer Laboratory (2003) Also available as Technical Report UCAM-CL-TR-579.

4. Carvalho, N., Cachopo, J., Rodrigues, L., Rito Silva, A.: 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 (March 2008)
5. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. **36**(4) (2004) 372–421
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7) (1970) 422–426
7. Marathe, V.J., Scott, M.L.: A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept. (Jun 2004)
8. Herlihy, M., Moss, J.E.B.: 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, ACM (1993) 289–300
9. Bernstein, P., Shipman, D., Wong, W.: Formal aspects of serializability in database concurrency control. IEEE Transactions on Software Engineering **5**(3) (1979) 203–216
10. Shavit, N., Touitou, D.: Software transactional memory. In: PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (1995) 204–213
11. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science Computer Programmming **63**(2) (2006) 172–185
12. Protic, J., Tomasevic, M., Milutinovic, V.: Distributed Shared Memory: Concepts and Systems. IEEE Parallel Distrib. Technol. **4**(2) (1996) 63–79
13. Li, K., Hudak, P.: Memory Coherence in Shared Virtual Memory Systems. ACM Trans. Comput. Syst. **7**(4) (1989) 321–359
14. Kontothanassis, L., Stets, R., Hunt, G., Rencuzogullari, U., Altekar, G., Dwarkadas, S., Scott, M.L.: Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks. ACM Trans. Comput. Syst. **23**(3) (2005) 301–335
15. Bolosky, W.J., Scott, M.L.: 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 (1993)
16. Bershad, B.N., Zekauskas, M.J., Sawdon, W.A.: The midway distributed shared memory system. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA (1993)
17. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **28**(9) (1979) 690–691
18. Morin, C., Puaut, I.: A Survey of Recoverable Distributed Shared Virtual Memory Systems. IEEE Transactions on Parallel and Distributed Systems **8**(9) (1997) 959–969
19. Carter, J.B., Bennett, J.K., Zwaenepoel, W.: Techniques for reducing consistency-related communication in distributed shared-memory systems. ACM Trans. Comput. Syst. **13**(3) (1995) 205–243
20. Kotselidis, C., Ansari, M., Jarvis, K., Lujan, M., Kirkham, C., Watson, I.: Distm: A software transactional memory framework for clusters. Parallel Processing, 2008. ICPP '08. 37th International Conference on (Sept. 2008) 51–58
21. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. SIGPLAN Not. **41**(10) (2006) 253–262

22. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, Composing, Deploying for the Grid (chapter 9). In: Grid Computing: Software Environments and Tools. Springer (2006) 205–229
23. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. SIGARCH Comput. Archit. News **32**(2) (2004) 102
24. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. **33**(4) (2001) 427–469
25. Powell, D.: Group communication. Communications of the ACM **39**(4) (1996) 50–53
26. Birman, K., van Renesse, R., eds.: Reliable Distributed Computing With the ISIS Toolkit. IEEE CS Press (March 1994)
27. Lamport, L.: Time, clocks and the ordering of events in a distributed system. CACM **21**(7) (July 1978) 558–565
28. Rodrigues, L., Mocito, J., Carvalho, N.: 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, ACM (2006) 723–727
29. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM (1996) 173–182
30. Pedone, F., Guerraoui, R., Schiper, A.: The Database State Machine Approach. Distrib. Parallel Databases **14**(1) (2003) 71–98
31. Kemme, B., Alonso, G.: 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 (September 2000)
32. Lin, Y., Kemme, B., no Martínez, M.P., Jiménez-Peris, R.: Middleware based Data Replication providing Snapshot Isolation. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, Baltimore, Maryland, USA, ACM (June 2005)
33. Felber, P., Schiper, A.: Optimistic active replication. In: Proceedings of 21st International Conference on Distributed Computing Systems, Phoenix, Arizona, USA, IEEE Computer Society (April 2001)
34. Schneider, F.: Replication management using the state-machine approach. In Mullender, S., ed.: Distributed Systems, 2nd Edition. ACM-Press. Addison-Wesley (1993)
35. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. SIGMOD Rec. **24**(2) (1995) 1–10
36. Broder, A., Mitzenmacher, M.: Network Applications of Bloom Filters: A Survey. Internet Mathematics **1**(4) (2003) 485–509
37. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Netw. **8**(3) (2000) 281–293
38. Mitzenmacher, M.: Compressed bloom filters. IEEE/ACM Trans. Netw. **10**(5) (2002) 604–612
39. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. Distributed Computing Systems, 2001. 21st International Conference on. (Apr 2001) 707–710
40. García, D.F., García, J., García, M., Peteira, I., García, R., Valledor, P.: Benchmarking of Web Services Plattforms - An Evaluation with the TPC-APP Benchmark. In: Proceedings of 2nd International Conference on Web Information Sys-

tems and Technologies, WEBIST 2006, Setúbal, Portugal, INSTICC Press (Apr 2006) 75–80

41. Ansari, M., Kotselidis, C., Watson, I., Kirkham, C.C., Luján, M., Jarvis, K.: Lee-tm: A non-trivial benchmark suite for transactional memory. In Bourgeois, A.G., Zheng, S.Q., eds.: ICA3PP. Volume 5022 of Lecture Notes in Computer Science., Springer (2008) 196–207
42. Dragojevic, A., Guerraoui, R., Kapalka, M.: Stretching Transactional Memory. Technical report, EPFL, Lausanne, Switzerland (2008)
43. Guerraoui, R., Kapalka, M., Vitek, J.: Stmbench7: a benchmark for software transactional memory. SIGOPS Operating Systems Review **41**(3) (2007) 315–324