Exploiting Atomic Broadcast in Replicated Databases

Fernando Pedone   Rachid Guerraoui   André Schiper

Département d'Informatique

Ecole Polytechnique Fédérale de Lausanne

1015 Lausanne, Switzerland

Abstract

Database replication protocols have historically been built on top of distributed database systems, and have consequently been designed and implemented using distributed transactional mechanisms, such as atomic commitment. We argue in this paper that this approach is not always adequate to efficiently support database replication and that more suitable alternatives, such as atomic broadcast primitives, should be employed instead. More precisely, we show in this paper that fully replicated database systems, based on the deferred update replication model, have better throughput and response time if implemented with an atomic broadcast termination protocol than if implemented with atomic commitment.

1 Introduction

Replication is considered a cheap software based way to increase data availability when compared to hardware based specialised techniques [16]. However, designing a replication scheme that provides reasonable performance and maintains data consistency is still an active area of research both in the database and in the distributed systems communities.

In the database context, replication techniques based on the *deferred update model* have received increasingly attention in the past years [13]. According to the deferred update model, transactions are processed locally at one server (i.e., one replica manager) and, at commit time, are forwarded for certification to the other servers (i.e., the other replica managers). Deferred update replication techniques offer many advantages over *immediate update* techniques, which synchronise every transaction operation across all servers. Among these advantages, one may

cite: (a) better performance, by gathering and propagating multiple updates together, and localising the execution at a single, possibly nearby, server (thus reducing the number of messages in the network), (b) more flexibility, by propagating the updates at a convenient time (e.g., during the next dial-up connection), (c) better support for fault tolerance, by simplifying server recovery (i.e., missing updates may be demanded to other servers), and (d) lower deadlock rate, by eliminating distributed deadlocks [13].

Nevertheless, deferred update replication techniques have two limitations. Firstly, the termination protocol used to propagate the transaction to other servers for certification is usually an atomic commitment protocol (e.g., a 2PC algorithm [12]), whose cost directly impacts transaction response time. Secondly, the certification procedure that is usually performed at transaction termination time consists in aborting all conflicting transactions.[1] Such certification procedure typically leads to a high abort rate if conflicts are frequent, and hence impacts transaction throughput.

In the context of client-server distributed systems, most replication schemes (that guarantee strong replica consistency) are based on *atomic broadcast* communication [6, 20]. An *atomic broadcast* communication primitive enables to send messages to a set of processes, with the guarantee that the processes agree on the *set* of messages delivered, and on the *order* according to which the messages are delivered [18]. With this guarantee, consistency is trivially ensured if every operation on a replicated server is distributed to all replicas using atomic broadcast. Although several authors have mentioned the possibility of using atomic broadcast to support replication schemes in a database context (e.g., [6, 24]), little work has been done in that direction (we will mention some exceptions in Section 6).

In this paper, we show that atomic broadcast can successfully be used to improve the performance of the database deferred update replication technique. In particular, we show that, for different resilience scenarios, the deferred update replication technique based on atomic broadcast provides better transaction throughput and response time than a similar scheme based on atomic commitment.

The paper is organised as follows. In Section 2, we present the replicated database model, and we recall the principle of the deferred update replication technique. In Section 3, we describe a variation of this replicated technique based on atomic broadcast, and we prove it correct. In Section 4, we compare the transaction throughput of deferred update replication based on atomic broadcast with the throughput of deferred update replication based on classical atomic

[1] Note that we do not consider here replication protocols that allow replica divergence and require reconciliations. We focus on replication schemes that guarantee *one-copy serializability* [4].

commitment. In Section 5, we consider different resilience scenarios, and for each one, we compare the transaction response time of our solution with that of a classical atomic commit one. In Section 6 we discuss related work that study alternatives to the traditional two phase commit for database replication. Section 7 summarises the main contributions of the paper and discusses some research perspectives.

2 The Deferred Update Technique

In this section we describe our database model, and we recall the principle of the deferred update replication technique [4].

2.1 Replicated Database Model

We consider a replicated database system composed of a set of processes $\Sigma = \{p_1, p_2, \ldots, p_n\}$, each one executing in a different processor, without shared memory and access to a common clock. Each process has a replica of the database and plays the role of a replica manager. Processes may fail by crashing, and can recover after a crash. We assume that processes do not behave maliciously (i.e., we exclude Byzantine failures). If a process p is able to execute requests at a certain time τ (i.e., p did not fail or p has failed but recovered) we say that the process p is *up* at time τ. Otherwise the process p is said to be *down* at time τ. We say that a process p is correct if there is a time after which p is forever up.[2]

Processes execute transactions, that are sequences of read and write operations followed by a commit or abort operation. Transactions are submitted by client processes executing in any processor (with or without a replica of the database). Our correctness criterion for transaction execution is one-copy serializability (1SR) [4].

Committed transactions are represented as T_i, T_j and T_k. The set Π_p^τ contains all committed transactions in process p at time τ. Non-committed transactions are represented as t_m and t_n. When a transaction t_m commits, it is represented as T_i, where i indicates the serial order of t_m, related to the already committed transactions in the database.

[2]The notion of *forever up* is a theoretical assumption to guarantee that correct processes do useful computation. This assumption prevents cases where processes fail and recover successively without being up enough time to make the system evolve. *Forever up* means, for example, from the beginning until the end of a termination protocol.

2.2 The Deferred Update Principle

In the deferred update replication technique, transactions are locally executed in one process (e.g., the process at the site where the transaction was issued), and during their execution, no interaction among other processes is necessary (there is only local synchronisation). When a transaction requests the commit, its updates (e.g., the redo log records) and its control structures are propagated to the other processes (Figure 1), so that the transaction can be certified and, if possible, committed. This procedure, starting with the commit request, is called *termination protocol*. The certification test aims at ensuring one-copy serializability. That is, it guarantees that committed transactions have an effect over the database that is equivalent to a serial execution of these transactions on one copy of the database.

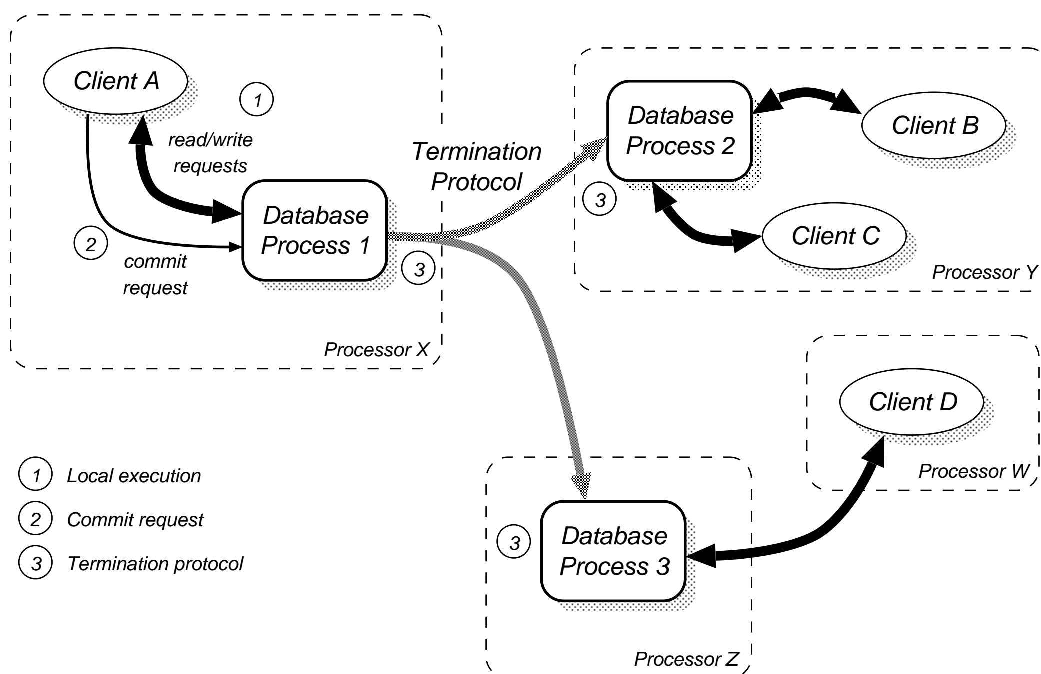

Figure 1: Deferred Update Technique

The certification test decides to abort a transaction if its commit leads the database to an inconsistent state. For example, consider two concurrent transactions, t_m and t_n, that are executed in different processes, and that update a common data item. On requesting the commit, if t_m arrives before t_n in a process but after t_n in another process, both might have to be aborted since one process would see t_m's updates after t_n's, and the other t_n's updates after t_m's.

In the deferred update technique, a transaction passes through some well-defined states (see Figure 2). It starts in the *executing* state, when its read and write operations are locally executed

by the process where it initiated. When the transaction requests the commit, it passes to the *committing* state and is sent to the other processes. A transaction received by a process is also in the committing state. A transaction in a process remains in the committing state until its fate is known by the process (i.e., *commit* or *abort*). The executing and committing states are transitory states, whereas the commit and abort states are final states.

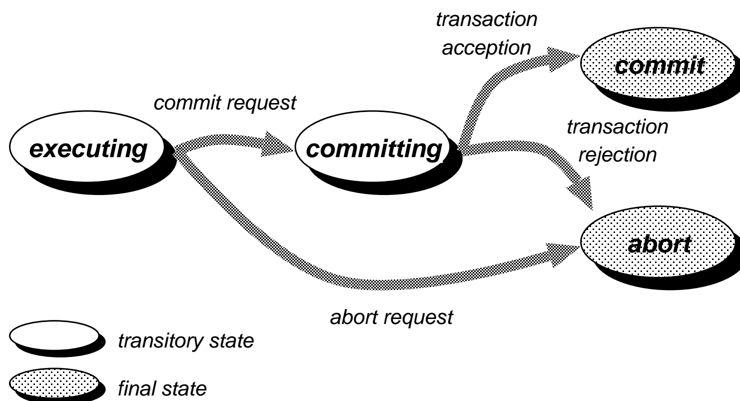

Figure 2: Transaction states

2.3 The Deferred Update Algorithm

We summarise here the principle of the deferred update replication technique. To simplify the presentation, we consider a particular transaction t_m executing in some process p_i. Hereafter, the *readset (RS)* and *writeset (WS)* are the sets of data items that a transaction reads and writes, respectively, during its execution.

1. Transaction t_m is initiated and executed at process p_i. Read and write operations request the appropriate locks, that are locally granted according to the strict two phase locking rule. During this phase, transaction t_m is in the executing state.

2. When transaction t_m requests the commit, t_m passes to the committing state. Its read locks are released (the write locks are maintained by p_i until t_m reaches a final state), and process p_i then triggers the termination protocol for t_m: the updates performed by t_m, as well as its readset and writeset, are sent to all the other processes.

3. As part of the termination protocol, t_m is certified by every process. The certification test guarantees one-copy serializability. The outcome of this test is the commit or abort of t_m,

according to concurrent transactions that executed in other processes. We will come back to this issue in Sections 3 and 4.

4. If t_m passes the certification test, all its write locks are requested and its updates are applied to the database. Hereafter, transaction t_m is represented as T_i. Transactions in the execution state whose locks conflict with T_i's write locks are aborted for T_i's sake.

During the time when a process p is down, p may miss some transactions by not participating in their termination protocol. However, as soon as process p is up again, p catches up with another process that has seen all transactions in the system. This recovery procedure depends on the implementation of the termination protocol (we discuss this issue in Section 5).

As described above, the objective of the termination protocol is twofold: (i) propagating committing transactions to the other processes and (ii) certifying them. The exact way this is done is implementation specific. In a typical deferred update implementation [4], the transaction propagation is performed using an atomic commitment protocol, and the certification test is based on a conflicting rule ([22, 26]). The atomic commitment protocol ensures that all processes receive the transaction, which is committed only if all processes agree to do so, and the conflicting rule leads to abort a transaction if it is involved in any *read/write* or *write/write* conflict.

In the next section we describe a termination protocol based on (i) an atomic broadcast primitive to propagate the transactions, and (ii) a certification test specifically designed to be used with this atomic broadcast primitive. In Sections 4 and 5, we show that this termination protocol provides better throughput and response time than a classical termination protocol based on atomic commitment.

3 A Replication Scheme based on Atomic Broadcast

In the following subsections, we recall the definition of an atomic broadcast primitive, present a deferred update replication algorithm using this primitive and prove it correct.

3.1 Atomic Broadcast

An atomic broadcast primitive enables to send messages to a set of processes, with the guarantee that all processes agree on the *set* of messages delivered and the *order* according to which the messages are delivered [18]. More precisely, atomic broadcast ensures that (i) if some process delivers message m then every correct process delivers m *(uniform atomicity)*; (ii) no two

processes deliver any two messages in different orders *(order)*; and (iii) if a process broadcasts message m and does not fail, then every correct process eventually delivers m *(termination)*.

It is important to notice that the properties of atomic broadcast are defined in terms of message *delivery* and not in terms of message *reception*. Typically, a process first receives a message, then performs some computation to guarantee the atomic broadcast properties, and then delivers the message. The notion of delivery captures the concept of irrevocability (i.e., a process must not forget that it has delivered a message). In the following, we use the expression *delivering a transaction* t_m to mean *delivering the message that contains transaction* t_m. We discuss the implementation of atomic broadcast primitives in Section 5.

3.2 A Termination Protocol based on Atomic Broadcast

We describe now the termination protocol for the deferred update replication technique (Section 2) based on atomic broadcast. On delivering a committing transaction, each process certifies the transaction and, in case of success, commits it. Once the transaction is delivered and certified successfully, it passes to the commit state and its writes are processed.

Figure 3 abstractly presents the main components involved in the termination protocol based on atomic broadcast and the way these components are related to each other.[3] The *Certifier* executes the certification test for an incoming transaction. It receives the transactions delivered by the *Atomic Broadcast* module and cannot change their relative order. On certifying a transaction, the Certifier may ask information to the *Data Manager* about already committed transactions. If the transaction is successfully certified, its write operations are transmitted to the *Lock Manager*. The Lock Manager treats the requests that come from the Certifier sequentially.

In order for a process to certify a committing transaction t_m, it has to know which transactions conflict with t_m. The notion of conflict is defined by the *precedes* relation between transactions and the operations issued by the transactions. A transaction T_j *precedes* another transaction t_m, denoted $T_j \rightarrow t_m$, if T_j committed before t_m started its execution. The relation $T_j \not\rightarrow t_m$ (not $T_j \rightarrow t_m$) means that T_j committed after t_m started its execution. Based on these definitions, we say that a transaction T_j conflicts with t_m if (1) $T_j \not\rightarrow t_m$ and (2) t_m and T_j have conflicting operations.[4]

[3] In a database implementation, these distinctions may be much less apparent, and the modules more tightly integrated [14]. However, for presentation clarity, we have chosen to separate the modules.

[4] Two operations conflict if they are issued by different transactions, access the same data item and at least one of them is a write.

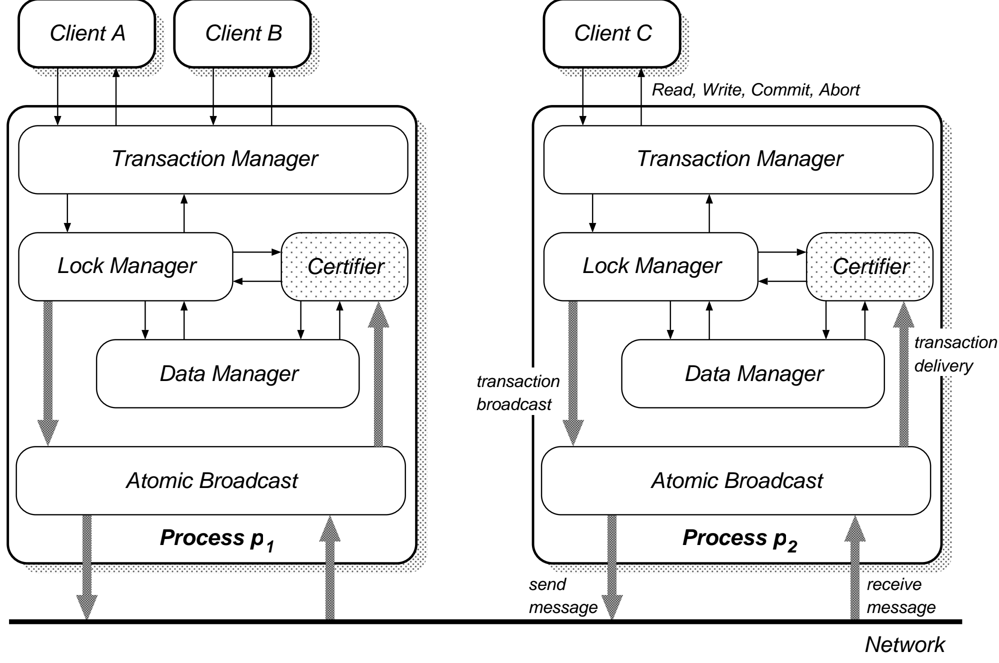

Figure 3: Deterministic execution

More precisely, process p_j performs the following steps after delivering t_m.

1. *Certification test.* Process p_j aborts t_m if there is any committed transaction T_j that conflicts with t_m and that updated data items read by t_m. This can be formally described as follows.

$$Aborted(t_m, \tau', p_j), \tau' > \tau \equiv \left[\begin{array}{c} state(t_m, \tau, p_j) = committing \\ \wedge \\ \exists T_j \in \Pi^{\tau}_{p_j}, T_j \not\rightarrow t_m \\ \wedge \\ WS(T_j) \cap RS(t_m) \neq \emptyset \end{array} \right]$$

2. *Commitment.* If t_m is not aborted, it passes to the commit state (hereafter t_m will be expressed as the committed transaction T_i, where i represents the sequential order of transaction t_m, and $\Pi^{\tau'}_{p_j} = \Pi^{\tau}_{p_j} \cup \{T_i\}$); process p_j tries to grant all its write locks, and processes its updates. There are two cases to consider.

   (a) *There is a transaction t_n in execution at p_j whose read or write locks conflict with T_i's write locks.* In this case t_n is aborted by p_j and, therefore, all t_n's read and write locks are released.[5] More precisely, p_j aborts transaction t_n if the following predicate

[5]If the conflict is due to a write lock held by t_n in p_j, t_n should be aborted because otherwise a multiversion

is evaluated true.

$$Aborted(t_n, \tau'', p_j), \tau'' > \tau' \equiv \left[\begin{array}{c} state(t_n, \tau', p_j) = executing \\ \wedge \\ \exists T_i \in \Pi_{p_j}^{\tau'}, T_i \not\rightarrow t_n \\ \wedge \\ \left(\begin{array}{c} RS(t_n) \cap WS(T_i) \neq \emptyset \\ \vee \\ WS(t_n) \cap WS(T_i) \neq \emptyset \end{array} \right) \end{array} \right]$$

(b) *There is a transaction t_n, that is executed locally at p_j and requested the commit, but is delivered after T_i.* If t_n executed locally at p_j, it has all write locks on the data items it updated. If t_n commits, its writes will supersede T_i's (i.e., the ones that overlap) and, in this case, T_i need neither request these write locks nor process the updates over the database. This is similar to Thomas' Write Rule [26]. If t_n is later aborted (i.e., it does not pass the certification test), the database should be restored to a state without t_n, for instance, by applying T_i's redo log entries to the database.

Given a committing transaction t_m, the set of transactions T_j so that $T_j \not\rightarrow t_m$ (see certification test above) is determined as follows. Given a process p, let $last_p(\tau)$ represent the last delivered and committed transaction in p at local time τ. A transaction t_m that starts its execution in process p at time τ_0 has to be checked at certification time τ_p against the transactions $\{T_{last_p(\tau_0)+1}, \ldots, T_{last_p(\tau_p)}\}$. The information $last_p(\tau_0)$ is broadcast together with the writes of T_i, and $last_p(\tau_p)$ is determined by each process p at certification time.

3.3 Proof of Correctness (Sketch)

Proving the safety of our replication protocol comes down to showing that every history it generates is one-copy serializable. In the following, we give an intuitive idea of the proof. For further details see the appendix.

As each process is provided with the same input (i.e., committing transactions) in the same order, and each process uses the same deterministic certifying algorithm to commit (or abort) transactions, it suffices to show that one process generates only one-copy serializable executions to prove that the overall database system generates only one-copy serializable executions (i.e., to prove the correctness of our algorithm). For the development of the proof, we model our system

mechanism is necessary to keep track of multiple concurrent writes. If the conflict is due to a read lock, the best to be done is abort t_n because with T_i's commit, it is doomed to abort (its reads are too old and it would not pass the certification test later).

as a multiversion database (i.e., transactions executing at a process have their own version of the data items) and use the *Multiversion Graph* theorem of Bernstein et al. [4]. This theorem states that a multiversion history is one-copy serializable if it produces an acyclic multiversion serialization graph.

Consequently, our proof is composed of two parts. In the first part, we define a multiversion serialization graph $MVSG_p$, that just contains committed transactions in p, as a sequence of states $MVSG_p^0, MVSG_p^1, \ldots, MVSG_p^n$ and prove, by induction on the states and using the atomicity and order properties of the atomic broadcast primitive, that every two processes p and q, $p, q \in \Sigma$, produce the same $MVSG^x_{p/q}, x \geq 0$. In the second part, we prove that every $MVSG_p^x$ is acyclic by showing that, for every edge $T_i \rightarrow T_j$ in $MVSG_p^x$, T_i is delivered before T_j. We show that transaction T_i is committed (and so included in $MVSG_p^x$) if T_i does not include any read-from or version-order relation of the form $T_i \rightarrow T_j$ to a previous committed transaction T_j in $MVSG_p$. Using the *Multiversion Graph* theorem, we conclude that our replication protocol guarantees one-copy serializability.

3.4 Liveness Characterisation

The termination property of the atomic broadcast primitive assures that every transaction eventually reaches the committing state in all database processes and, depending on its conflicts with other already committed transactions, is either committed or aborted. If a process is down, it will deliver the transaction as soon as it recovers. This guarantee reflects the liveness property of the algorithm.

In the next section, we discuss the conditions under which a transaction is committed in our algorithm, and we compare these conditions with those of a traditional deferred update replication technique based on atomic commitment.

4 Transaction Throughput

In this section, we first recall the abort/commit conditions for a termination protocol based on atomic commitment. Then we compare the throughput of this solution with a deferred update termination protocol based on atomic broadcast. The throughput measures the transaction commit rate. It is a liveness parameter.

4.1 The Termination Protocol based on Atomic Commit

With an atomic commit implementation of the deferred update termination protocol, no total order knowledge is available for the processes and a conflict has to be resolved by aborting all conflicting transactions. We describe below the termination protocol based on atomic commit.

1. *Certification test.* A process can make a decision on the commit or abort of a transaction t_m when it knows that all the other processes also know about t_m. This requires an interaction among processes that ensures that when a transaction is certified, all committing transaction are taken into account. On certifying transaction t_m, process p_j decides for its abort if there is a transaction t_n that does not precede t_m ($t_n \not\rightarrow t_m$) and with which t_m has conflicting operations. This is formally described as follows.

$$Aborted(t_m, \tau', p_j), \tau' > \tau \equiv \left[\begin{array}{c} state(t_m, \tau, p_j) = committing \\ \wedge \\ \exists t_n, state(t_n, \tau, p_j) = committing \wedge t_n \not\rightarrow t_m \\ \wedge \\ \left(\begin{array}{c} RS(t_m) \cap WS(t_n) \neq \emptyset \\ \vee \\ WS(t_m) \cap RS(t_n) \neq \emptyset \\ \vee \\ WS(t_m) \cap WS(t_n) \neq \emptyset \end{array} \right) \end{array} \right]$$

2. *Commitment.* If t_m is not aborted, it passes to the commit state and p_j tries to grant all its write locks and processes its updates. Transactions that are being locally executed in p_j and have read or write locks on data items updated by t_m are aborted. More formally, the following transactions in execution in p_j are aborted. (Henceforth t_m is represented as the committed transaction T_i.)

$$Aborted(t_n, \tau'', p_j), \tau'' < \tau' \equiv \left[\begin{array}{c} state(t_n, \tau', p_j) = executing \\ \wedge \\ \exists T_i \in \Pi_{p_j}^{\tau'}, T_i \not\rightarrow t_n \\ \wedge \\ \left(\begin{array}{c} RS(t_n) \cap WS(T_i) \neq \emptyset \\ \vee \\ WS(t_n) \cap WS(T_i) \neq \emptyset \end{array} \right) \end{array} \right]$$

In order to certify and commit transactions, the system has to detect transactions that executed concurrently in different processes. For example, in [1] this is done associating *vector clocks* timestamps with local events (e.g., execution or commit of a transaction). These times-

tamps have the property of being ordered if the events are causally related. Each time a process communicates to another, it sends its vector clock.

4.2 Atomic Broadcast *vs.* Atomic Commit: Throughput

The values presented in Figures 4 and 5 were obtained by means of a simulation involving a database with 10000 data items replicated in 5 database processes. Each transaction has a readset and a writeset with 10, different, data items. In Figure 4, each data item has the same probability of being accessed (no hot spots). In Figure 5, 1% of the data items in the database have 10 times more chance of being accessed than the other 99% (i.e., we assume here that 1% of the data items are hot spots). These experiments consider concurrent committing transactions (i.e., transactions that have already requested the commit).

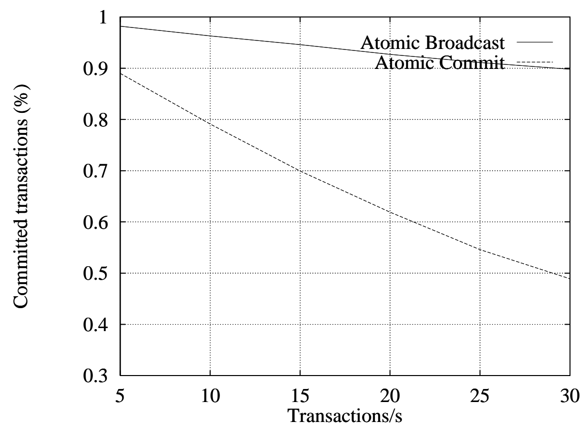

Figure 4: Equiprobable accesses

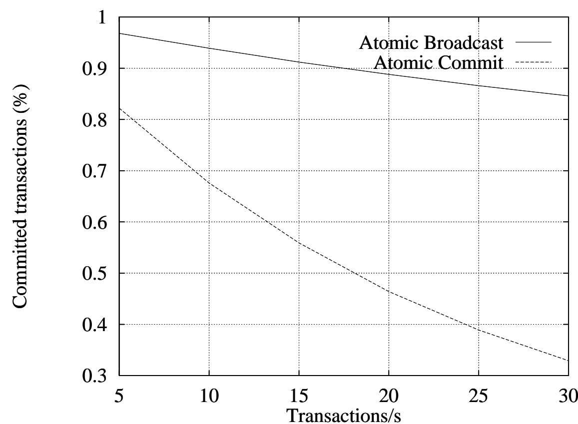

Figure 5: Hot spots

These results report the effects of total ordering committing requests. They show that total order can indeed produce better results. Intuitively, the idea is that without total order, processes have to resolve conflicts by aborting all transactions involved, instead of aborting just some of them. For example, if transactions t_m and t_n conflict, and the processes know that t_m is delivered everywhere before t_n, t_n's abort is enough to resolve the conflict (there is no need to abort t_m). If this order knowledge is not available, aborting just one transaction could create inconsistencies. Some processes might abort t_m while others t_n.[6]

[6]Some systems resolve this problem by requiring that transactions be commutable [13]. In this case, we cannot really talk about conflicting transactions.

5 Response Time

In this section we discuss the implementation of the atomic broadcast primitive. We consider different implementations that correspond to different resilience degrees, and we compare their cost to atomic commit protocols with the same resilience degrees. When comparing the costs of the atomic broadcast and atomic commit implementations, we consider failure free runs because these are the most frequent ones in practice. For the following discussion, we assume that the communication links are reliable (i.e., messages are neither lost nor corrupted).

5.1 Sequencer-based Atomic Broadcast

The implementation described in this section is based on a sequencer process [5, 20]. It uses a centralised approach that distinguishes between a coordinator and participant processes. The coordinator receives messages from any process (step 1 in Figure 6) and broadcasts them to all processes (step 2). The coordinator includes a sequence number in the messages so that all processes deliver the messages in the same order (i.e., according to the sequence numbers).[7] That is, a message number n is just delivered if message number $n-1$ has already been delivered.

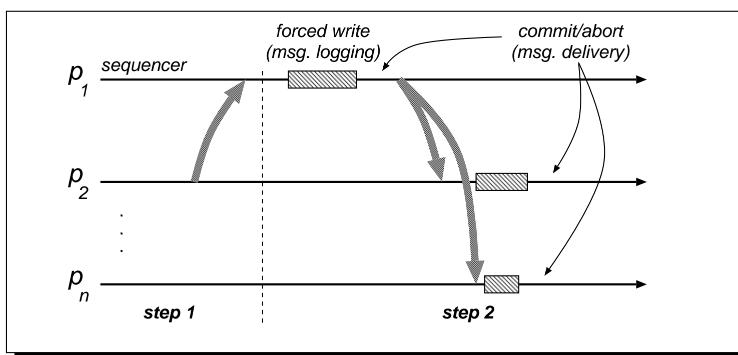

Figure 6: Sequencer-based Atomic Broadcast (failure-free execution)

In the absence of failures, this implementation clearly fulfils the properties of the atomic broadcast primitive. It is blocking because when the coordinator is down, no broadcast is possible. The coordinator has to keep the broadcast messages because if a process fails, and later recovers, it can be brought up to date by asking the coordinator the messages it missed when it was down. Stable messages can be discarded by the coordinator. A message m is stable for a process p when p knows that all the other processes have delivered m.

[7] Note that the sequence number is not necessary if the communication links are FIFO.

The recover of the coordinator is more complicated because, when a failure occurs, some processes may not have received all messages (i.e., some messages may be unstable in the coordinator). Therefore, correctness is just guaranteed if the coordinator *remembers* all broadcast messages (or at least the unstable ones). To solve this problem, before broadcasting a message the coordinator stores it in stable storage. When recovering, the coordinator checks each process and sends the messages that it has not received yet.

This implementation satisfies the atomicity property of the atomic broadcast primitive because a process can just deliver a message if it was sent by the coordinator, and the coordinator keeps all messages in stable storage until they become stable. Processes that are up eventually receive the message from the coordinator and deliver them. Processes that are down contact the coordinator as soon as they recover and receive the missed messages. By logging messages, the coordinator makes sure it will not lose any of them even if it fails. The order property is assured by the unique sequence numbers associated with messages. Once the coordinator receives a message to be broadcast, eventually all correct processes will also receive and deliver it, guaranteeing the termination property.

In a failure free execution with n processes, this implementation requires $n+1$ messages and n forced writes in stable storage (log). Although this implementation offers a good performance, as a disadvantage, it relies on the fact that the sequencer must be a correct process. This requirement is similar to the two phase commit, where the coordinator must be a correct process. In the next sections, we present two implementations of the atomic broadcast that looses this requirement.

5.2 Atomic Broadcast with a Majority Always Up

The atomic broadcast implementation presented in this section was proposed by Chandra and Toueg [7]. It considers that there is always a majority of processes up during the execution of an atomic broadcast, and for each atomic broadcast it is always the same majority. The execution is divided into rounds and one process acts as the coordinator of each round. If no failure (or suspicion of failure) occurs during the execution, the algorithm terminates in one round, otherwise, if the coordinator fails or is suspected, another round is initiated and another predetermined process becomes the coordinator. In an implementation without failures, a process wanting to broadcast a message m sends it to the coordinator (step 1 in Figure 7), that forwards it to all processes (step 2). Each process, on receiving m, answers with an ack to the coordinator (step 3). When the coordinator receives the ack from more than half processes

it sends a control message saying that m can be delivered (step 4). If the coordinator fails or is suspected of having failed, a new coordinator is elected following the rotating coordinator principle. In an execution without failures, this implementation requires $3n + 2$ messages.

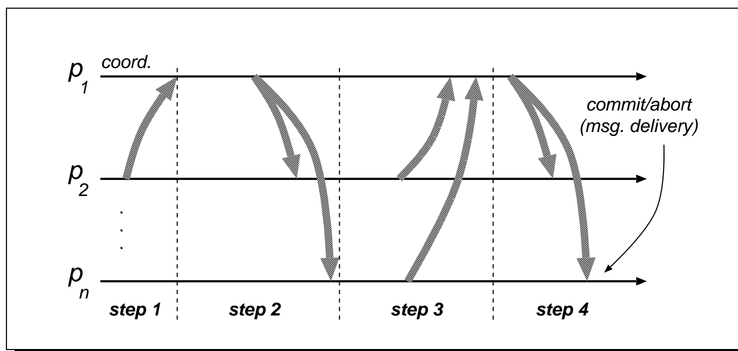

Figure 7: Atomic broadcast with a majority always up (failure-free execution)

5.3 Atomic Broadcast with a Majority Correct

The majority assumption in the implementation of Section 5.2 means that there is at least one process up that has seen all delivered messages in the system. This assumption, however, may be considered too restrictive for a crash-recover model. In the following, we describe a variation of the previous implementation for a system where a majority of processes is correct.

The situation that must be prevented is the one where some processes deliver a message and, before storing it in stable storage, fail and forget what they have delivered. This problem is solved if processes log the messages before delivering them (Figure 8). It guarantees that, on recovering, processes can determine the messages that have already been delivered or are being delivered. This implementation still needs a majority of processes up to deliver a message, but contrary to the previous case, the system resumes normal execution as soon as there is such a majority of processes.[8]

In a failure-free execution with n processes, this implementation requires $3n + 2$ messages and n forced writes (logs).

5.4 Atomic Broadcast *vs.* Atomic Commit: Response Time

The cost of an atomic broadcast has a direct effect on the system's response time: the more efficient the broadcast is implemented, the faster the user receives the result for the request (e.g.,

[8]A full description of this protocol in case of failures is out of the scope of this paper.

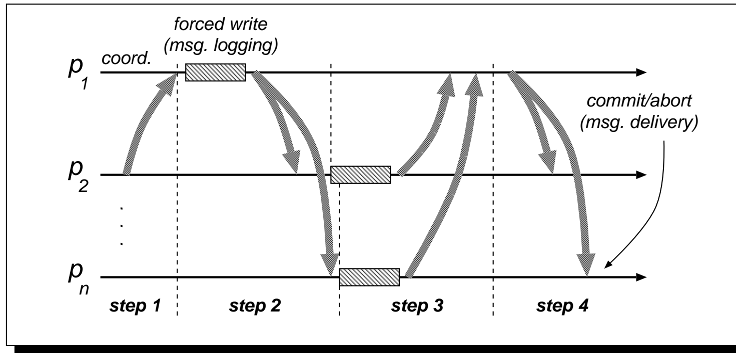

Figure 8: Atomic broadcast with a majority correct (failure-free execution)

transaction commit). In the following, we compare the atomic broadcast implementation given in the previous sections to traditional atomic commit implementations.[9]

Figure 9 presents a qualitative analysis of termination protocols. The characteristics of the two-phase commit (2PC) and three-phase commit (3PC) protocols come from [4]. It only considers forced log writes. The Light 3PC is a variation of the 3PC, that assumes a majority of processes up. This satisfies the write voting rule and therefore assures consistency in a replicated database system [11].

Protocol	Resilience	Number of communication steps	Cost of the termination protocol
Sequencer based ABP	blocking	2	$(n+1)$ messages $+n$ forced write
Two Phase Commit	blocking	3	$3n$ messages $+n$ forced writes
ABP (majority up)	non-blocking	4	$4n$ messages
Light 3PC	non-blocking	5	$5n$ messages
ABP (majority correct)	non-blocking	4	$4n$ messages $+n$ forced writes
Three Phase Commit	non-blocking	5	$5n$ messages $+n$ forced writes

Figure 9: Cost of the termination protocol

Figures 10 and 11 depict quantitative results. These measures have been obtained with Sparc 20 workstations (running Solaris 2.3), an Ethernet network using the TCP protocol, and transactions (messages and logs) of 1024 bytes. The measures convey an average time to deliver a transaction message.

The results in Figures 10 and 11 show that atomic broadcast protocols have a better performance than atomic commit, when compared under the same degree of resilience.

[9]Many optimisations have been proposed for atomic commit protocols [17]. In this section, we use the most representative implementations of atomic commit protocols. Optimisations for these implementations can also be done for atomic broadcast primitives.

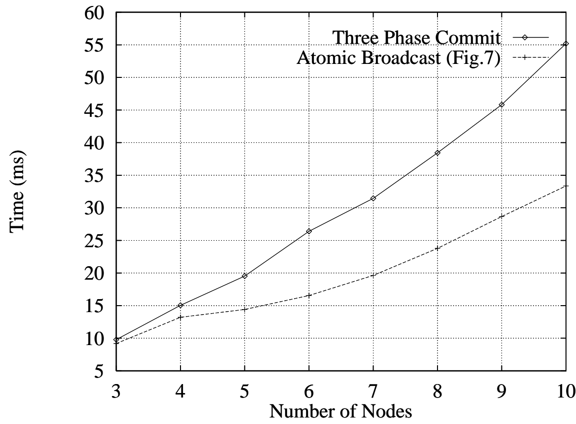

Figure 10: Termination without logging

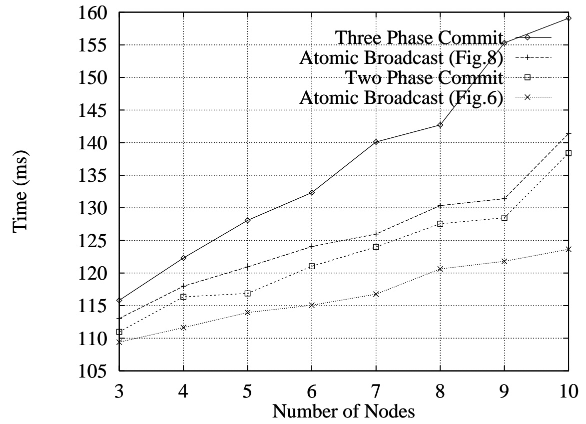

Figure 11: Termination with logging

6 Related Work

The limitations of traditional atomic commitment protocols in replicated contexts have been recognised by several authors. We summarise some alternative propositions below.

In [15], the authors point out the fact that atomic commitment leads to abort transactions in situations where a single replica manager crashes. They propose a variation of the three phase commit protocol [25] that commits transactions as long as a majority of replica managers are up.

In [10], a class of *epidemic* replication protocols is proposed as an alternative to traditional replication protocols. However, solutions based on epidemic replication end up being either a case of lazy propagation where consistency is relaxed, or solved with semantic knowledge about the application [19]. In [2], a replication protocol based on the deferred update model is presented. Transactions that execute at the same process share the same data items, using locks to solve local conflicts. The protocol is based on a variation of the three phase commit protocol to certificate and terminate transactions.

It is only recently that atomic broadcast has been considered as a possible candidate to support replication, as termination protocols. Schiper and Raynal [24] pointed out some similarities between the properties of atomic broadcast and static transactions (transactions whose operations are known in advance). Atomic broadcasting static transactions was also addressed in [21]. In [23], we present a reordering technique, based on atomic broadcast, that allows for a greater transaction throughput in replicated databases.

In [1], a family of protocols for the management of replicated database based on the immediate and the deferred models are proposed. These protocols use an atomic broadcast primitive that has the same characteristics as the primitive described in this paper but its behaviour in

case of failures is not discussed. The immediate update replication consists in atomic broadcasting every write operation. The authors describe two possible implementations for the deferred update replication. In the first alternative, two atomic broadcasts are necessary to commit a transaction. In the second alternative, transactions execute locally, deferring all their writes to commit time. At commit time, transactions are broadcast and all write locks are requested. This approach has two drawbacks: (a) it requires a sort of multiversion mechanism to deal with the writes during transaction execution (if a transaction writes a data item, a later read should reflect this write) and (b) transactions in execution may be aborted by other transactions that executed locally with them, as conflicts are all dealt with at commit time (even the ones that happen locally).

Amir et al. [3] also utilise atomic broadcast to implement replicated databases. However, the scheme proposed considers that clients submit individual object operations rather than transactions.

7 Concluding Remarks

In a recent paper, Cheriton and Skeen expressed their scepticism about the adequateness of atomic broadcast to support replicated databases [8]. The reasons that were raised were (1) atomic broadcast replication schemes consider individual operations, whereas in databases, operations are gathered inside transactions, (2) atomic broadcast does not guarantee uniform atomicity (e.g., a server could deliver a message and crash, with the other servers not even receiving it), whereas uniform atomicity is fundamental in distributed databases (all processes, even those that have later crashed, must agree to commit or not a transaction), and (3) atomic broadcast is usually considered in a crash-stop process model (i.e., a process that crashes never recovers), whereas in databases processes are supposed to recover after a crash. In this paper, we have considered an atomic broadcast primitive in a crash-recovery model that guarantees uniformity, i.e., if one process delivers a transaction, all the others also deliver it. We have shown how that primitive can efficiently be used to propagate transaction control information in a deferred update model of replication. The choice for that replication model was not casual, as some recent research has shown that immediate update models are inviable due to the nature of the applications or the side effects of synchronisation [13].

Indirectly, this paper points out the fact that existing database replication protocols are built on top of distributed database systems, using mechanisms that were developed to deal with distributed information, but not necessarily designed with replication in mind. A flagrant

example of this is the atomic commitment mechanism, which we have shown can be favourably replaced by an atomic broadcast primitive, providing better throughput and response time. Our performance figures help dis-mystify a common (mis)belief that total order atomic broadcast primitives are too expensive, when compared to traditional transactional primitives, and so inappropriate for high performance systems. As already stated in [27], we also believe that replication can be successfully applied for performance, and this can be achieved without sacrificing consistency (e.g., as in [9]) or making semantic assumptions about the transactions (e.g., as in [13, 19]).

It is important however to notice that the replication scheme based on atomic broadcast presented in this paper is based on two important assumptions: (1) processes certify and commit transactions sequentially, and (2) the database is fully replicated. The first assumption can be bypassed since concurrency between transactions that come from the *Certifier* is allowed if the transactions have disjoint write sets. The second assumption about a fully replicated database is reasonable in traditional closed information systems, but is inappropriate for an open system with a large number of nodes or a large database. In such systems, replication can only be partial (i.e., processes store only subsets of the database), and the extend to which atomic broadcast can be useful in this context is an open issue.

References

[1] D. Agrawal, A. El Abbadi, and R. Steinke. Epidemic algorithms in replicated databases. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson, Arizona, 12–15 May 1997.

[2] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. Technical report, University of California at Santa Barbara and Swiss Federal Institute of Technology, 1996.

[3] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and efficient replication using group communication. Technical Report CS94-20, Hebrew University of Jerusalem, 1994.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[6] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the ISIS Toolkit.* IEEE Press, 1994.

[7] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[8] D. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, December 1993.

[9] D. J. Delmolino. Strategies and techniques for using Oracle 7 replication. Technical report, Oracle Corporation, 1995.

[10] A. Demers et al. Epidemic algorithms for replicated database maintenance. In Fred B. Schneider, editor, *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, BC, Canada, August 1987. ACM Press.

[11] D. K. Gifford. Weighted voting for replicated data. In *ACM SOSP 7, Pacific Grove CA*, December 1979.

[12] J. N. Gray. Notes on data base operating systems. In *Springer Verlag (Heidelberg, FRG and NewYork NY, USA) LNCS, 'Operating Systems, an Advanced Course', Bayer,Graham, Seegmuller(eds)*, volume 60. 1978.

[13] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996.

[14] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[15] R. Guerraoui, R. Oliveira, and A. Schiper. Atomic updates of replicated data. In *EDCC, European Dependable Computing Conference*, Taormina, Italy, 1996. LNCS 1050.

[16] R. Guerraoui and A. Schiper. Software based replication for fault tolerance. *IEEE Computer*, 30(4), April 1997.

[17] R. Gupta, J. Haritsa, and K. Ramamritham. Revisiting commit processing in distributed database systems. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona,, May 1997.

[18] V. Hadzilacos and S. Toueg. *Distributed Systems, 2ed*, chapter 3, Fault-Tolerant Broadcasts and Related Problems. Addison Wesley, 1993.

[19] H. V. Jagadish, I. S. Mumick, and M. Rabinovich. Scalable versioning in distributed databases with commuting updates. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.*, pages 520–531. IEEE Computer Society Press, April 1997.

[20] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the amoeba distributed operating system. In *11th International Conference on Distributed Computing Systems*, pages 222–230, Washington, D.C., USA, May 1991. IEEE Computer Society Press.

[21] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[22] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[23] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *16th IEEE Symposium on Reliable Distributed Systems*, Durham, USA, October 1997. IEEE Computer Society Press.

[24] A. Schiper and M. Raynal. From group communication to transaction in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[25] D. Skeen. Nonblocking commit protocols. In Y. Edmund Lien, editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 133–142, Ann Arbor, Michigan, April 1981. ACM, New York.

[26] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2):180–209, June 1979.

[27] P. Triantafillou. High availability is not enough. In J.-F. Paris and H. G. Molina, editors, *Proceedings of the Second Workshop on the Management of Replicated Data*, pages 40–43, Monterey, California, November 1992. IEEE Computer Society Press.

Appendix: Correctness Proof

For the following proofs, $C(H)_p$ is a history derived from the system history H, just containing operations of committed transactions involving data items stored at p. $C(H)_p$ is a multiversion history: transactions that execute at p use one version of the data items and transactions that execute at other processes use different versions of them. We denote $w_i[x_i]$ a write by T_i (as writes generate new data versions, the subscript in x for data writes is always the same as the one in T) and $r_i[x_j]$ a read by T_i of data item x_j.

As shown in [4], normal serialization graphs (SG) alone are not powerful enough to prove multiversion histories correct (i.e., one copy serializable). For this reason, we will use the multiversion formalism of [4] to prove that all the histories produced by our algorithm are correct. This formalism employs a multiversion serialization graph ($MVSG(C(H)_p)$ or $MVSG_p$ for short) and consists in showing that such a graph is acyclic. We denote $MVSG_p^x$ a particular state of the multiversion serialization graph for process p. Whenever a transaction is committed, the multiversion serialization graph passes from one state $MVSG_p^x$ into another $MVSG_p^{x+1}$.

Our proof is divided into two parts. Lemma 1 shows that, by the properties of the atomic broadcast primitive and the determinism of the certification test, all processes $p \in \Sigma$ eventually construct the same $MVSG_p^x$, $x \geq 0$. Lemmas 2 and 3 show that every $MVSG_p^x$ is acyclic.

Lemma 1 *If one process $p \in \Sigma$ constructs a multiversion serialization graph $MVSG_p^x, x \geq 0$, then every process p eventually constructs the same multiversion serialization graph $MVSG_p^x$.*

PROOF: The proof is by induction. *Basic step:* when the database is initialised, every process p_i has the same empty multiversion serialization graph $MVSG_{p_i}^0$. *Inductive step (assumption):* assume that every process p_i that has constructed a multiversion serialization graph $MVSG_{p_i}^x$ has constructed the same $MVSG_{p_i}^x$. *Inductive step (conclusion).* Consider T_i the transaction whose processing generates, from $MVSG_{p_i}^x$, a new multiversion serialization graph $MVSG_{p_i}^{x+1}$ in one process p_i. In order to do so, process p_i must deliver the message m containing T_i, certify it and commit it. By the order property of the atomic broadcast primitive, every process p_i that delivers a message after installing $MVSG_{p_i}^x$, delivers message m, and, by the atomicity property, if one process delivers message m, then every process delivers m. To certify T_i, every process p_i takes into account the transactions that it has already locally committed (i.e., the transactions in $MVSG_{p_i}^x$). Thus, based on the same local state ($MVSG_{p_i}^x$), the same input (message m), and the same (deterministic) certification algorithm, if one process constructs a new multiversion serialization graph $MVSG_p^{x+1}$, then every process eventually constructs the same $MVSG_p^{x+1}$.$\square$

We show next that every history $C(H)_p$ produced by a process p has an acyclic $MVSG_p$ and, therefore, is $1SR$. Given a multiversion history $C(H)_p$ and a version order $\ll$, the multiversion serialization graph for $C(H)_p$ and $\ll$, $MVSG_p$, is a serialization graph with read-from and version order edges. A read-from relation $T_i \rightarrow T_j$ is defined by an operation $r_j[x_i]$. There are two cases where a version-order relation $T_i \rightarrow T_j$ is in $MVSG_p$: (a) for each $r_k[x_j]$, $w_j[x_j]$ and $w_i[x_i]$ in $C(H)_p$ (i, j, and k are distinct) and $x_i \ll x_j$, and (b) for each $r_i[x_k]$, $w_k[x_k]$ and $w_j[x_j]$ in $C(H)_p$ and $x_k \ll x_j$. The version order is defined by the delivery order of the transactions. Formally, a version order can be expressed as follows: $x_i \ll x_j$ iff $delivery(T_i) < delivery(T_j)$ and T_i, $T_j \in MVSG_p$.

To prove that $C(H)_p$ has an acyclic multiversion serialization graph ($MVSG_p$) we show that the read-from and version-order relations in $MVSG_p$ follow the order of delivery of the committed transactions in $C(H)_p$. That is, if $T_i \rightarrow T_j \in MVSG_p$ then $delivery(T_i) < delivery(T_j)$.

Lemma 2 *If there is a* read-from *relation* $T_i \rightarrow T_j \in MVSG_p$ *then* $delivery(T_i) < delivery(T_j)$.

PROOF: A read-from relation $T_i \rightarrow T_j$ is in $MVSG_p$ if $r_j[x_i] \in C(H)_p, i \neq j$. For a contradiction, assume that $delivery(T_j) < delivery(T_i)$. This can only happen if T_j reads uncommitted data from T_i. Since transactions just read committed data (assured by the strict two phase locking rule) this situation is not possible. □

Lemma 3 *If there is a* version-order *relation* $T_i \rightarrow T_j \in MVSG_p$ *then* $delivery(T_i) < delivery(T_j)$.

PROOF: According to the definition of version-order edges, there are two cases to consider.

(a) Let $r_k[x_j]$, $w_j[x_j]$ and $w_i[x_i]$ be in $C(H)_p$ and $x_i \ll x_j$: we include $T_i \rightarrow T_j$ in $MVSG_p$ and have to show that $delivery(T_i) < delivery(T_j)$. If $x_i \ll x_j$ then x was updated by T_i and after updated by T_j. Since updates follow the order of delivery (steps 2a and 2b), the only way this can happen is by having $delivery(T_i) < delivery(T_j)$.

(b) Let $r_i[x_k]$, $w_k[x_k]$ and $w_j[x_j]$ be in $C(H)_p$ and $x_k \ll x_j$: we include $T_i \rightarrow T_j$ in $MVSG_p$ and have to show that $delivery(T_i) < delivery(T_j)$. For a contradiction, assume that $delivery(T_j) < delivery(T_i)$. On certifying T_i, process p takes T_j into account, since T_j is in the commit state and T_i and T_j are concurrent. This would lead p to abort T_i ($WS(T_j) \cap RS(T_i) \neq \emptyset$), contradicting the hypotheses that T_i is in $C(H)_p$. □

Theorem 1 *Every history H produced by our algorithm is $1SR$.*

PROOF: By Lemmas 2 and 3, every process p produces a serialization graph $MVSG_p^x$ such that every edge $T_i \rightarrow T_j \in MVSG_p^x$ satisfies the relation $delivery(T_i) < delivery(T_j)$. Hence, every process produces an acyclic multiversion serialization graph $MVSG_p^x$. By Lemma 1, all processes construct the same $MVSG_p^x$. By the *Multiversion Graph* theorem of [4], every history produced by our replication algorithm is 1SR. □