The Database State Machine Approach

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 present the database state machine approach, a new way to deal with database replication in a cluster of servers. This approach relies on a powerful atomic broadcast primitive to propagate transactions between database servers, and no atomic commitment is necessary. Transaction commit is based on a certification test, and abort rate is reduced by the reordering certification test. The approach is evaluated using a detailed simulation model that shows the scalability of the system and the benefits of the reordering certification test.

1 Introduction

Software replication is considered a cheap way to increase data availability when compared to hardware based techniques [16]. However, designing a replication scheme that provides synchronous replication (i.e., all copies are kept consistent) at good performance is still an active area of research both in the database and in the distributed systems community. For example, commercial databases are typically based on the asynchronous replication model that tolerates inconsistencies among replicas [30, 12].

This paper investigates a new approach for synchronous database replication on a cluster of database servers (e.g., a group of workstations) connected by a local area network. The replication mechanism presented is based on the state machine approach [28], and differs from traditional replication mechanisms in that it does not handle replication using distributed transactional mechanisms, such as atomic commitment [13, 6]. The state machine approach was proposed as a general mechanism for dealing with replication, however no previous study has addressed its use in the domain of a cluster of database servers.

Our database state machine is based on the *deferred update* technique. According to the deferred update technique, transactions are processed locally at one database 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 its alternative, *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) better support for fault tolerance, by simplifying server recovery (i.e., missing updates may be treated by the communication module as lost messages), and (e) lower deadlock rate, by eliminating distributed deadlocks [12].

The main drawback of the deferred update technique is that the lack of synchronisation during transaction execution may lead to large transaction abort rates. We show how the database state machine approach can be used to reduce the transaction abort rate by using a reordering certification test, which looks for possible serializable executions before deciding to abort a transaction.

We have developed a simulation model of the database state machine and conducted several experiments with it. The results obtained by our simulation model allowed us to assess some important points about the system, like its scalability, and the effectiveness of the reordering technique. Particularly, in the former case, it shows which parts of the system are more prone to become resource bottlenecks. Evaluations of the reordering technique have shown that transaction aborts due to serialization problems can be reduced from 20% to less than 5% in clusters of 8 database servers.

The paper is organised as follows. In Section 2, we introduce the replicated database model where our results are based on, and the two main concept used in our approach. In Section 3, we recall the principle of the deferred update replication technique. In Section 4, we show how to transform deferred update replication into a state machine. An optimisation of this approach that reduces the number of aborted transactions is described in Section 5. In Section 6, we present the simulation tool we used to evaluate the protocols discussed in the paper and draw some conclusions about them. In Section 7 we discuss related, and Section 8 concludes the paper.

2 System Model and Definitions

In this section, we describe the system model and the two main concepts involved in our approach, that is, those of state machine, and atomic broadcast. The state machine approach delineates the replication strategy, and the atomic broadcast constitutes a sufficient order mechanism to implement a state machine.

2.1 Database and Failures

We consider a system composed of a set Σ of sites. Sites in Σ communicate through message passing, and do not have access to a shared memory or to a common clock. To simplify the presentation, we assume the existence of a discrete global clock, even if sites do not have access to it. The range of the clock's ticks is the set of natural numbers. The set Σ is divided into two disjoint subsets: a subset of client sites, denoted Σ_C, and a subset of database sites, denoted Σ_D. Hereafter, we consider that $\Sigma_C = \{c_1, c_2, \ldots, c_m\}$, and $\Sigma_D = \{s_1, s_2, \ldots, s_n\}$. Each database site plays the role of a replica manager, and each one has a full copy of the database.

Sites fail independently and only by crashing (i.e., we exclude Byzantine failures [22]). We also assume that every database site eventually recovers after a crash. If a site is able to execute requests at a certain time τ (i.e., the site did not fail or failed but recovered) we say that the site is *up* at time τ. Otherwise the site is said to be *down* at time τ. For each database site, we consider that there is a time after which the site is forever up.[1]

Transactions are sequences of read and write operations followed by a commit or abort operation. A transaction is called a query (or read-only) if it does not contain any write operations, otherwise it is called an update transaction. Transactions, denoted t_a, t_b, and t_c, are submitted by client sites, and executed by database sites. Our correctness criterion for transaction

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

execution is one-copy serializability (1SR) [6].

2.2 The State Machine Approach

The state machine approach, also called active replication, is a non-centralised replication coordination technique. Its key concept is that all replicas (or database sites) receive and process the same sequence of requests. Replica consistency is guaranteed by assuming that when provided with the same input (e.g., an external request) each replica will produce the same output (e.g., state change). This assumption implicitly implies that replicas have a deterministic behaviour.

The way requests are disseminated among replicas can be decomposed into two requirements [28]:

1. **Agreement.** Every non-faulty replica receives every request.
2. **Order.** If a replica first processes request req_1 before req_2, then no replica processes request req_2 before request req_1.

The order requirement can be weakened if some semantic information about the requests is known. For example, if two requests commute, that is, independently of the order they are processed they produce the same final states and sequence of outputs, then it is not necessary that order be enforced among the replicas for these two requests.

2.3 Atomic Broadcast

An atomic broadcast primitive enables to send messages to database sites, with the guarantee that all database sites agree on the *set* of messages delivered and the *order* according to which the messages are delivered [17] (implementation details are discussed in Section 6.2). Atomic broadcast is defined by the primitives $broadcast(m)$ and $deliver(m)$. More precisely, atomic broadcast ensures the following properties.

1. **Agreement.** If a database site delivers message m then every database site delivers m.
2. **Order.** No two database sites deliver any two messages in different orders.
3. **Termination.** If a database site broadcasts message m and does not fail, then every database site eventually delivers m.

The total order induced on the *deliver* is represented by the relation $\prec$. If message m_1 is delivered before message m_2, then $deliver(m_1) \prec deliver(m_2)$.

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 database site 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 database site must not forget that it has delivered a message). From Section 2.2, it should be clear that atomic broadcast is sufficient to guarantee the correct dissemination of requests to replicas acting as state machines.

3 Deferred Update Replication

The deferred update replication technique [6] is a way of dealing with requests in a replicated database environment, and it will be the base for the database state machine presented in Section 4. In this section, we first recall the principle of the deferred update replication technique, and then provide a detailed characterisation of it.

3.1 Deferred Update Replication Principle

In the deferred update replication technique, transactions are locally executed at one database site, and during their execution, no interaction between other database sites occurs (see Figure 1). Transactions are locally synchronised at database sites according to some concurrency control mechanism [6]. However, we assume throughout the paper that the concurrency control mechanism used by every database site to local synchronise transactions is the strict two phase locking rule. When a client requests the transaction commit, the transaction's updates (e.g., the redo log records) and some control structures are propagated to all database sites, where the transaction will be certified and, if possible, committed. This procedure, starting with the commit request, is called *termination protocol*. The objective of the termination protocol is twofold: (i) propagating transactions to database sites, and (ii) certifying them.

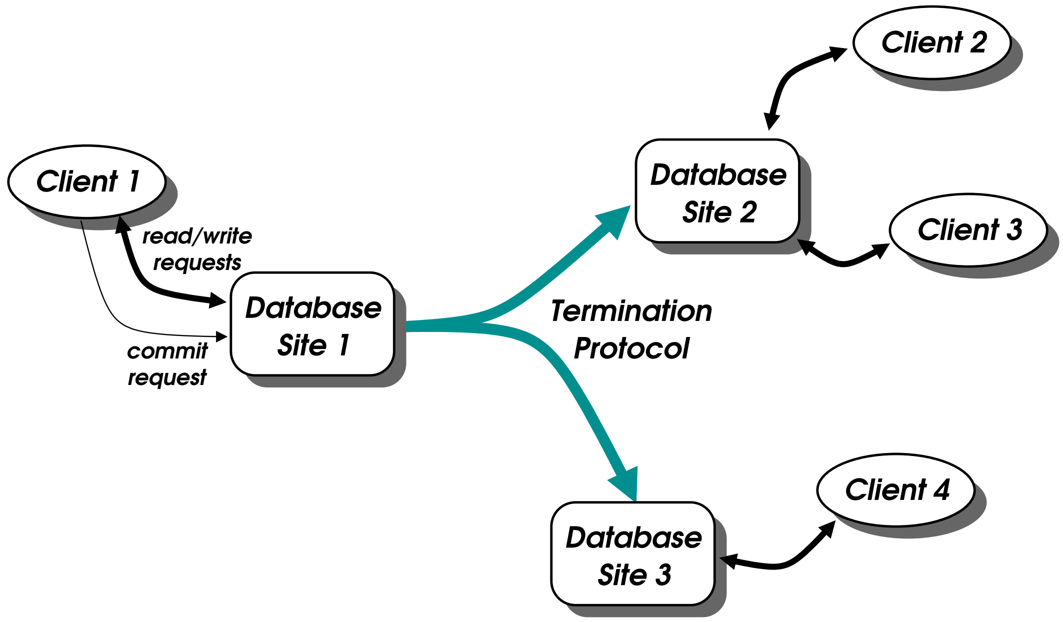

Figure 1: Deferred update technique

The certification test aims at ensuring one-copy serializability. It decides to abort a transaction if the transaction's commit would lead the database to an inconsistent state (i.e., non-serializable). For example, consider two concurrent transactions, t_a and t_b, that are executed at different database sites, and that update a common data item. On requesting the commit, if t_a arrives before t_b at the database site s_i but after t_b at the database site s_j $(i \neq j)$, both transactions t_a and t_b might have to be aborted, since otherwise, site s_i would see transaction t_a before transaction t_b, and site s_j would see transaction t_b before transaction t_a, violating one-copy serializability.

While a database site s_i is down, s_i may miss some transactions by not participating in their termination protocol. However, as soon as database site s_i is up again, s_i catches up with another database site that has seen all transactions in the system.

3.2 Transaction States

During its processing, a transaction passes through some well-defined states (see Figure 2). The transaction starts in the *executing state*, when its read and write operations are locally executed at the database site where it was initiated. When the client that initiates the transaction requests the commit, the transaction passes to the *committing state* and is sent to the other database sites. A transaction received by a database site is also in the committing state, and it remains in the committing state until its fate is known by the database site (i.e. *commit* or *abort*). The different states of a transaction t_a at a database site s_i are denoted $Executing(t_a, s_i)$, $Committing(t_a, s_i)$, $Committed(t_a, s_i)$, and $Aborted(t_a, s_i)$. The executing and committing states are transitory states, whereas the committed and aborted states are final states.

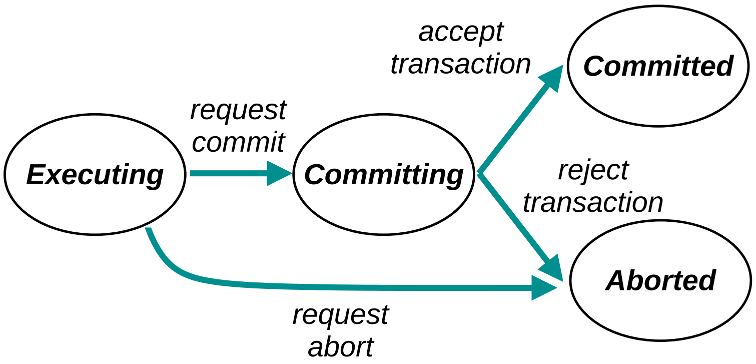

Figure 2: Transaction states

3.3 General Algorithm

We describe next a general algorithm for the deferred update replication technique. To simplify the presentation, we consider a particular client c_k that sends requests to a database site s_i in behalf of a transaction t_a.

1. Read and write operations requested by the client c_k are executed at s_i according to the strict two phase locking (strict 2PL) rule. From the start until the commit request, transaction t_a is in the executing state.

2. When c_k requests t_a's commit, t_a is immediately committed if it is a read-only transaction. If not, t_a passes to the committing state, and the database site s_i triggers the termination protocol for t_a: the updates performed by t_a, as well as its readset and writeset, are sent to all database sites.

3. Eventually, every database site s_j certifies t_a. The certification test takes into account every transaction t_b known by s_j that *conflicts* with t_a (see Section 3.4). It is important that all database sites reach the same common decision on the final state of t_a, which may require some coordination among database sites. Such coordination can be achieved, for example, by means of an atomic commit protocol, or, as it will be shown in Section 4, by using the state machine approach.

4. If t_a is serializable with the previous committed transactions in the system (e.g., t_a passes the certification test), all its updates will be applied to the database. Transactions in the execution state at each site s_j holding locks on the data items updated by t_a are aborted.

5. The client c_k receives the outcome for t_a from site s_i as soon as s_i can determine whether t_a will be committed or aborted. The exact moment this happens depends on how the termination protocol is implemented, and will be discussed in Section 4.

Queries do not execute the certification test, nevertheless, they may be aborted during their execution due to local deadlocks and by non-local committing transactions when granting their write locks. The algorithm presented above can be modified in order to reduce or completely avoid aborting read-only transactions. For example, if queries are pre-declared as so, once an update transaction passes the certification test, instead of aborting a query that holds a read lock on a data item it wants to update, the update transaction waits for the query to finish and release the lock. In this case, update transactions have the highest priority in granting write locks, but they wait for queries to finish. Read-only transactions can still be aborted due to deadlocks, though. However, multiversion data item mechanisms can prevent queries from being aborted altogether. In [26], read-only transactions are executed using a fixed view (or version) of the database, without interfering with the execution of update transactions.

3.4 Transaction Dependencies

In order for a database site s_i to certify a committing transaction t_a, s_i must be able to tell which transactions at s_i conflict with t_a. A transaction t_b *conflicts* with t_a if t_a and t_b have *conflicting operations* and t_b does not *precede* t_a. Two operations conflict if they are issued by different transactions, access the same data item and at least one of them is a write. The precede relation between two transactions t_a and t_b is defined as follows. (a) If t_a and t_b execute at the same database site, t_b precedes t_a if t_b enters the committing state before t_a. (b) If t_a and t_b execute at different database sites, say s_i and s_j, respectively, t_b precedes t_a if t_b commits at s_i before t_a enters the committing state at s_i. Let $site(t)$ be the identity of the database site where transaction t was executed, and $committing(t)$ and $commit(t)_{s_j}$ be the events that represent, respectively, the request for commit and the commit of t at s_j. The event $committing(t)$ only happens at the database site s_i where t was executed, and the event $commit(t)_{s_j}$ happens at every database site s_j. We formally define that transaction t_b precedes transaction t_a, denoted $t_b \rightarrow t_a$, as

$$t_b \rightarrow t_a = \begin{cases} committing(t_b) \xrightarrow{e} committing(t_a) & site(t_a) = site(t_b), \\ commit(t_b)_{site(t_a)} \xrightarrow{e} committing(t_a) & \text{otherwise}, \end{cases}$$

where $\xrightarrow{e}$ is the local (total) order relation for the events $committing(t)$ and $commit(t)_{s_j}$. The relation $t_a \not\rightarrow t_b$ (t_a not $\rightarrow t_b$) establishes that t_a does not precede t_b.[2]

The deferred update replication does not require any distributed locking protocol to synchronise transactions during their execution. Therefore, network bandwidth is not consumed by synchronising messages, and there are no distributed deadlocks. However, transactions may be aborted due to conflicting accesses. In the next sections, we show that the deferred update replication technique can be implemented using the state machine approach, and that this approach allows optimisations that can reduce the transaction abortion due to conflicting accesses.

4 The Database State Machine Approach

The deferred update replication technique can be implemented as a state machine. In this section, we discuss the details of this approach, and the implications to the way transactions are processed.

[2] Since local events are totally ordered at database sites, $t_a \not\rightarrow t_b$ is equivalent to $t_b \rightarrow t_a$.

4.1 The Termination Protocol as a State Machine

The termination protocol presented in Section 3 can be turned into a state machine as follows. Whenever a client requests a transaction's commit, the transaction's updates, its readset and writeset (or, for short, the transaction) are atomically broadcast to all database sites. Each database site will behave as a state machine, and the agreement and order properties required by the state machine approach are ensured by the atomic broadcast primitive.

The database sites, upon delivering and processing the transaction, should eventually reach the same state. In order to accomplish this requirement, delivered transactions should be processed with certain care. Before delving deeper into details, we describe the database modules involved in the transaction processing. Figure 3 abstractly presents such modules and the way they are related to each other.[3] Transaction execution, as described in Section 3, is handled by the *Transaction Manager*, the *Lock Manager*, and the *Data Manager*. The *Certifier* executes the certification test for an incoming transaction. It receives the transactions delivered by the *Atomic Broadcast* module. On certifying a transaction, the Certifier may ask information to the data manager about already committed transactions (e.g., logged data). If the transaction is successfully certified, its write operations are transmitted to the Lock Manager, and once the write locks are granted, the updates can be performed.

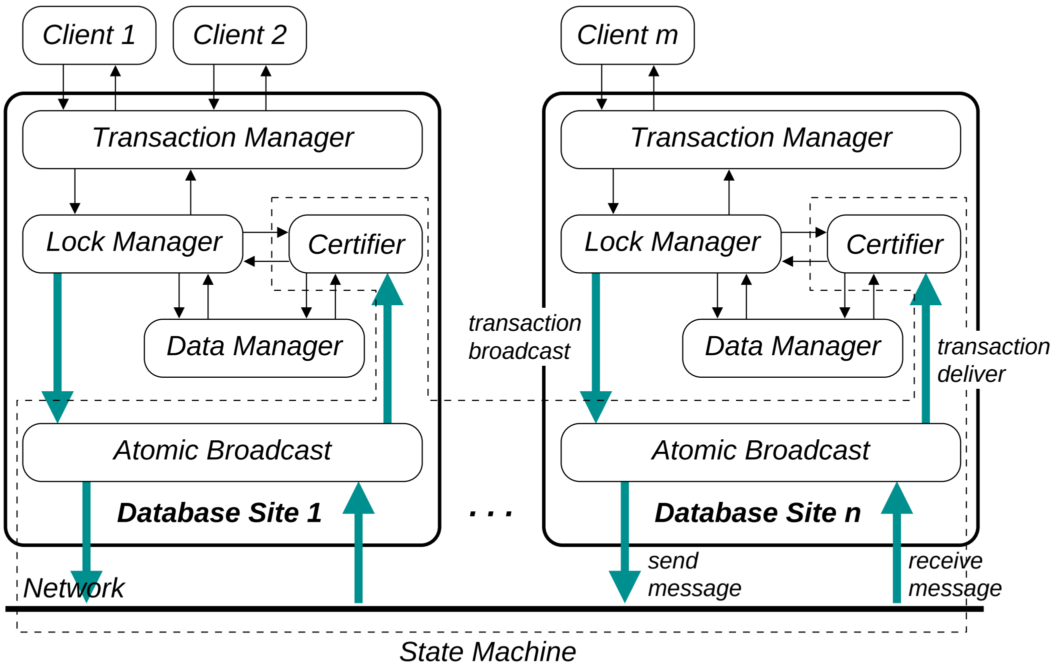

Figure 3: Termination protocol based on atomic broadcast

To make sure that each database site will reach the same state after processing committing transactions, each certifier has to guarantee that write-conflicting transactions are applied to the database in the same order (since transactions whose writes do not conflict are commutable). This is the only requirement from the certifier, and can be attained if the certifier ensures that write-conflicting transactions grant their locks following the order they are delivered. This requirement is straightforward to implement, nevertheless, it reduces concurrency in the certifier.

[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.

4.2 The Termination Algorithm

The procedure executed on delivering the request of a committing update transaction t_a is detailed next. For the discussion that follows, the *readset (RS)* and the *writeset (WS)* are sets containing the identifiers of the data items read and written by the transaction, respectively, during its execution. Transaction t_a was executed at database site s_i. Every database site s_j, after delivering t_a, performs the following steps.

1. *Certification test.* Database site s_j commits t_a (i.e., t_a passes from the committing state to the committed state) if there no committed transaction t_b that conflicts with t_a. Since only committed transactions are taken into account to certify t_a, the notion of conflicting operations defined in Section 3.4 is weakened, and just write operations performed by committed transactions and read operations performed by t_a are considered.

   The certification test is formalised next as a condition for a state transition from the committing state to the committed state (see Figure 2).

$$Committing(t_a, s_j) \rightsquigarrow Committed(t_a, s_j) \equiv \left[\begin{array}{l} \forall t_b,\ Committed(t_b, s_j): \\ \\ t_b \rightarrow t_a \vee (WS(t_b) \cap RS(t_a) = \emptyset) \end{array} \right]$$

   The condition for a transition from the committing state to the aborted state is the complement of the right side of the expression shown below.

   Once t_a has been certified by database site s_i, where it was executed, s_i can inform t_a's outcome to the client that requested t_a's execution.

2. *Commitment.* If t_a is not aborted, it passes to the commit state, all its write locks are requested, and once granted, its updates are performed. On granting t_a's write locks, there are three cases to consider.

   (a) *There is a transaction t_b in execution at s_j whose read or write locks conflict with t_a's writes.* In this case t_b is aborted by s_j, and therefore, all t_b's read and write locks are released.

   (b) *There is a transaction t_b, that was executed locally at s_j and requested the commit, but has not been delivered yet.* Since t_b executed locally at s_j, t_b has its write locks on the data items it updated. If t_b commits, its writes will overwrite t_a's (i.e. the ones that overlap) and, in this case, t_a need neither request these write locks nor process the updates over the database. This is similar to Thomas' Write Rule [31]. However, if t_b is later aborted (i.e., it does not pass the certification test), the database should be restored to a state without t_b, for example, by applying t_a's redo log entries to the database.

   (c) *There is a transaction t_b that has passed the certification test and has granted its write locks at s_j, but it has not released them yet.* In this case, t_a waits for t_b to finish its updates and release its write locks.

An important aspect of the termination algorithm presented above is that the atomic broadcast is the only form of interaction between database sites. The atomic broadcast properties guarantee that every database site will certify a transaction t_a using the same set of preceding transactions. It remains to be shown how each database site builds such a set. If transactions t_a and t_b execute at the same database site, this can be evaluated by identifying transactions that execute at the same site (e.g., each transaction carries the identity of the site where it was initiated) and associating local timestamps to the committing events of transactions.

If t_a and t_b executed at different sites, this is done as follows. Every transaction commit event is timestamped with the order of deliver of the transaction (the atomic broadcast ensures that each database site associates the same timestamps to the same transactions). Each transaction t has a *committing*(t) field associated to it that stores the timestamp of the last locally committed transaction when t passes to the committing state, and is broadcast to all database sites. When certifying t_a, all committed transactions that have been delivered by the database site with commit timestamp greater than *committing*(t) take part in the set of preceding transactions, used to certify t_a.

4.3 Algorithm Correctness

The database state machine algorithm is proved correct using the multiversion formalism of [6]. Although we do not explicitly use multiversion databases, our approach can be seen as so, since replicas of a data item located at different database sites can be considered as different versions of the data item.

We first define $C(H)_{s_i}$ as a multiversion history derived from the system history H, just containing operations of committed transactions involving data items stored at s_i. We denote $w_a[x_a]$ a write by t_a (as writes generate new data versions, the subscript in x for data writes is always the same as the one in t) and $r_a[x_b]$ a read by t_a of data item x_b.

The multiversion formalism employs a multiversion serialization graph ($MVSG(C(H)_{s_i})$ or $MVSG_{s_i}$ for short) and consists in showing that all the histories produced by the algorithm have a multiversion serialization graph that is acyclic. We denote $MVSG^k_{s_i}$ a particular state of the multiversion serialization graph for database site s_i. Whenever a transaction is committed, the multiversion serialization graph passes from one state $MVSG^k_{s_i}$ into another $MVSG^{k+1}_{s_i}$.

We exploit the state machine characteristics to structure our proof in two parts. In the first part, Lemma 1 shows that, by the properties of the atomic broadcast primitive and the determinism of the certifier, every database site $s_i \in \Sigma_D$ eventually constructs the same $MVSG^k_{s_i}$, $k \geq 0$. In the second part, Lemmas 2 and 3 show that every $MVSG^k_{s_i}$ is acyclic.

Lemma 1 *If a database site $s_i \in \Sigma_D$ constructs a multiversion serialization graph $MVSG^k_{s_i}, k \geq 0$, then every database s_j eventually constructs the same multiversion serialization graph $MVSG^k_{s_j}$.*

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

We show next that every history $C(H)_{s_i}$ produced by a database site s_i has an acyclic $MVSG_{s_i}$ and, therefore, is $1SR$ [6]. Given a multiversion history $C(H)_{s_i}$ and a version order $\ll$, the multiversion serialization graph for $C(H)_{s_i}$ and $\ll$, $MVSG_{s_i}$, is a serialization graph with read-from and version order edges. A read-from relation $t_a \hookrightarrow t_b$ is defined by an operation $r_b[x_a]$. There are two cases where a version-order relation $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$: (a) for each

$r_c[x_b]$, $w_b[x_b]$ and $w_a[x_a]$ in $C(H)_{s_i}$ (a, b, and c are distinct) and $x_a \ll x_b$, and (b) for each $r_a[x_c]$, $w_c[x_c]$ and $w_b[x_b]$ in $C(H)_{s_i}$ and $x_c \ll x_b$. The version order is defined by the delivery order of the transactions. Formally, a version order can be expressed as follows: $x_a \ll x_b$ iff $deliver(t_a) \prec deliver(t_b)$ and $t_a, t_b \in MVSG_{s_i}$.

To prove that $C(H)_{s_i}$ has an acyclic multiversion serialization graph ($MVSG_{s_i}$) we show that the read-from and version-order relations in $MVSG_{s_i}$ follow the order of delivery of the committed transactions in $C(H)_{s_i}$. That is, if $t_a \hookrightarrow t_b \in MVSG_{s_i}$ then $deliver(t_a) \prec deliver(t_b)$.

Lemma 2 *If there is a* read-from *relation $t_a \hookrightarrow t_b \in MVSG_{s_i}$ then $deliver(t_a) \prec deliver(t_b)$.*

PROOF: A read-from relation $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$ if $r_b[x_a] \in C(H)_{s_i}, a \neq b$. For a contradiction, assume that $deliver(t_b) \prec deliver(t_a)$. If t_a and t_b were executed at different database sites, by the time t_b was executed, t_a had not been committed at $site(t_b)$, and thus, t_b could not have read a value updated by t_a. If t_a and t_b were executed at the same database site, t_b must have read uncommitted data from t_a, since t_a had not been committed yet. However, this contradicts the strict two phase locking rule. □

Lemma 3 *If there is a* version-order *relation $t_a \hookrightarrow t_b \in MVSG_{s_i}$ then $deliver(t_a) \prec deliver(t_b)$.*

PROOF: According to the definition of version-order edges, there are two cases to consider. (1) Let $r_c[x_b]$, $w_b[x_b]$ and $w_a[x_a]$ be in $C(H)_{s_i}$ (a, b and c distinct), and $x_a \ll x_b$, which implies $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$. It follows from the definition of version-order that $deliver(t_a) \prec deliver(t_b)$. (2) Let $r_a[x_c]$, $w_c[x_c]$ and $w_b[x_b]$ be in $C(H)_{s_i}$, and $x_c \ll x_b$, which implies $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$, and have to show that $deliver(t_a) \prec deliver(t_b)$. For a contradiction, assume that $deliver(t_b) \prec deliver(t_a)$. From the certification test, when t_a is certified, either $t_b \rightarrow t_a$ or $WS(t_b) \cap RS(t_a) = \emptyset$. But since $x \in RS(t_a)$, and $x \in WS(t_b)$, it must be that $t_b \rightarrow t_a$.

Assume that t_a and t_b were executed at the same database site. By the definition of precedence (Section 3.4), t_b requested the commit before t_a (that is, $committing(t_b) \xrightarrow{e} committing(t_a)$). However, t_a reads x from t_c, and this can only happen if t_b updates x before t_c, that is, $x_b \ll x_c$, contradicting that $x_c \ll x_b$. A similar argument follows for the case where t_a and t_b were executed at distinct database sites. □

Theorem 1 *Every history H produced by the database state machine algorithm is $1SR$.*

PROOF: By Lemmas 2 and 3, every database site s_i produces a serialization graph $MVSG^k_{s_i}$ such that every edge $t_a \hookrightarrow t_b \in MVSG^k_{s_i}$ satisfies the relation $deliver(t_a) \prec deliver(t_b)$. Hence, every database site s_i produces an acyclic multiversion serialization graph $MVSG^k_{s_i}$. By Lemma 1, every database site s_i constructs the same $MVSG^k_{s_i}$. By the *Multiversion Graph* theorem of [6], every history produced by database state machine algorithm is 1SR. □

5 The Reordering Certification Test

Transactions running without any synchronisation between database sites may lead to high abort rates. In this section, we show how the certification test can be modified such that more transactions pass the certification test, and thus, do not abort.

5.1 General Idea

The reordering certification test [24] is based on the observation that the serial order in which transactions are committed does not need to be the same order in which transactions are delivered to the certifier. The idea is to dynamically build a serial order (that does not necessarily follow the delivery order) in such a way that less aborts are produced. By being able to reorder a transaction t_a to a position other than the one t_a is delivered, the reordering protocol increases the probability of committing t_a.

The atomic broadcast based termination protocol augmented with the reordering feature differs from the atomic broadcast based termination protocol presented in Section 4 in the way the certification test is performed for committing transactions (see Figure 4). The certifier distinguishes between committed transactions already applied to the database and committed transactions in the *Reorder List*. The Reorder List contains committed transactions whose write locks have been granted but whose updates have not been applied to the database yet, and thus, have not been seen by transactions in execution. The bottom line is that transactions in the Reorder List may change their relative order.

The Reorder List has a pre-determined threshold, called *Reorder Factor*, that limits the number of transactions it contains. Whenever the Reorder Factor is reached, the leftmost transaction t_a in the Reorder List is removed, its updates are applied to the database, and its write locks are released. If no transaction in the Reorder List is waiting to acquire a write lock just released by t_a, the corresponding data item is available to executing transactions. The reordering technique reduces the number of aborts, however, introduces some data contention since data items remain blocked longer. This tradeoff was indeed observed by our simulation model (see Section 6.3).

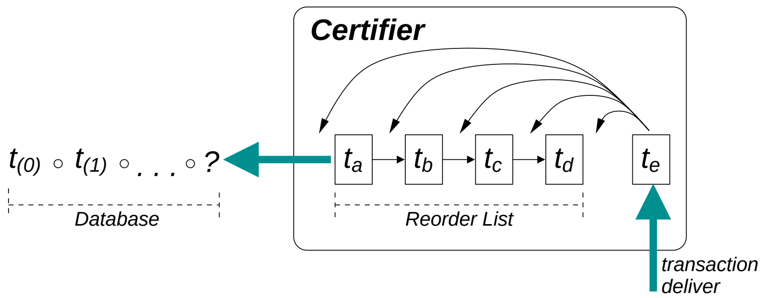

Figure 4: Reorder technique (reorder factor = 4)

5.2 The Termination Protocol based on Reordering

Let $database_{s_i} = t_{(0)} \circ t_{(1)} \circ \ldots \circ t_{(last_{s_i}(\tau))}$ be the sequence containing all transactions on database site s_i, at time τ, that have passed the certification test augmented with the reordering technique (order of delivery plus some possible reordering). The sequence $database_{s_i}$ includes transactions that have been applied to the database and transactions in the Reorder List. We define $pos(t)$ the position transaction t has in $database_{s_i}$, and extend below the termination protocol described in Section 4.2 to include the reordering technique.

1. *Certification test.* Database site s_j commits t_a if there is a position in which t_a can be included in the Reorder List. Transaction t_a can be included in a certain position p in the Reorder List if the following conditions are true.

   (a) There is no transaction t_b in the Reorder List, $pos(t_b) < p$, that conflicts with t_a. That is, for all transactions t_b in the Reorder List, $pos(t_b) < p$, either t_b precedes t_a, or t_b has not updated any data item that t_a has read (this is essentially the certification test described in Section 3.3).

(b) There is no transaction t_b in the Reorder List, $pos(t_b) \geq p$, that satisfies the conditions: (b.1) t_b does not precede t_a, or t_a has not read any data item written by t_b. In which case, t_a can come before t_b in $database_{s_i}$, since t_a did not read any data value that was only written after its execution (by t_b), and (b.2) t_a did not update any data item read by t_b.

The certification test with reordering is formalised in the following as a state transition from the committing state to the committed state.

$$Committing(t_a, s_j) \rightsquigarrow Committed(t_a, s_j) \equiv$$

$$\left[\begin{array}{l} \exists \text{position } p \text{ in the Reorder List s.t. } \forall t_b,\ Committed(t_b, s_j): \\ pos(t_b) < p \Rightarrow t_b \rightarrow t_a \vee WS(t_b) \cap RS(t_a) = \emptyset\ \wedge \\ pos(t_b) \geq p \Rightarrow \left(\begin{array}{c} (t_b \not\rightarrow t_a \vee WS(t_b) \cap RS(t_a) = \emptyset) \\ \wedge \\ WS(t_a) \cap RS(t_b) = \emptyset \end{array} \right) \end{array}\right]$$

The condition for a transition from the committing state to the aborted state is the complement of the right side of the expression shown below.

2. *Commitment.* If t_a passes the certification test it is included in the Reorder List: all transaction on the right side of p, including p, are shifted one position to the right, and t_a is included, assuming now position p. If, with the inclusion of t_a, the Reorder List reaches the Reorder Factor threshold, the leftmost transaction in Reorder List is removed and its updates are applied to the database.

5.3 Algorithm Correctness

From Lemma 1, every database site builds the same multiversion serialization graph. It remains to show that all the histories produced by every database site using reordering have a multiversion serialization graph that is acyclic, and, therefore, 1SR.

We redefine the version-order relation $\ll$ for the termination protocol based on reordering as follows: $x_a \ll x_b$ iff $pos(t_a) < pos(t_b)$ and $t_a, t_b \in MVSG_{s_i}$.

Lemma 4 *If there is a* read-from *relation* $t_a \hookrightarrow t_b \in MVSG_{s_i}$ *then* $pos(t_a) < pos(t_b)$.

PROOF: A read-from relation $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$ if $r_b[x_a] \in C(H)_{s_i}, a \neq b$. For a contradiction, assume that $pos(t_b) < pos(t_a)$. The following cases are possible: (a) t_b was delivered and committed before t_a, and (b) t_b was delivered and committed after t_a but reordered to a position before t_a.

In case (a), it follows that t_b reads uncommitted data (i.e., x) from t_a, which is not possible: if t_a and t_b executed at the same database site, reading uncommitted data is avoided by the strict 2PL rule, and if t_a and t_b executed at different database sites, t_a's updates are only seen by t_b after t_a's commit. In case (b), from the certification test augmented with reordering, when t_b is certified, we have that $(t_a \not\rightarrow t_b \vee WS(t_a) \cap RS(t_b) = \emptyset) \wedge WS(t_b) \cap RS(t_a) = \emptyset$ evaluates true. (Being t_b the committing transaction, the indexes a and b have been inverted, when compared to expression given in the previous section.) Since t_b reads-from t_a, $WS(t_a) \cap RS(t_b) \neq \emptyset$, and so, it must be that $t_a \not\rightarrow t_b$. If t_a and t_b executed at the same database site, $t_a \not\rightarrow t_b$ implies $committing(t_b) \xrightarrow{e} committing(t_a)$. However, this is only possible if t_b reads x from t_a before t_a commits, contradicting the strict 2PL rule. If t_a and t_b executed at different database sites,

$t_a \not\rightarrow t_b$ implies $commit(t_a)_{site(t_b)} \overset{e}{\not\rightarrow} committing(t_b)$, and so, t_b passed to the committing state before t_a committed at $site(t_b)$, which contradicts the fact that t_b reads from t_a, and concludes the Lemma. □

Lemma 5 *If there is a* version-order *relation* $t_a \hookrightarrow t_b \in MVSG_s$ *then* $pos(t_a) < pos(t_b)$.

PROOF: According to the definition of version-order edges, there are two cases of interest. (1) Let $r_c[x_b]$, $w_b[x_b]$, and $w_a[x_a]$ be in $C(H)_{s_i}$ (a, b and c distinct), and $x_a \ll x_b$, which implies $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$. It follows from the definition of version-order that $pos(t_a) < pos(t_b)$. (2) Let $r_a[x_c]$, $w_c[x_c]$, and $w_b[x_b]$ be in $C(H)_{s_i}$ (a, b and c distinct), and $x_c \ll x_b$, which implies $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$. We show that $pos(t_a) < pos(t_b)$. There are two situations to consider.

(a) t_c and t_b have been committed when t_a is certified. Assume for a contradiction that $pos(t_b) < pos(t_a)$. From the certification test, we have that either $t_b \rightarrow t_a$ or $WS(t_b) \cap RS(t_a) = \emptyset$. Since $x \in WS(t_b)$ and $x \in RS(t_a)$, $WS(t_b) \cap RS(t_a) \neq \emptyset$, and so, it must be that $t_b \rightarrow t_a$. However, t_a reads x from t_c and not from t_b, which can only happen if $x_b \ll x_c$, a contradiction.

(b) t_c and t_a have been committed when t_b is certified. Assume for a contradiction that $pos(t_b) < pos(t_a)$. From the certification test, we have that $(t_a \not\rightarrow t_b \vee WS(t_a) \cap RS(t_b) = \emptyset) \wedge WS(t_b) \cap RS(t_a) = \emptyset$ evaluates true, which leads to a contradiction since $x \in WS(t_b)$ and $x \in RS(t_a)$, and thus, $WS(t_b) \cap RS(t_a) \neq \emptyset$. □

Theorem 2 *Every history H produced by the database state machine algorithm augmented with the reordering technique is $1SR$.*

PROOF: By Lemmas 4 and 5, every database site s_i produces a serialization graph $MVSG^k_{s_i}$ such that every edge $t_a \hookrightarrow t_b \in MVSG^k_{s_i}$ satisfies the relation $pos(t_a) < pos(t_b)$. Hence, every database site produces an acyclic multiversion serialization graph $MVSG^x_s$. By Lemma 1, every database site s_i constructs the same $MVSG^k_{s_i}$. By the *Multiversion Graph* theorem of [6], every history produced by the reordering algorithm is 1SR. □

6 Simulation Model

The simulation model we have developed abstracts the main components of a replicated database system (our approach is similar to [3]). In this section, we describe the simulation model and analyse the behaviour of the database state machine approach using the output provided by the simulation model.

6.1 Database Model and Settings

Every database site is modelled as a processor, some data disks, and a log disk as local resources. The network is modelled as a common resource shared by all database sites. Each processor is shared by a set of execution threads, a terminating thread, and a workload generator thread (see Figure 5). All threads have the same priority, and resources are allocated to threads in a first-in-first-out basis. Each execution thread executes one transaction at a time, and the terminating thread is responsible for doing the certification. The workload generator thread creates transactions at the database site. Execution and terminating threads at a database site share the database data structures (e.g., lock table).

Committing transactions are delivered by the terminating thread and then certified. If a transaction passes the certification test, its write locks are requested and its updates are

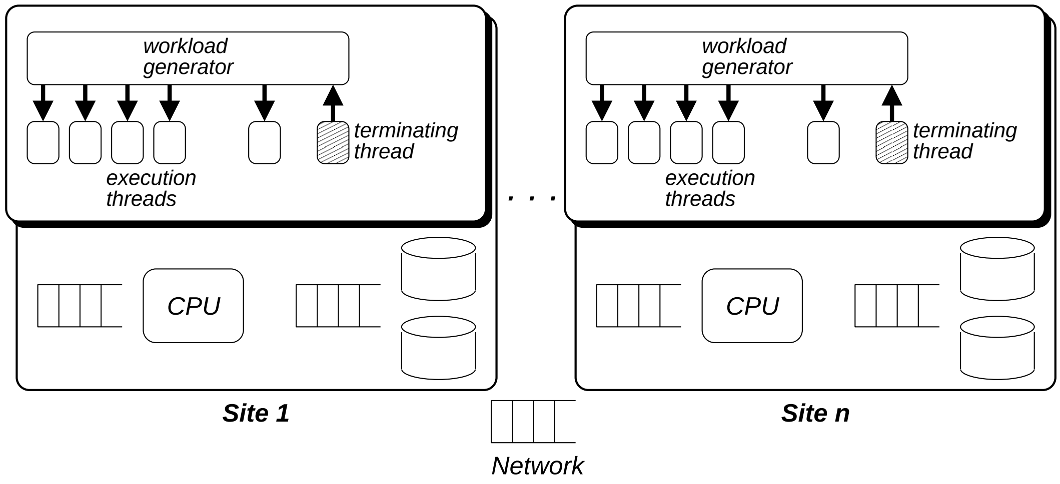

Figure 5: Simulation model

performed. However, once the terminating thread acquires the transaction's write locks, it makes a log entry for this transaction (with its writes) and assigns an execution thread to execute the transaction's updates over the database. This releases the terminating thread to treat the next committing transaction.

The parameters considered by our simulation model with the settings used in the experiments are shown in Figure 6. The workload generator thread creates transactions and assigns them to executing threads according to the profile described (percentage of *update transactions*, percentage of *writes in update transactions*, and number of operations). We have chosen a relative small *database size* in order to reach data contention quickly and avoid extremely long simulation runs that would be necessary to obtain statistically significant results.

We use a closed model, that is, each terminated transaction (committed or aborted) is replaced by a new one. Aborted transactions are sent back to the workload generator thread, and some time later resubmitted at the same database process. The *multiprogramming level* determines the number of executing threads at each database process. Local deadlocks are detected with a timeout mechanism: transactions are given a certain amount of time to execute (*transaction timeout*), and transactions that do not reach the committing state within the timeout are aborted.

Processor activities are specified as a number of instructions to be performed. The settings are an approximation from the number of instructions used by the simulator to execute the operations. The certification test is efficiently implemented by associating to each database item a version number [3]. Each time a data item is updated by a committing transaction, its version number is incremented. When a transaction first reads a data item, it stores the data item's version number (this is the transaction read set). The certification test for a transaction consists thus in comparing each entry in the transaction's read set with the current version of the corresponding data item. If all data items read by the transaction are still current, the transaction passes the certification test. We consider that version numbers are stored in main memory. The reordering test (Section 5.2) is more complex, since it requires handling read sets and write sets of transactions in the reorder list. The *control data size* contains the data structures necessary to perform the certification test (e.g., readset and writeset). Atomic broadcast settings are described in the next section.

Database parameters		Processor parameters	
Database size (data items)	2000	Processor speed	100 MIPS
Number of servers (n)	1..8	Execute an operation	2800 ins
Multiprogramming level (MPL)	8	Certify a transaction	5000 ins
Data item size	2 KB	Reorder a transaction	15000 ins
Transaction parameters		**Disk parameters (Seagate ST-32155W)**	
Update transactions	10%	Number of data disks	4
Writes in update transactions	30%	Number of log disks	1
Number of operations	5..15	Miss ratio	20%
Transaction timeout	0.5 sec	Latency	5.54 msec
Reorder factor	$0, n, 2n, 3n, 4n$	Transfer rate (Ultra-SCSI)	40 MB/sec
General parameters		**Communication parameters**	
Control data size	1 KB	Atomic Broadcasts per second	$\infty, 180, 800/n$
		Communication overhead	12000 ins

Figure 6: Simulation model parameters

6.2 Atomic Broadcast Implementation

The literature on atomic broadcast algorithms is abundant (e.g., [7], [9], [8], [11], [23], [33], [5]), and the multitude of different models (synchronous, asynchronous, etc.) and assumptions about the system renders any fair comparison difficult. However, known atomic broadcast algorithms can be divided into two classes, according to scalability issues. An atomic broadcast algorithm is scalable, and belongs to the first class, if the number of atomic broadcasts that can be executed per time unit in the system does not depend on the number of sites that delivery the messages atomically broadcast. If the number of atomic broadcasts that can be executed per time unit decreases with the number of database sites, the atomic broadcast algorithm is not scalable, and belongs to the second class.

The first class, to which we refer as k-abcast algorithms, has a constant deliver time, independent of the number of sites that deliver the messages. Thus, the k factor determines the number of atomic broadcasts executed per time unit. The second class is referred to as k/n-abcast algorithms, where n is the number of sites that deliver the messages. In this case, the more sites are added, the more time it takes to execute an atomic broadcast, and so, the number of atomic broadcast per time unit executed in the system decreases exponentially with the number of sites.

As a reference to the scalability classes, we also define an atomic broadcast *time zero*, that is, an atomic broadcast that delivers messages instantaneously. This is referred to as a ∞-abcast, since a time zero atomic broadcast algorithm would allow in theory an infinite number of atomic broadcasts executed per time unit.

The values chosen for k in Figure 6 are an approximation based on experiments with SPARC 20 workstations running Solaris 2.3 and an FDDI network (100Mb/s) using the UDP transport protocol with messages of 20 Kbytes. Each time a site executes an atomic broadcast, it incurs in some *communication overhead.*

6.3 Experiments and Results

In the following, we discuss the experiments we conducted and the results obtained with the simulation model. Each point plotted in the graphs has a confidence interval of 95%, and was determined from a sequence of simulations, each one containing 100000 submitted transactions. In order to remove initial transients [19], only after the first 1000 transactions had been submitted, the statistics started to be gathered.

In the following, we analyse update and read-only transactions separately, although the values presented were observed in the same simulations (i.e., all simulations contain update and read-only transactions).

Update Transactions Throughput. The experiments shown in Figures 7 and 8 evaluate the effect of the atomic broadcast algorithm on the transaction throughput. In these experiments, each cluster of database sites processed as many transactions as possible, that is, transaction throughput was only limited by the resources available. Figure 7 shows the number of update transactions submitted, and Figure 8 the number of committed update transactions. As can be seen in Figure 7, the choice of a particular atomic broadcast algorithm is not relevant for a cluster with less than five database sites: whatever the atomic broadcast algorithm, the transaction throughput increases linearly with the number of database sites. This happens because until four database sites, all three configurations are limited by the same resource, namely, local data disks. Since the number of data disks increases linearly with the number of database sites, transaction throughput also increases linearly. For clusters with more than four database sites, contention is determined differently for each configuration. For the ∞-*abcast* based execution, after five database sites, contention is caused by the certification procedure. For the k-*abcast* and k/n-*abcast* based executions, contention is caused by the network (the limit being 180 and $800/n$ atomic broadcasts per second, respectively).

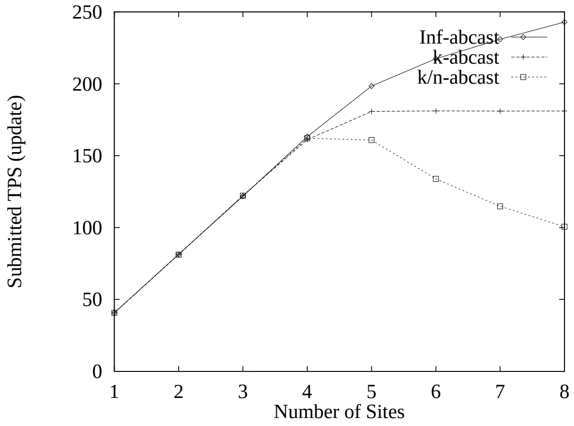

Figure 7: Submitted TPS (update)

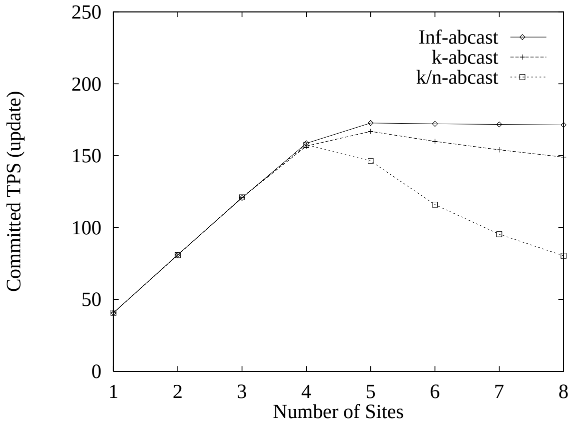

Figure 8: Committed TPS (update)

It is indeed expected that after a certain point (system load), the terminating thread would become a bottleneck, and transaction certification critical. However, as can be seen in Figure 8, this only happens for the ∞-*abcast* based execution (about 170 update transactions per second), since for the others, the network limits the execution before that point is reached. It can also be seen in Figure 8 that for the k-abcast based execution, although the number of transactions submitted per second for clusters with more than four sites is constant, the aborts increase (with the number of database sites). This is due to the fact that the more database sites, the more transactions are executed under an optimistic concurrency control and thus, the higher the probability that a transaction aborts.

Queries Throughput. Figures 9 and 10 show submitted and committed queries per second in the system. The curves in Figure 9 have the same shape as the ones in Figure 7. This is so because the simulator enforces a constant relation between submitted queries and submitted update transactions (see Figure 6, *update transactions* parameter). Update transactions are determined by resource contention, and so, queries present the same behaviour. This could be avoided if some executing threads were only assigned to queries, however, the relation between submitted queries and update transactions would be determined by internal characteristics of the system and not by an input parameter, as we would like it to be. Queries are only aborted during their execution to solve (local) deadlocks they are involved in, or on behalf of committing update transactions that have passed the certification test and are requesting their write locks (Section 4.2). As shown in Figure 9 and 10, submitted and committed queries, for all atomic broadcast based executions, are very close to each other, which amounts to a small abort rate.

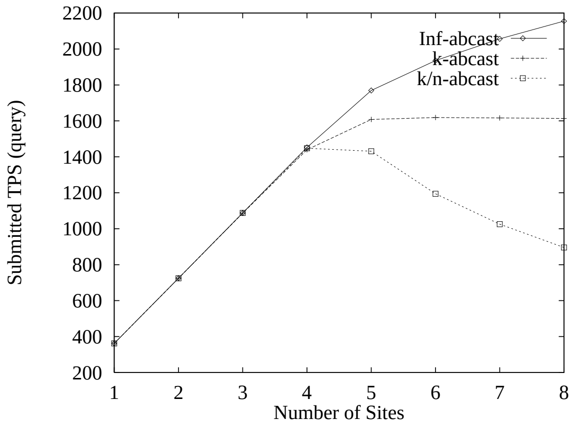

Figure 9: Submitted TPS (query)

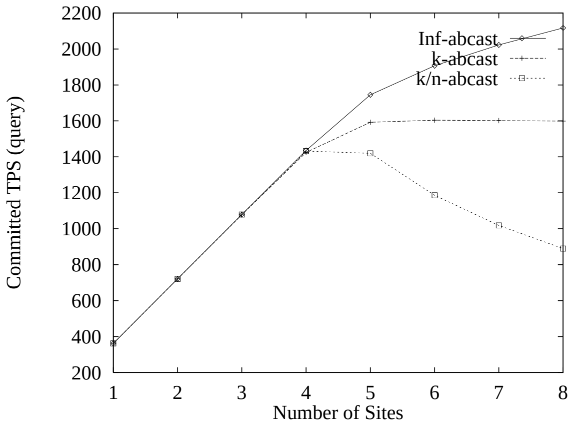

Figure 10: Committed TPS (query)

Response Time. The next graphs (Figures 11 and 12) depict the response time of queries and update transactions. Figure 11 presents the response time for the executions shown in Figures 7 and 8. It can be seen that the price paid for the higher throughput presented by the ∞-*abcast* based execution, when compared to the k-*abcast* based execution, is a higher response time. However, as it was expected, when all atomic broadcast based executions process the same number of transactions per second (Figure 12), the ∞-*abcast* based execution is faster. Queries have the same response time for all atomic broadcast based executions of the simulation.

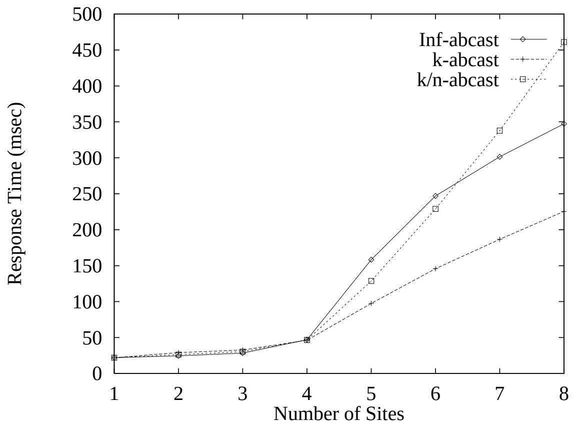

Figure 11: Response time (update)

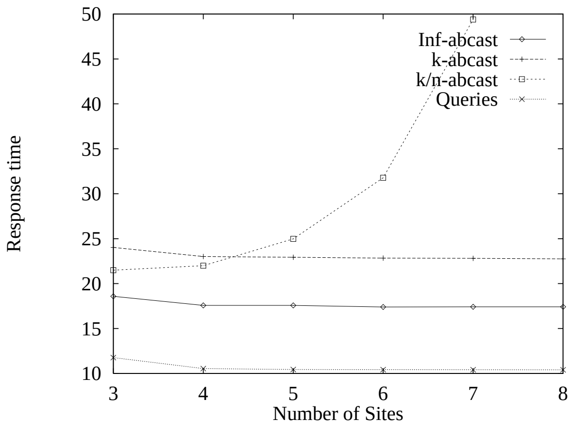

Figure 12: Response time (TPS = 1000)

Reordering. We consider next the effect of the reordering technique. Figures 13 and 14 show the abort rate for the k-*abcast* and the k/n-*abcast* based configurations, with different reorder factors. We do not consider the ∞-*abcast* based solution because reordering does not bring any improvement to the abort rate in this case (even if more transactions could pass the certification test, the terminating thread would not be able to process them). In both cases, reorder factors smaller then $4n$, have proved to reduce the abort rate without introducing any side effect. For reordering factors equal to or greater than $4n$, the data contention introduced by the reordering technique leads to an increase on the abort rate that is greater than the reduction on the abort rate that can be obtained with its use (i.e., the reordering technique increases the abort rate of update transactions).

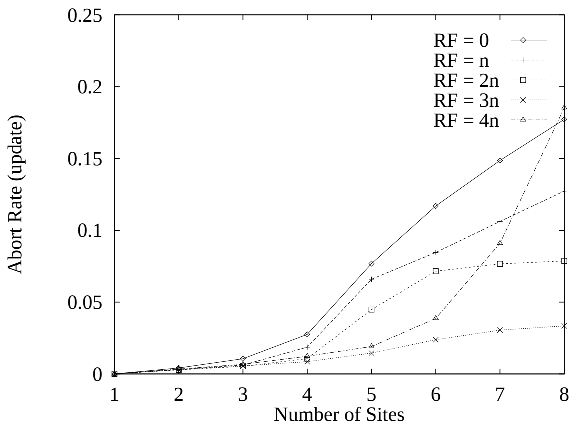

Figure 13: Reordering (k-*abcast*)

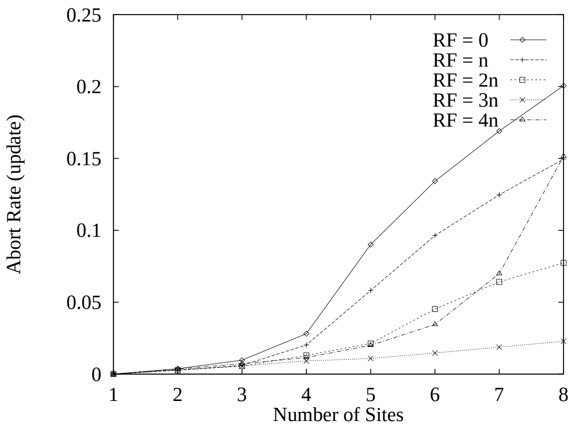

Figure 14: Reordering (k/n-*abcast*)

Overall Discussion. Besides showing the feasibility of the database state machine approach, the simulation model allows to draw some conclusions about its scalability. Update transactions scalability is determined by the scalability of the atomic broadcast primitive, which has showed to be a potential bottleneck of the system. This happens because the network is the only resource shared by all database sites (and network bandwidth does not increase as more database sites are added to the system). As for queries, only a slight grow in the abort rate was observed as the number of sites increase. This is due to the fact that queries are executed only locally, without any synchronisation among database sites.

The above result about update transactions scalability deserves a careful interpretation since in which concerns network utilisation, techniques that fully synchronise transactions among database sites (e.g., distributed 2PL protocol [6]) probably will not *do better* than the database state machine approach. The argument for this statement is very simple: they consume more network resources. A typical k/n-abcast algorithm needs about $4n$ [25] messages to deliver a transaction, where n is the number of database servers, and a protocol that synchronises transaction operations needs around $m \times n$ messages, where m is the number of transaction operations (assuming that reads and writes are synchronised). Thus, unless transactions are very small ($m \leq 4$), the database state machine approach outperforms any fully synchronisation technique in number of messages. Furthermore, the simulation model also shows that any effort to improve the scalability of update transactions should be concentrated on the atomic broadcast primitive.

Finally, if on the one hand the deferred update technique has no distributed deadlocks, on the other hand its lack of synchronisation may lead to high abort rates. The simulation model has showed that, if well tuned, the reordering certification test can overcome this drawback.

7 Related Work

The work here presented is at the intersection of two axes of research. First, relying on a certification test to commit transactions is an application of optimistic concurrency control. However, terminating transactions with an atomic broadcast primitive is an alternative to solutions based on atomic commit protocols.

Although most commercial database systems are based on (pessimistic) 2PL synchronisation [14], optimistic concurrency control have received increasing attention, since it introduction in [21] (see [32] for a brief survey). It has been shown in [3] that if sufficient hardware resources are available, optimistic concurrency control can offer better transaction throughput than 2PL. This result is explained by the fact that an increase in the multiprogramming level, in order to reach high transaction throughput, also increases locking contention, and thus, the probability of transaction waits due to conflicts, and transaction restarts to solve deadlocks. The study in [3] is for a centralised single-copy database. One could expect that in a replicated database, the cost of synchronising distributed accesses by message passing would be non negligible as well. In fact, a more recent study [12] has shown that fully synchronising accesses in replicated database contexts (as required by 2PL) is *dangerous*, since the probability of deadlocks is directly proportional to the third power of the number of database sites in the system.

The limitations of traditional atomic commitment protocols in replicated contexts have been recognised by several authors. 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 [29] 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 [18]. 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 [27] 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 [20]. In [24], 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. The immediate update replication consists in atomic broadcasting every write operation to all database sites. For the deferred update replication, two atomic broadcasts are necessary to commit a transaction. An alternative solution is also proposed, using 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).

Amir et al. [4] also utilise atomic broadcast to implement replicated databases. However, the scheme proposed considers that clients submit individual object operations rather than transactions.

8 Concluding Remarks

This paper shows how the state machine approach can be used to implement a replication protocol in a cluster of database servers. Replication in this scenario is used to improve both fault tolerance (e.g., by increasing data availability), and performance (e.g., by sharing the workload among servers). The database state machine approach implements a form of deferred update technique. The agreement and order properties of the state machine approach are provided by an atomic broadcast primitive. This approach has the benefit that it encapsulates all communication between database sites in the atomic broadcast primitive. The paper also shows how transaction aborts, due to synchronisation reasons, can be reduced by the reordering certification test.

The state machine approach is evaluated by means of a detailed simulation model. The results obtained show the role played by the atomic broadcast primitive, and its importance for scalability. In particular, the simulation model also evaluates the reordering certification test and shows that in certain cases, specifically for 8 database servers, it reduces the number of transactions aborted from 20% to less than 5%.

Finally, in order to simplify the overall approach, we did not address some issues that may deserve further analysis. For example, one such point concerns recoverability conditions for atomic broadcast primitives. Another issue concerns how clients choose the servers that will execute their requests.

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] R. Agrawal, M. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

[4] 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.

[5] Y. Amir, L. E. Moser, P. M. Melliar-Smith, P. A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In Robert Werner, editor, *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 551–560, Pittsburgh, PA, May 1993. IEEE Computer Society Press.

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

[7] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

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

[9] J. M. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[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] H. Garcia-Molina and A. Spauster. Ordered and Reliable Multicast Communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[12] 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.

[13] J. N. Gray, R. Lorie, G. Putzolu, and I. Traiger. *Readings in Database Systems*, chapter 3, Granularity of Locks and Degrees of Consistency in a Shared Database. Morgan Kaufmann, 1994.

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

[18] H. V. Jagadish, I. S. Mumick, and M. Rabinovich. Scalable versioning in distributed databases with commuting updates. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, pages 520–531, April 1997.

[19] Raj Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling.* John Wiley and Sons, Inc., New York, 1991.

[20] 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.

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

[22] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[23] S. W. Luan and V. D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Trans. Parallel & Distributed Syst.*, 1(3):271–285, July 1990.

[24] 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.

[25] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. *Lecture Notes in Computer Science*, 1470:513–520, 1998.

[26] O. T. Satyanarayanan and D. Agrawal. Efficient execution of read-only transactions in replicated multiversion databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):859–871, October 1993.

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

[28] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[29] 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.

[30] D. Stacey. Replication: Db2, oracle, or sybase? *SIGMOD Record*, 24(5):95–101, dec 1995.

[31] 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.

[32] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):173–189, January 1998.

[33] U. Wilhelm and A. Schiper. A Hierarchy of Totally Ordered Multicasts. In *14th IEEE Symp. on Reliable Distributed Systems (SRDS-14)*, pages 106–115, Bad Neuenahr, Germany, September 1995.