Transaction Reordering in Replicated Databases

Fernando Pedone* Rachid Guerraoui André Schiper
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Abstract

This paper presents a fault-tolerant lazy replication protocol that ensures 1-copy serializability at a relatively low cost. Unlike eager replication approaches, our protocol enables local transaction execution and does not lead to any deadlock situation. Compared to previous lazy replication approaches, we significantly reduce the abort rate of transactions and we do not require any reconciliation procedure.

Our protocol first executes transactions locally, then broadcasts a transaction certification message to all replica managers, and finally employs a certification procedure to ensure 1-copy serializability. Certification messages are broadcast using a non-blocking atomic broadcast primitive, which alleviates the need for a more expensive non-blocking atomic commitment algorithm. The certification procedure uses a reordering technique to reduce the probability of transaction aborts.

1. Introduction

Although replication has long been considered an intuitive way to increase availability, designing an efficient distributed protocol that provides fault-tolerance and ensures replica consistency is still a difficult issue. In the context of databases, Gray et al. [8] distinguish two classes of replication protocols: *eager* and *lazy*[1]. Eager protocols are considered to be too expensive as they require remote communication during transaction execution (each data access is synchronized by all replica managers). Lazy protocols enable local transaction processing (i.e., do not require any remote communication during transaction execution), but before committing, the transactions have to be certified. The certification procedure checks (with other replica managers) whether the transactions violate data consistency, in which case they are either aborted, or committed and later *reconciliated* with other transactions. The major drawback of lazy protocols is the high rate of aborts and reconciliation procedures. In this paper we present a lazy replication protocol that ensures data consistency (i.e., *one-copy serializability* without reconciliations) with a low abort rate.

Intuitively, our replication protocol performs as follows. A transaction T_i is first executed locally at a given replica server. Then, using a non-blocking atomic broadcast primitive, a certification message for T_i is broadcast to all replica servers. The atomic broadcast primitive alleviates the need of a more expensive non-blocking atomic commitment protocol. Every server that delivers T_i's certification message executes a certification procedure, deciding for the commit or the abort of T_i. Like any lazy replication approach, our certification procedure checks if T_i can be serialized after committed transactions. If not, rather than aborting T_i, we use a *reordering* technique to find out whether T_i can be serialized somewhere *before* some committed transactions. If T_i cannot be reordered, it is ultimately aborted. We show that the abort rate of our protocol is significantly lower than the abort rates of previous lazy protocols. As we will point out in Section 6, a similar idea was used in distributed shared memory systems to reorder *old writes* and hence ensure sequential consistency at a low cost [2].

The remainder of the paper is organized as follows. Section 2 defines the replicated database model. Section 3 presents our replication protocol and sketches its correctness proof. Section 4 is dedicated to implementation issues and performance measurements. We compare the abort rate of our replication protocol to well-known replication protocols. Then we discuss the memory cost of reordering transactions and present a tradeoff between storage space and abort rate. Finally, we compare the cost of a non-blocking atomic broadcast primitive to traditional atomic commitment protocols. Section 5 discusses some related work and Section 6 concludes the paper. The correctness proof of the

---

*Supported by Colégio Técnico Industrial, University of Rio Grande, Brazil

[1]The distinction between *primary backup (master)* and *state machine* protocols leads to an orthogonal classification [10].

replication protocol is detailed in the Appendix.

2. Replicated Database Model

In the following we describe our distributed system model and recall the definition of atomic broadcast ($ABCAST$). As we will see in Section 3, the $ABCAST$ primitive allows a modular description of our replication protocol and simplifies the management of replicas. To describe our replication protocol we do not make any assumption on how the $ABCAST$ primitive is implemented. This issue will be discussed in Section 4.

2.1. Processes

We consider a distributed system composed of a set of processes communicating by message passing. A process may fail only by crashing, and a correct process is a process that does not crash. Processes do not behave maliciously (we do not consider Byzantine failures), and messages are not corrupted. Processes are connected through eventually reliable channels, which guarantee that every message sent by a correct process to another correct process is eventually received (this is implemented by retransmitting lost messages).

We distinguish two kinds of processes: *client processes*, that generate the transactions (e.g., by submitting operation requests), and *server processes*, that execute the transactions. The database is fully replicated at each server and there is no centralized control over the replicas (i.e., there is no master or primary server).

2.2. Atomic Broadcast

A server process executes transactions locally and, at commit time, broadcasts a *certification message* to all other servers. This is performed using an atomic broadcast primitive ($ABCAST$), which ensures that the certification messages are *delivered* in the same total order by all correct servers. Note that this guarantee applies to message *delivery* and not to message *reception*. Basically, a server first receives a message, performs some coordination with the other servers (to ensure the above guarantee), and then delivers the message.

Let $ABCAST(m)$ denote the event of sending a message m, using the $ABCAST$ primitive, to the set of server processes. The $ABCAST$ primitive satisfies the following three properties:

Order. Consider $ABCAST(m_1)$ and $ABCAST(m_2)$, and two server processes s_i and s_j. If s_i and s_j deliver messages m_1 and m_2, they deliver them in the same order (i.e., either m_1 before m_2, or m_2 before m_1).

Atomicity. Consider $ABCAST(m)$. If one server process s delivers m, then every correct server process delivers m.

Termination. If a correct server process s executes $ABCAST(m)$, then every correct server process eventually delivers m.

The termination condition is a liveness condition. It ensures progress of the system and expresses the *non-blocking* characteristic of the $ABCAST$ primitive.

3. The Lazy Transaction Reordering Protocol

In this section we first give an abstract overview of the protocol and then describe the two main tasks executed by every server: the execution task and the certification task.

3.1. Protocol Overview

A transaction T_i is first executed at a given replica server, e.g., the closest server to the client that initiated the transaction. The server handles the transaction operations through its *execution task*. At commit time, using the $ABCAST$ primitive, the server broadcasts a certification message for T_i to all other replica servers. Every server delivers T_i's certification message, within its *certification task*, and checks whether each data item read by T_i is still up to date. Like in any lazy replication approach, we first check if T_i can be serialized after committed transactions. If not, rather than aborting T_i, we use a *reordering* technique to find out whether T_i can be serialized somewhere before some committed transactions. If T_i cannot be reordered, it is ultimately aborted.

In a given server, a transaction can be in one of four states (Figure 1): *(1) active* (i.e., the transaction's read/write operations are being performed by the execution task of a server), *(2) certify* (i.e., the transactions operations have terminated and its certification message has been delivered in the server), *(3) committed* or *(4) aborted*. A transaction cannot change its state after reaching states (3) or (4). While in state (1), a transaction cannot reach state (3), unless it has first passed by state (2).

3.2. The Execution Task

The servers implement a multiversion concurrency control, with transactions using *snapshots* of the database [3]. Snapshots are an abstraction that reflect the (committed) data as of the time the transaction started. In practice, snapshots are implemented by keeping one current version of the database and bringing previous data versions from the log [4, 9].

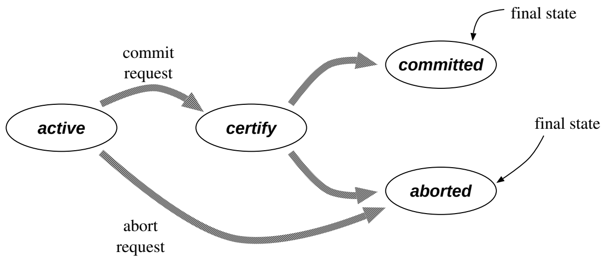

Figure 1. Transaction states

To execute a transaction T_i, a client process first contacts a server process, which associates with T_i a snapshot of the database. All T_i's operations are executed in the same server, and T_i's write operations are tentative (not seen by other transactions) until T_i commits. No transaction is blocked during its processing (we use optimistic concurrency control without locking), but aborts may occur if the system is not able to maintain a consistent snapshot of the database. When a transaction is ready to commit, the execution task broadcasts a *certification* message, containing transaction control information (e.g., the *deferred updates*, the *readset* and the *writeset*), to the other servers.

Figure 2 depicts the protocol model. *Server 1* first creates a snapshot of the database that will be used by the incoming transaction T_i, issued by *Client A*. All operation requests submitted by *Client A* are sent to *Server 1*. When *Client A* issues a commit request, *Server 1*'s execution task contacts all correct servers, sending them T_i's certification message (using the $ABCAST$ primitive). On delivering this message, every server (including *Server 1*) executes the *certification* task and tries to commit the transaction. If the commit is possible, a new version of the database is generated. In any case, T_i's snapshot is no longer necessary.

The procedure performed in the execution task is shown in Figure 3. Tentative transactions (in the active or certify state) are denoted $T_i, T_j, ...$ The structures *readset* ($RS(T_i)$) and *writeset* ($WS(T_i)$) represent the data accessed by a transaction, and the structures $pred[T_i]$ denotes the last committed transaction when T_i started. The variable $last$ denotes the last committed transaction in the server.

The execution procedure performs as follows. Lines 5-9 initialize the transaction control structures and creates the snapshot that the transaction will use. All *reads* will be performed on that snapshot. The only exception is for data items the transaction updates itself (line 16). When a transaction terminates its operations, its *deferred writes*, *readset* and *writeset* are broadcast to all correct replicas within a certification message (line 22). This message is delivered by the *certification* tasks (Section 3) of all (available) servers.

3.3. The Certification Task

The *certification* task delivers the *certification* message and checks whether the execution is consistent (serializable). If consistent, the transaction's deferred updates are applied and a new version of the database is generated. The certification and updating procedures are done atomically, in a deterministic way (as in the *state machine* approach [14]). The $ABCAST$ primitive ensures that all servers certify all transactions in the same order, and if one correct server certifies a transaction, then all other correct servers do likewise.

In this section, we describe our certification technique and show how it ensures *one-copy serializability*. As this technique can be viewed as an extension to the classical certification introduced by Kung and Robinson [11], we first recall their approach.

3.3.1. Kung-Robinson's Certification

Intuitively, Kung-Robinson's technique consists in constructing an abstract sequential order, called *serialization order*, according to which, committed transactions appear to have executed. This order corresponds to the transaction *certification order*. We use $T_{(l)}$ to denote a committed transaction, and $RS(T_{(l)})$ and $WS(T_{(l)})$ to denote the *readset* and *writeset* of $T_{(l)}$, respectively. Given that transactions $\{T_{(l)}, T_{(l+1)}, \ldots, T_{(n)}\}$ have committed after T_i had started, T_i can be committed if

$$(WS(T_{(l)}) \cup \ldots \cup WS(T_{(n)})) \cap RS(T_i) = \emptyset. \quad (1)$$

To illustrate the certification test, consider the following situation where transactions T_i and T_j executed concurrently, possibly at different servers, and are being certified (Figure 4). If T_j's certification message is delivered before T_i's (the $ABCAST$ primitive guarantees that this happens at all correct servers), T_i will be aborted because T_i read data items that were updated by T_j. However, if T_i's certification message is delivered before T_j's, both transactions will be accepted and committed because T_j did not read any item that T_i wrote. Since T_i and T_j are concurrent, they can assume any relative serial order, but depending on the order they assume, less aborts are produced. The implications of this observation will be developed in the next section.

3.3.2. Reordering Transactions

Kung-Robinson's certification technique aborts concurrent transactions whose certification messages were not lucky enough to be delivered in a favorable order. Our reordering technique is based on the observation that the transaction serialization order does not need to be the same as the transaction certification order. The idea is to dynamically build

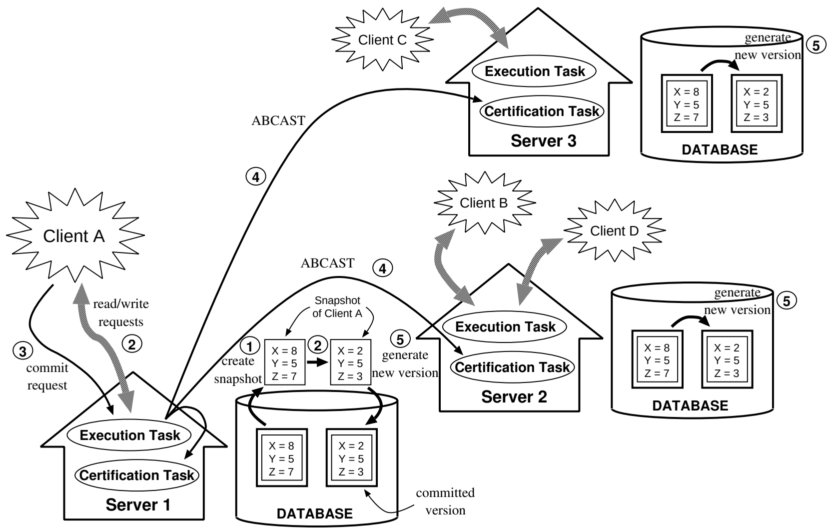

Figure 2. Protocol model

a serialization order (that does not necessarily follow the certification order), in such a way that less aborts are produced. By reordering a transaction to a position other than the current one (i.e., after the concurrent transactions that have already committed), our protocol increases the possibilities of committing.

Our certification algorithm maintains an abstract serialization order, according to which, transactions appear to have executed. A transaction T_i may be reordered to any position between two transactions that executed concurrently with it[2] but that have already committed. Consider $\{T_{(l)}, T_{(l+1)}, \ldots, T_{(n)}\}$ as these concurrent transactions, and assume that they appear in the serialization order as follows: $T_{(l)}; T_{(l+1)}; \ldots; T_{(n)}$. The condition for reordering T_i to the m^{th} position is stated below.

$$\begin{array}{c} (WS(T_{(l)}) \cup \ldots \cup WS(T_{(m-1)})) \cap RS(T_i) = \emptyset \\ and \\ WS(T_i) \cap (RS(T_{(m)}) \cup \ldots \cup RS(T_{(n)})) = \emptyset \end{array} \quad (2)$$

According to condition (2), although T_i reaches the certify state after transactions $\{T_{(l)}, T_{(l+1)}, \ldots, T_{(n)}\}$ have committed, T_i cannot read any data value that was updated by transactions ordered before it, and T_i must not update items that were read by transactions ordered after it. The first expression is similar to condition (1) and the second depicts the fact that, if the temporal order included T_i in the m^{th} position, then all transactions coming after T_i would have checked whether their $readset$ had any common element with T_i's $writeset$.

[2] Two transactions T_i and T_j are concurrent if T_i started at server S_k before T_j has committed at S_k, and T_j started at S_k before T_i has committed at S_k.

3.3.3. The Certification Algorithm

Figure 5 presents the algorithm executed when a certification message is delivered at server processes. The algorithm uses a data structure seq which denotes the transaction serialization order. If a transaction $T_{(l)}$ precedes another transaction $T_{(m)}$ in seq, then, independently of the order along which they were delivered and committed, everything happens as if $T_{(l)}$ had executed before $T_{(m)}$, e.g., $T_{(l)}$'s updates are performed after $T_{(m)}$'s. The algorithm also uses a function $Pos(T_{(l)})$ that returns the location of $T_{(l)}$ in seq.

The certification procedure (Figure 5) is the direct implementation of equation (2), with some optimizations. It starts looking for a range of positions in seq where the committing transaction could be reordered. This is done by calling functions $MinPos()$ (lines 13-15) and $MaxPos()$ (lines 16-19). If no such range is found, no command is executed and the procedure aborts the transaction (line 12). $MaxPos()$ evaluates the first part of equation (2). The test it performs (line 17) is slightly different from the first expression in equation (2) because it optimizes for the cases where transactions concurrent to T_i were reordered to a position before the transactions from which T_i read, and so, have no effect over T_i's reads. $MinPos()$ is an optimization that tries to find a position for T_i that is before $T_{(pred[T_i])}$. This is possible if T_i did not read any item updated by $T_{(pred[T_i])}$.

The second part of equation (2) is evaluated in the following. The acceptance test (line 3) checks whether T_i's writes might invalidate reads of already committed trans-

```
/* Task 1: Execution Procedure - Server Site */
repeat forever
    receive(operation, parameters);
    case operation of
    BeginTransaction:
        pred[T_i] ← last;
        create a snapshot for T_i;
        RS(T_i) ← ∅;
        WS(T_i) ← ∅;
        return(OK);
    ReadData:
  /* value ← ReadData(T_i, data item) */
        update RS(T_i);
        process read using the snapshot;
        return(value read);
    WriteData:
  /* status ← WriteData(T_i, data item, value) */
        update WS(T_i);
        process write operation tentatively;
        return(status);
    Abort: status ← Abort(T_i)
        release all resources allocated to T_i;
        return(ABORTED);
    CommitTransaction:
        status ← ABCAST(RS(T_i), WS(T_i),
deferred updates, pred[T_i]);
        release all resources allocated to T_i;
        return(status);
```

Figure 3. Execution procedure for transaction T_i

actions. Transaction T_i may be reordered to position m if T_i's writes do not affect any read executed by transactions $\{T_{(m)}, \ldots, T_{(last)}\}$. In the same way, T_i's writes must not overwrite transactions in greater positions (line 4). If a valid position is found, some internal data structures are updated (lines 5-9) and T_i's effective writes are made available to new incoming transactions (line 10).

3.3.4. Correctness: Proof Sketch

As assured by the termination property of the $ABCAST$ primitive (see Section 2.2), after requesting the commit, every transaction eventually reaches the certify state and, depending on the conflicts generated, is committed or aborted. Hence, every transaction issued by a correct client to a correct server is eventually terminated at all correct servers. In Section 4.1 we calculate the abort rate for our algorithm, which may be seen as a liveness parameter.

Proving the safety of our replication protocol, comes down to showing that every history it generates is one copy

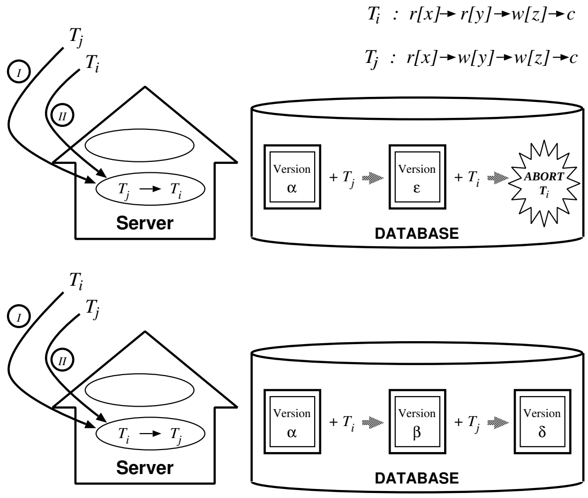

Figure 4. Concurrent transactions

serializable. We use the *Multiversion Graph* theorem of Bernstein et al. [4] which states that a multiversion history is one-copy serializable if it produces an acyclic multiversion serialization graph.

Our proof is composed of two parts. In the first part, we show that all correct servers create the same multiversion serialization graph. We prove this by induction on the size of the graphs, using the properties of the $ABCAST$ primitive and the determinism of the certification task. That is, based on the same local state (the same initial graph), the same input (a transaction certification message), and the same (deterministic) certification procedure, all servers produce the same final graph. In the second part, we prove that every such graph is acyclic by showing that, for every edge $T_i \rightarrow T_j$ in a multiversion serialization graph, we have $Pos(T_i) < Pos(T_j)$ (remember that $Pos()$ defines a transaction sequential relation seq - Section 3.3.3). Using the *Multiversion Graph* theorem, we conclude that our replication protocol guarantees one-copy serializability. The full version of the proof appears in [13].

4. Evaluation of the Protocol

In this section, we evaluate the performance of our protocol and discuss some implementation issues. We compare the abort rate of our protocol with the abort rates of previous replication protocols (eager and lazy) using a probabilistic analysis based on Gray's model [9, 8]. Then we discuss the memory cost of reordering transactions and point out a tradeoff between memory storage and abort rate. Finally, we consider the cost of broadcasting transaction certification messages and compare our solution (based on a non-blocking atomic broadcast) with traditional atomic commit-

/ Task 2: Certification Procedure - Server Site */*

upon delivering $[RS(T_i), WS(T_i), deferred\ updates, pred[T_i]]$ message;
for $l = MinPos()$ **to** $MaxPos()$ **do** $\{MinPos()$ *and* $MaxPos()$ *are indexes of 'seq'*$\}$
  if $WS(T_i) \cap \bigcup_{m=l}^{last} RS(T_{(seq[m])}) = \emptyset$ **then** $\{$*do* T_i*'s writes change past transactions' view?*$\}$
    $EWS(T_i) \leftarrow WS(T_i) - \bigcup_{m=l}^{last} WS(T_{(seq[m])})$; $\{$*... no: determine Effective Write Set of* $T_i\}$
    $last \leftarrow last + 1$; $\{$*one more transaction*$\}$
    for $m = last$ **downto** $l + 1$ **do** $seq[m] \leftarrow seq[m-1]$; $\{$*insert* T_i *in the* m^{th} *position*$\}$
    $seq[l] \leftarrow last$; $\{$*ditto*$\}$
    $RS(T_{(last)}) \leftarrow RS(T_i)$; $\{$*update read set ...*$\}$
    $WS(T_{(last)}) \leftarrow WS(T_i)$; $\{$*... and write set of* $T_{(last)}\}$
    process atomically all updates in $EWS(T_i)$; $\{$*writes become public*$\}$
    return(COMMITTED); $\{$*done*$\}$
return($Abort(T_i)$); $\{$*there is no position to insert* $T_i\}$

function $MinPos()\ :\ integer$;
/ Determines the minimal position for* T_i *in seq. We assume a transaction* $T_{(0)}$ *that initializes */*
/ the database so that* $\forall T_i,\ RS(T_i) \cap WS(T_{(0)}) \neq \emptyset$. **/*
for $l = pred[T_i]$ **downto** 0 **do** $\{$*for all transactions that had committed when* T_i *started*$\}$
  if $RS(T_i) \cap WS(T_{(l)}) \neq \emptyset$ **then** $\{$*did* T_i *read something that was written by* $T_{(l)}$*?*$\}$
    return($Pos(T_{(l)}) + 1$); $\{$*... yes:* T_i *cannot be inserted before* $T_{(l)}\}$

function $MaxPos()\ :\ integer$;
/ Determines the maximal position for* T_i *in seq. */*
for $l = pred[T_i] + 1$ **to** $last$ **do** $\{$*for all transactions that committed after* T_i *started*$\}$
  if $RS(T_i) \cap (WS(T_{(l)}) - \bigcup_{m=Pos(T_{(l)})+1}^{Pos(T_{(pred[T_i])})} WS(T_{(seq[m])})) \neq \emptyset$ **then** $\{$*has* $T_{(l)}$ *an effect over* T_i*?*$\}$
    return($Pos(T_{(l)})$); $\{$*... yes:* T_i *cannot be inserted after* $T_{(l)}\}$
return($last + 1$); $\{$*all items* T_i *read are up to date*$\}$

Figure 5. Certification procedure for transaction T_i

ment algorithms.

4.1. Abort Rate

Figures 6 and 7 represent the abort rate for several database replication approaches. These rates were obtained by means of a probabilistic analysis of the protocols (for further details see [13]). For all cases $DB\_Size = 10000$ data items, $Nodes = 4$, $Reads = 10$, $Writes = 10$ and $Action\_Time = 0.01$ seconds.

In the case of an eager approach, a transaction is aborted if it is involved in a deadlock with other transactions. For the Kung-Robinson's lazy replication approach, a transaction aborts each time there is a conflict with another transaction, and with a multiversion database model, the Kung-Robinson's certification only considers write-read conflicts. As shown in Figure 6, reordering transactions significantly reduces the abort rate of lazy protocols, which becomes closer to that of an eager protocol (Figure 7).

4.2. Memory Cost

Lazy protocols do not incur any cost with locking, deadlocks or unnecessary waits, but introduce the drawback of maintaining transaction's *readsets* and *writesets*. In order to commit a transaction, our protocol (like any lazy protocol) has to certify the transaction with all concurrent transactions. Hence, the *readsets* and the *writesets* of these transactions need to be maintained. As transactions might start at different servers without any synchronization, it is *a pri-*

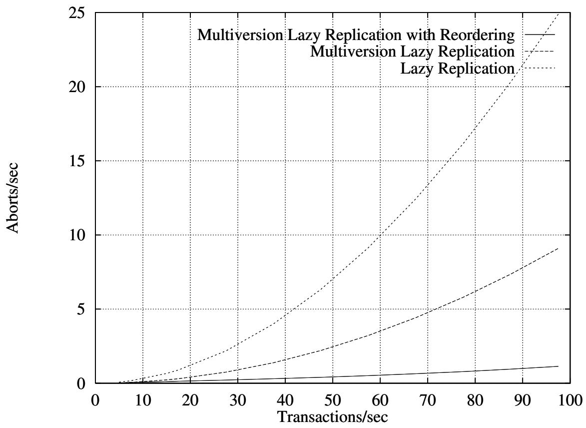

Figure 6. Lazy replication

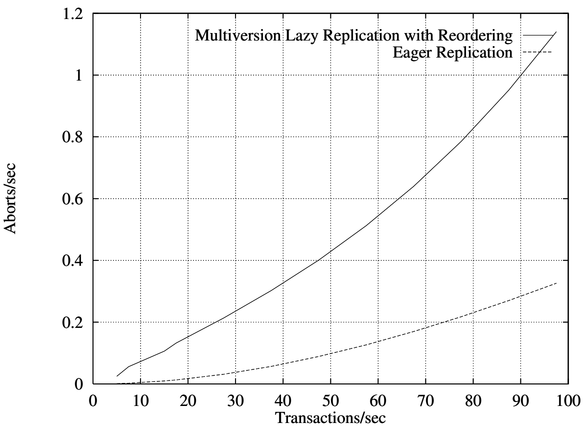

Figure 7. Eager *vs.* lazy replication

ori impossible to know until when these sets are necessary. One way to deal with this problem is to have the servers synchronize themselves periodically for garbage collecting unnecessary information.

Representing the *readsets* and *writesets* can be viewed as a tradeoff between storage space and abort rate. A simple solution consists in representing directly the data items accessed by the transactions, but this might consume a lot of space (mainly for the *readsets*). Another option consists in representing this information in a higher level, e.g., the SQL query statement for a relational database or the class for an object oriented database. This last option requires less storage space but would lead to a higher abort rate (more conflicts).

4.3. The Cost of Broadcasting Transactions

In this section, we discuss the cost of implementing the $ABCAST$ primitive. We consider the $ABCAST$ algorithm presented by Chandra and Toueg [5], which is non-blocking (i.e., fault-tolerant) and tolerates an infinite number of false failures detections[3]. We present the cost of broadcasting a transaction using the $ABCAST$ algorithm and compare this cost with that of atomic commitment protocols. This comparison is actually unfair: on one side, the $ABCAST$ primitive ensures total order, which an atomic commitment protocol does not; on the other side, atomic commitment enables server processes to vote for the outcome of the transaction [4], which $ABCAST$ does not[4]. The comparison is however important as most replication techniques assume atomic commitment protocols. Even though the protocol that is generally used is the well-know $2PC$ (Two Phase Commit [7]), we also consider the cost of the non-blocking, fault tolerant, $3PC$ (Three Phase Commit [15]).

The table in Figure 8 shows, for each algorithm (2PC, 3PC, and $ABCAST$), the number of communication steps and the number of messages required to deliver a transaction certification message. We consider here a distributed system composed of n server processes. The results show that $ABCAST$ provides better performances than $3PC$. Furthermore, with broadcast network communications, the price for the $ABCAST$ is not very high, when compared to the cost of a $2PC$ (which is blocking).

Protocol	Resilience	Number of communication steps	No. of messages (point-to-point)	No. of messages (broadcast)
2PC	blocking	3	$3n$	$n+2$
3PC	non-blocking	5	$5n$	$2n+3$
ABCAST	non-blocking	4	$4n$	$n+3$

Figure 8. Cost of delivering a transaction certification message

5. Related Work

In the following, we compare our replication protocol to protocols that adopt a similar approach. We focus on three recent proposals, which can be viewed as optimizations of classical lazy protocols [11].

1. Agrawal et al. [1] present a lazy replication protocol, where transactions that execute at the same server share the same data (no multiversion abstraction), and locks are used to detect local conflicts. At commit time of a transaction, a certification message is sent to all available servers within an atomic broadcast primitive. A transaction is committed if it can be serialized after the transactions that have already committed. The certification test is similar to the classical Kung-Robinson's approach and this leads to an abort rate that is significantly higher than that of our protocol (see Section 4).

2. Gray et al. [8] propose a lazy replication protocol for mobile computing. Two kinds of servers are considered: *mobile servers* and *base servers*. Mobile servers store old versions of the database and issue tentative transactions that later, when connected to base servers, may be committed or aborted (depending on the conflicts generated). Base servers work in a master, lazy-oriented fashion. Most data items have their masters in base servers, and these servers are assumed to be always connected. There are two major differences from our approach. First, [8] assumes the existence of one master per data item, whereas in our case, there is no

[3]We do not recall here Chandra-Toueg's implementation, as this is out of the scope of the paper.

[4]Voting is useless in our case as server processes are deterministic.

master replica. A master constitutes a single point of failure, and during the periods when masters are unavailable, or partitioned, transactions cannot be performed. This difference reflects the fact that we focus on fault-tolerance, whereas [8] focuses on disconnected operations. Second, [8] assumes a user defined rule to check whether a transaction can be committed or not. In our case, the rule consists in checking whether a transaction can be reordered or not. It is important to notice that if we also distinguished the servers (mobile vs. base), and applied the changes to a subset of all replicas, our reordering technique might be used in a mobile computing context.

3. Oracle Version 7.1 [6] provides a lazy group and a lazy master replication protocols. In both cases, Oracle permits snapshots to be changed locally and later forwarded to the other replicas. In order to reduce the transaction abort rate, stale reads might be accepted but one-copy serializability is violated.

6. Concluding Remarks

This paper presents a fault-tolerant lazy replication protocol that localizes transaction execution at one server and, when committing the transaction, sends the transaction control information to other replica servers for certification. The transaction control information is sent to the servers using an atomic broadcast primitive. To reduce conflict resolution aborts, we propose a reordering technique that explores the serializability property of transactions.

Reordering transactions comes from the observation that traditional lazy concurrency control algorithms usually implement a property stronger than serializability. Indeed, although not necessary for the serializability property, the temporal order along which concurrent transactions are processed is also usually guaranteed. We have shown how we can by-pass this inconvenience, while still preserving serializability. In [2], a similar approach was proposed in the context of distributed shared memory. The authors point out the fact that ensuring *sequential consistency* (i.e., serializability) is less expensive than ensuring *linearizability* (i.e., serializability and temporal precedence). One improvement that seems natural to our reordering technique is to explore the possibility of completely changing the relative order of committed transactions, instead of trying to reorder the committing transaction to another position. However, this solution would lead to an NP-complete problem [12].

The use of an atomic broadcast primitive allows a modular description of our protocol, and simplifies some of the problems normally associated with replicated systems, i.e., atomicity and message ordering. Furthermore, as shown by the performance measures, non-blocking atomic broadcast has a lower cost than non-blocking atomic commitment protocols (i.e., 3PC). Therefore, our replication protocol can be viewed as a useful application of an atomic broadcast primitive in the context of replicated databases.

References

[1] D. Agrawal, G. Alonso, A. E. 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.

[2] H. Attiya and R. Friedman. A correctness condition for high-performance multiprocessors (extended abstract). In *Proceedings of the Twenty Fourth Annual ACM Symposium on Theory of Computing*, pages 679–690, Victoria, British Columbia, Canada, 4–6 May 1992.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2):1–10, June 1995.

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

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

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

[7] 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.

[8] 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*, June 1996.

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

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

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

[12] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[13] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. Technical report, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1997.

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

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