, , 1–23 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Independent Updates and Incremental Agreement in Replicated Databases

S. CERI ceri@ipmel2.elet.polimi.it
Dipartimento di Elettronica e Informatica
Politecnico di Milano (Italy)

M.A.W. HOUTSMA houtsma@trc.nl
Department of Computer Science
University of Twente (the Netherlands)

A.M. KELLER ark@cs.stanford.edu
Department of Computer Science
Stanford University (U.S.A.)

P. SAMARATI samarati@dsi.unimi.it
Dipartimento di Scienze dell'Informazione
Università di Milano - Milano (Italy)

Abstract. Update propagation and transaction atomicity are major obstacles to the development of replicated databases. Many practical applications, such as automated teller machine networks, flight reservation, and part inventory control, do not require these properties. In this paper we present an approach for incrementally updating a distributed, replicated database without requiring multi-site atomic commit protocols. We prove that the mechanism is correct, as it asymptotically performs all the updates on all the copies. Our approach has two important characteristics: it is progressive, and non-blocking. *Progressive* means that the transaction's coordinator always commits, possibly together with a group of other sites. The update is later propagated asynchronously to the remaining sites. *Non-blocking* means that each site can take unilateral decisions at each step of the algorithm. Sites which cannot commit updates are brought to the same final state by means of a *reconciliation* mechanism. This mechanism uses the history logs, which are stored locally at each site, to bring sites to agreement. It requires a small auxiliary data structure, called reception vector, to keep track of the time unto which the other sites are guaranteed to be up-to-date. Several optimizations to the basic mechanism are also discussed.

Keywords: Replicated databases, update propagation, transaction reconciliation, failures.

1. Introduction

Replicated databases are becoming of increasing interest for real-life applications. They may provide increased performance, availability, and site autonomy. However, these advantages mainly apply to read-only applications, and are jeopardized by the need for propagating updates to all sites.

The first problem with propagation of updates is to guarantee that a sufficient number of copies be updated so that their consistency is constantly preserved; strategies developed for this purpose include voting or token passing [1], [18], [26];

they are generally very complex and difficult to implement. In [8] we have given a classification and overview of many of these update strategies.

The second problem with propagation of updates is transaction atomicity; two-phase commit is the most widely used commit protocol in commercial database systems [11]. Commit protocols have intrinsic disadvantages, such as cost, delay, and reduced availability. Sites may be blocked once they have transferred their right to decide on abort or commit to the commit coordinator. Network partitions or site failures may thus lead to smaller availability of the database system.

Not all applications should pay the price for maintaining a consistent view on replicated data and immediate update propagation [16], [20], [30]. Examples of such applications are automated teller machine networks, flight reservation, and part inventory control. For instance, it is clearly unacceptable for a flight reservation system to become globally unavailable in case of a site failure or network partition. It is preferable to take the risk to overbook a plane (and possibly correct this situation at the end of the failure), than to stop making reservations. Therefore, current research focuses on delayed propagation of updates [32], [33], and transformation of global constraints into local constraints (thereby increasing local autonomy) [4], [5]. Some commercial systems are also developing products for dealing with asynchronous updates to replicated databases (e.g., Sybase, [12]).

In this paper we allow updates to be initiated at any site, and propagate them to other sites immediately if possible, or otherwise later. However, if a site cannot accept an update, the other sites still continue with the transaction. Also, we do not use an atomic commitment protocol across sites, thereby avoiding sites to get blocked; atomicity of transactions at each site is enforced during transaction execution. Transactions consist of actions being performed on data objects; we initially assume commutative actions, later we drop this assumption. We propose a *reconciliation mechanism* that will incrementally bring sites to agreement. Eventually, all sites will reflect all transactions, some of which may have been executed independently (e.g., during a network partition). The disadvantages of our approach are loss of strict serializability of transactions, and less strict constraint enforcement, but such disadvantages are acceptable for many applications. Some global constraints can be split into local constraints (e.g., using the demarcation protocol [4]) and therefore still be enforced; compensating actions can be applied to restore consistency after violation of global constraints [17], [35].

The main contribution of the paper is to combine gossip techniques for update propagation and techniques for transaction reconciliation in a unique framework. Algorithms for indipendent update executions and for reconciliations in such a framework are illustrated together with the proofs of their correctness.

1.1. Previous related work

Several update propagation strategies to replicated databases guarantee full consistency [1], [6], [14], [18], [26]. In case of network partitions, either updates are accepted on a subset of the sites, or transactions that are known not to lead to inconsistency are the only ones allowed to run [3].

These protocols suffer from unrealistically strong assumptions [34]; therefore, a lot of work has been done to allow some more flexibility [4], [9], [25]. A noticeable example of this is the work on epsilon-serializability [32], [33]. Updates are allowed to propagate through the system asynchronously; eventual consistency of the system is an asymptotical property. A whole family of methods, each one different in the level of asynchrony, can be described using the concept of epsilon-serializability. Another example of research that does not enforce immediate propagation of updates is the work on quasi-copies [2]; here, each replica is allowed a certain bounded deviation from the actual data value. Along the same lines, the work on the demarcation protocol [5] allows some degree of independence among the sites, allowing some updates to execute locally and be propagated asynchronously to other sites.

Another way of providing somewhat more flexibility is to avoid regular two-phase commit. For instance, in [23] a distributed transaction is split into several single-site atomic transactions that are put into a logical tree structure. Each of these subtransactions commits locally, with the provision that if the root of the tree commits, all its children commit too. In this approach, persistent transmission of messages is required, for instance, by mean of stable queues [7], or gossip mechanisms [22], [37]. Another approach that avoids two-phase commit is given in [29]; transactions commit at a primary site and are propagated asynchronously. A partial order among transactions is defined for controlling update propagation; the use of a partial order is also proposed in [27] and in [15] (in the context of concurrency control for groupware systems).

Finally, some systems allow even more flexibility. In [13] a system is described for maintaining weak consistency among various database systems. Updates are accepted at any site, and are guaranteed to be reflected at all other sites eventually. In [36] a system is described that uses timestamp-based concurrency control and proposes to apply updates to replicated data in their arrival order, possibly restoring inconsistencies when arrivals violate the timestamp ordering of transactions. This mechanism achieves consistency by undoing and re-executing updates which are out-of-order, and saves some of these operations at the cost of restoring additional information, such as read/write sets for update transactions. In [19] a timestamp message delivery protocol that implements eventual delivery is proposed. The approach uses periodic exchanges of messages between pairs of principals to propagate messages to groups of sites. Incoming messages are stored in a log and later delivered to the application in a defined order. The protocol maintains summary information on the messages it has received to decrease communication and to purge messages from the log. The use of the history log for propagation of updates was suggested in [24], and a preliminary description of how to apply history logs for propagating independent updates was given by us in [10].

Update propagation was considered also in the context of maintaining replicated dictionaries, with an approach which has several similarities with the one proposed in this paper. In [16], a vector is introduced in order to keep track, at each site, of the events that originated in the system. However, only the insert and delete operations are allowed, and the vector does not give any indication of how up to date other sites of the system are. Therefore, this approach requires a site to send its entire copy of a dictionary at each message. In [37], the approach of [16] was extended by storing at each site a matrix, instead of a vector. The matrix at each site indicates how up to date all sites are, thus limiting the communication requirements among sites. As we will see, [37] has therefore many features in common with our paper; however there are also major differences. [37] is limited to solving the dictionary problem, and both [16], [37] do not consider a reconciliation phase, which is one of the main component of our approach. Instead of using a specific reconciliation mechanism, a site can synchrously send part of its log and matrix to other sites. The use of vectors for update propagation is also suggested in [27]; this is based on gossip messages, where update propagation is enforced by two kinds of messages: update messages through which events are propagated, and ack messages by which a site acknowledges the reception of updates.

1.2. Outline

The paper is organized as follows. In Sec. 2 we describe our model of the database system and the transactions, and we state our assumptions. In Sec. 3 we describe the regular execution of transactions. In Sec. 4 we describe the reconciliation mechanism that achieves incremental agreement among sites. In Sec. 5 we illustrate an example of transaction execution and reconciliation. In Sec. 6 we describe the reconciliation mechanism for non-commutative actions. In Sec. 7 we describe a technique for migrating part of the log to archive. In Sec. 8 we describe the possible application scenarios of the reconciliation mechanism. Finally, Sec. 9 presents our conclusions.

2. Assumptions and preliminary definitions

In this section, we illustrate our assumptions and explain our notation.

2.1. Assumptions

We assume a fully replicated database, characterized by the following properties.

Communication Communication is order-preserving between any pair of sender and receiver sites, and message content is assumed to be correctly received. Furthermore, we assume a time-out mechanism that allows a process to detect that its messages were not acknowledged within a given time interval.

Time Each site has a logical local clock. Global ordering of actions executed at different sites is possible using a Lamport-style timestamping mechanism [28].

Objects For the purpose of our reconciliation mechanism, we assume a universal space of object identifiers. Objects are distinct (i.e., non-overlapping). We do not, however, make any assumptions on the granularity of objects; for example, they could be pages in a disk-based system.

Actions inside transactions We assume that actions on the database are unary (i.e., they affect a single data object), take constant arguments, and are commutative. Note, however, that we do not allow conditional branching on the value of database items. An example of such an action is to increment a specific bank account with a given amount of money. In Sec. 6 we will drop the assumption that actions be commutative.

Logging Each site logs all actions on a local history log on stable storage using the Write Ahead Log protocol [21]. Conventional transaction mechanisms ensure that all actions that are logged and then committed, are subsequently correctly reflected in the database. We assume that the local history log kept at each site is a sequence of unique records. Each record has the following structure:

$$\langle \textit{trans\_id, timestamp, coordinator\_site, object\_id, action} \rangle$$

Its meaning is as follows. For each action that is executed in the course of a transaction, we record in the log the transaction identifier, the timestamp for when the action is recorded in the log of the coordinator, the site number of the transaction's coordinator, the object identifier, and the action executed. An action is usually described through the action's name, the identifiers of the items used by the action, and the input parameters provided by the user's transaction, e.g., $\langle$*sum, tuple-id.field-id, 10*$\rangle$. We assume that actions are executed on data items that are contained within a single object, therefore, items have smaller granularity than objects.

Locking When a transaction or the reconciliation process reads or writes an object, it follows the 2-phase locking protocol locally [21].

2.2. Notation and invariant condition

In the following a_q^o denotes an action a executed on object o with site q as coordinator of the transaction, $time(a_q^o)$ denotes the timestamp of action a_q^o. H_q denotes the history log at site q, containing records with the format illustrated in the previous section.

For each site q and each object o, we introduce a small auxiliary data structure called *reception vector*, denoted by RV_q^o. This vector has an entry for each site k, $RV_q^o[1, \ldots, n]$. (The idea of using a vector for detecting inconsistency among sites, was proposed before in [16], [27], [31].) The semantics of the reception vector is illustrated by the following invariant condition:

Invariant 1 *For all sites x and y, objects o, and actions a_y^o: $a_y^o \in H_x \Leftrightarrow time(a_y^o) \leq RV_x^o[y]$.*

The meaning of the invariant is that if site x has executed some action a_y^o on object o originated at site y, it has also executed all actions on o that were previously originated at y. Thus, $RV_x^o[y]$ is the timestamp of the latest action on object o with site y as coordinator that was executed at site x. (The invariant ensures that site x includes any action originated at y up to the time indicated by the entry $RV_x^o[y]$.) Initially all reception vector entries are assigned a value t_0.

3. Transaction execution

We now describe execution of transactions that update the database. First we describe the algorithm, then the behavior during failures, and finally show the correctness of the algorithm.

3.1. Regular execution

We now describe transaction execution in absence of failures. In the description we refer to a *coordinator*, which is the site where the transaction is originated, and to *participants* the other sites where the transaction is executed. Note however that the description is easily generalizable to a client/server environment. In a client/server architecture, the client process calls several servers at various sites and then calls a system process for commit coordination; each server does its own logging. Our approach can be extended to such an architecture by assuming that the commit coordinator commits all available servers and that servers perform logging of reception vectors; the coordinator site is the site of the commit coordinator.

The proposed algorithm has two significant properties:

1. It is *progressive*: the coordinator can always commit, possibly with other sites.
2. It is *non-blocking*: each site can take unilateral decisions at each step of the algorithm, and as a result is never blocked.

The coordinator site c executes all its reads locally. All writes are also executed locally, and they are transmitted to all other sites together with the reception vector entry for c, $RV_c^o[c]$. On each write, a participant p compares this entry with its own entry for the coordinator site, $RV_p^o[c]$. If the entries are equal, i.e., site p has executed all actions on o originated at c, p will execute the write. If the comparison fails for a write operation, p will abort the transaction locally and not accept any further write for this particular transaction. If p accepts a write on object o, it also updates its reception vector entry for site c, $RV_p^o[c]$, with the time of the write action.

At the end of a transaction, the coordinator, with a synchronous write, forces all log records together with the commit record; this causes the atomic, independent commit at the coordinator site. Note that, at this point, the coordinator can inform the calling client or user's process that the transaction is successfully executed. The coordinator then sends its decision to all other sites. Each site that has fully participated in the transaction (i.e., has accepted all writes), commits upon receipt of the coordinator's decision, by forcing all log records in the log together with the commit record. It then sends a message to the coordinator indicating the commitment has been successfully executed. Given that the decision to commit each subtransaction is made locally, the algorithm is non-blocking.

For each site that either did not participate, or executed a local abort, the coordinator does not receive an acknowledgement of its commit decision. This means that those sites do not agree with the coordinator on the objects written by the transaction, and agreement has to be reached later by means of the reconciliation mechanism. The need for reconciliation is stored at each site in the table *Rec_Info*, which contains binary tuples of the form (*object, site*). When the coordinator of a transaction notices that a participant p has not acknowledged its commit decision, for each object o written by the transaction, it inserts the tuple (o, p) in its local table $\mathit{Rec\_Info}_c$.

The algorithm just described is presented in Fig. 1. The interaction between the coordinator and an arbitrary participant is graphically depicted in Fig. 2, which shows the records written on the log at each site, and the messages exchanged between the two sites. With reference to Fig. 2, slanted arcs represent message exchanges between the sites and vertical lines represent log recordings at a site.

Note that it is not required by this algorithm that the copies of the objects written by a transaction at the various sites have the same final value. For example, sites might have executed different sets of transactions in the past originating at other coordinating sites, due to site failures or network partition that have not yet been reconciled.

3.2. Dealing with failures

In the description of the algorithm we assumed no failures. Let us now study what happens if failures occur. According to our objectives, transactions should commit, even if some of the participants abort, and most importantly, sites should not become blocked.

We consider three possible types of failure: site crash, message loss, and network partition.

1. **Site failures.**

   (A) **Coordinator fails before commit.** Upon recovery, the transaction's actions are undone at the coordinator site. Meanwhile, all participants will have timed out and locally aborted the transaction.

   (B) **Coordinator fails after commit.** Upon recovery, the transaction's actions are redone locally and reception vectors are updated properly. The

```
**Coordinator** $c$

Write *Begin_transaction* log record;
Send *Begin_transaction* to all sites;
Execute any read action $r(o)$ locally;
repeat enter a log record for write action $w(o)$;
       send $Write(RV_c^o[c], w(o), t_w)$ to all sites;
       set $RV_c^o[c] := t_w$;
until end of write actions;
Force a *Commit* record into the log;
Send a *Commit* message to all sites;
After timeout enter $(Rec(o,p))$ in the
       reconciliation log for all objects $o$ written
       and all sites $p$ that did not acknowledge

**Participant** $p$ (initiated by a *begin_transaction* message)

Write *Begin_transaction* log record
repeat receive *write* message
       if $RV_p^o[c] = RV_c^o[c]$
       then enter a log record for the action;
            set $RV_p^o[c] := t_w$
       else local abort; exit
until end of write actions
If *Commit* message is received before timeout
then Force a *Commit* record into the log;
     send *Ack* to $c$
else abort
```

Figure 1. Transaction execution algorithm

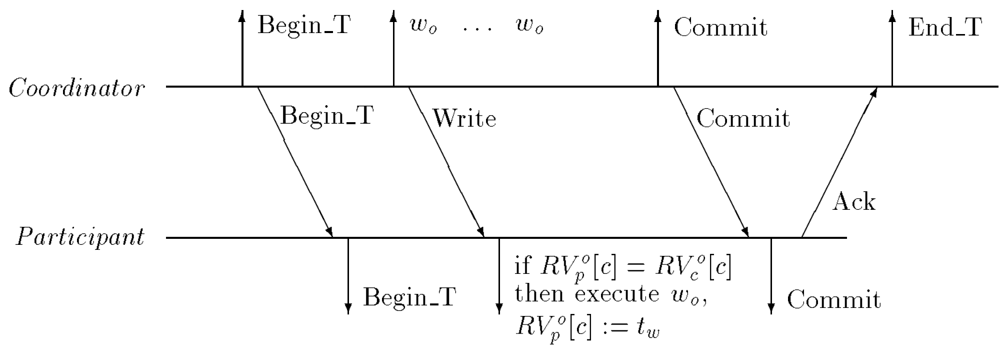

Figure 2. Normal transaction, without failures

coordinator assumes all the other sites have aborted and records the need for reconciliation in its *Rec_Info* table as described previously. Note that reconciliations may be requested for participants that have committed, although the coordinator was unaware of it. This conservative approach does not cause a problem (see Sec. 4.3).

(C) **Participant fails before commit.** Upon recovery, the transaction's actions are undone. Meanwhile, the coordinator has timed out on the *Ack* message and assumed that the participant has failed, recording the need for reconciliation in its *Rec_Info* table. The coordinator and participants that did receive the commit decision message have already decided on commit, and they will not change their decision.

(D) **Participant fails after commit.** Upon recovery, the transaction's actions are redone locally (unless the participant can safely assure that they were successfully executed on the database) and reception vectors are updated properly. Note that if the failure occurred before sending the *Ack* message, then the coordinator has assumed failure and recorded the need for reconciliation locally, but again the reconciliation will not cause problems (see Sec. 4.3). Note that neither in case 1(C) nor in case 1(D) does the participant become blocked. Also, the participant does not have to perform a remote recovery request, as the coordinator will record the need for reconciliation if it did not receive the commit acknowledgement from the participant.

2. **Message failures.** We assumed that messages were delivered in proper order; hence, if a message fails to reach a site, following messages will also fail to reach it and eventually time-outs will expire. Message failure is thus equivalent to some of the site failures just described.

3. **Network partition.** A network partition is equivalent to multiple site failures from the perspective of the coordinator. If a network partition occurs before the coordinator transmits its commit decision, the participant sites that become disconnected unilaterally abort as in case 1(A), and the coordinator commits as in case 1(C). If a network partition occurs after the coordinator has transmitted its decision, participants receiving the commit message will locally commit; upon timeout expiration the coordinator will record in its *Rec_Info* requests for reconciliation for participants that committed but did not successfully transmit an acknowledgement to the coordinator.

As the description shows, all sites with no local failures may commit a transaction.

3.3. Correctness

In this section we prove the correctness of transaction execution by proving the following theorem.

Theorem 1 *Transaction execution preserves Invariant 1.*

Proof By assumption the invariant held before the execution of a transaction. At all sites where the transaction is not committed, the data values, history log, and reception vectors remain unchanged. Moreover data values, history log, and reception vectors of all objects not written by the transaction remain unchanged.

Let us now consider a site p that committed a transaction (either during normal execution or during recovery after commit) and an object o written by the transaction. Since the object is locked during transaction execution no other action on o besides those in the transaction is executed at site p. Since the entries of the reception vector for sites different from the transaction coordinator do not change, we have to prove only the following: $a_c^o \in H_p \Leftrightarrow time(a_c^o) \leq RV_p^o[c]$, where c is the transaction coordinator.

($\Leftarrow$) Upon commit, $RV_p^o[c]$ is equal to the timestamp of the last write action on o in the transaction. Site p executed the transaction, therefore we know that before the execution of the first write action $RV_p^o[c] = RV_c^o[c]$. Since by assumption, the invariant held before the transaction execution, and $RV_c^o[c]$ is the time of the last action originated at site c, all actions originated at site c before the transaction execution are contained in H_p. Since p committed the transaction, all actions part of the transaction are in H_p. Moreover no other action could have been originated by c outside the transaction, which locked the object. Therefore, the invariant is satisfied.

($\Rightarrow$) Since $RV_p^o[c]$ is updated with the time of the latest action on o, after this action was inserted in the log, the implication trivially holds. $\square$

4. Reconciliation

We now describe the reconciliation mechanism. First we describe the algorithm, then the behavior in case of failures, and finally the correctness of the algorithm.

4.1. Algorithm

We now describe the *basic step* of the reconciliation mechanism, i.e., the reconciliation between two sites on a given object. Section 8 describes when to invoke the reconciliation mechanism, and the options that exist in its use. The basic step is as follows.

If a site s has a tuple (o, p) in its local table *Rec_Info*, it may start a reconciliation by sending a *reconciliation-request* to site p. Included with this message is the reception vector of s for the object o. If p is willing to take part in the reconciliation, it replies by sending its own reception vector for object o to s. From then on, the algorithm is completely symmetric (expressed by procedure *Reconcile* in Fig. 3).

Each site uses the reception vector it has received to scan its own history log and extracting those actions that have not been executed at the other site. Such actions are selected by comparing the timestamp of each of the actions with the entry for its coordinating site in the reception vector. From Invariant 1, the actions on object o which were executed at site x and not executed at site y, denoted with $\Delta^o_{x,y}$, are $\Delta^o_{x,y} = \{a^o_q \in H_x \mid time(a^o_q) > RV^o_y[q]\}$. These actions are thus extracted from the log at both sites and sent to the other site. When a site receives the actions from the other site, it executes them and appends them to its history log. At the end of the reconciliation, the reception vector at both sites is updated: each entry is assigned the maximum value of the corresponding entries of the two vectors.

After the update of the reception vector, a site commits the reconciliation locally by forcing the log records and a commit record into the log. Note that, once sites have transmitted the relevant actions to each other, there is no need for further communication between them. Indeed, either site s or site p could abort while the other one commits.

The description of the algorithm is given in Fig. 3 and is depicted graphically in Fig. 4.

4.2. Dealing with failures

In the description of the reconciliation algorithm, we assumed no failures. Let us now study what happens if failures occur. We have already noticed the algorithm is symmetric, except for the first part in which one of the sites behaves as the starter of the reconciliation process. This symmetry is also reflected in the behavior of sites upon failures.

If site x locally aborts a reconciliation on object o with site y, tuple (o, y) will not be deleted from *Rec_Info*$_x$. This ensures that the information regarding the need

```
**Starter** $s$

Send $\mathit{reconciliation}(s, RV_s^o, o)$ to site $p$;
Wait for $RV_p^o$ from site $p$;
Reconcile $(s, p, o, RV_p^o)$
Delete $rec(o, p)$ from $\mathit{Rec\_Info}_s$;
Commit

**Participant** $p$

Receive $\mathit{reconciliation}(s, RV_s^o, o)$ from site $s$;
**If** $rec(o, s)$ not in $\mathit{Rec\_Info}_p$
**then** insert it;
Send $RV_p^o$ to site $s$
Reconcile $(p, s, o, RV_s^o)$
Delete $rec(o, s)$ from $\mathit{Rec\_Info}_p$;
Commit


**Reconcile** $(x, y, o, RV_y^o)$

$\Delta_{x,y}^o := \{a_q^o \in H_x \mid time(a_q^o) > RV_y^o[q]\}$;
/* all actions executed at site $x$
        and not executed at site $y$ */
Send $\Delta_{x,y}^o$ to site $y$;
Wait for $\Delta_{y,x}^o$ from site $y$;
Execute all actions $a \in \Delta_{y,x}^o$;
        /* update object value */
$H_x := H_x \cup \Delta_{y,x}^o$;
        /* append actions to the history log */
$\forall q : RV_x^o[q] := \max(RV_x^o[q], RV_y^o[q])$;
        /* update reception vector */
```

Figure 3. Reconciliation algorithm

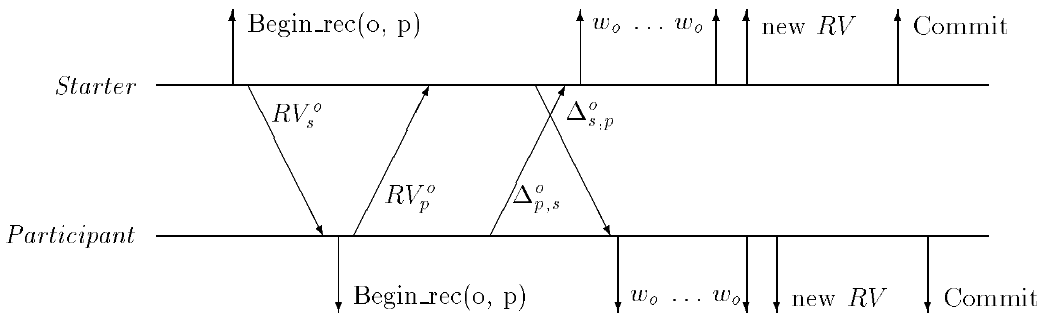

Figure 4. Reconciliation without failures

of a reconciliation will be deleted only if both sites commit the reconciliation. The insertion of this entry is always executed upon local abort and we will not indicate it in the following.

We now describe the behavior of the algorithm for the different types of failures.

1. **Site failures.**
   - (A) **Before commit.** Upon recovery, the site aborts the reconciliation locally, by undoing all write actions.
   - (B) **After commit.** Upon recovery, the site commits the reconciliation locally, by redoing all write actions.
2. **Lost messages.** The site that is waiting for the lost message will timeout and abort the reconciliation locally, by undoing all write actions.
3. **Network partition.**
   - (A) **Prior to message exchange.** This case is similar to case 2.
   - (B) **After message exchange.** Since the sites operate independently and do not need to interact after the reception of the history information, both of them will commit the reconciliation independently.

4.3. Correctness

We now prove that the reconciliation algorithm is correct. Correctness means that the value of an object o at commit of reconciliation reflects the timestamp-ordered execution of all the actions executed on o at the reconciling sites, and that Invariant 1 still holds.

Theorem 2 *If site x commits a reconciliation on object o with site y, the value of o at site x after reconciliation reflects the timestamp-ordered execution of all the distinct actions executed at x and y before reconciliation.*

Proof First, we prove that all actions executed at site y and not executed at site x are communicated to x from y upon reconciliation, i.e., $a_q^o \in H_y \wedge a_q^o \notin H_x \Rightarrow a_q^o \in \Delta_{y,x}^o$. Consider an action a_q^o such that $a_q^o \in H_y \wedge a_q^o \notin H_x$. Since Invariant 1 held at the start of reconciliation, we know that before reconciliation $RV_x^o[q] < time(a_q^o) \leq RV_y^o[q]$. Therefore, $a_q^o \in \Delta_{y,x}^o$ and the implication holds.

Moreover, we prove that no action has been executed more than once, i.e., actions already executed at site x are not communicated by y upon reconciliation: $a_q^o \in \Delta_{y,x}^o \Rightarrow a_q^o \notin H_x$ before reconciliation. Consider an action $a_q^o \in \Delta_{y,x}^o$. Since $a_q^o \in \Delta_{y,x}^o$, we know that before reconciliation $RV_x^o[q] < time(a_q^o) \leq RV_y^o[q]$. Since the invariant held at the start of reconciliation, we had $a_q^o \in H_y$ and $a_q^o \notin H_x$, and the implication holds.

Finally, we notice that since actions are commutative, any execution order will produce the same result. Hence, the theorem is satisfied. □

Theorem 3 *The reconciliation algorithm preserves Invariant 1.*

Proof By assumption the invariant held before the execution of the reconciliation. Consider site x that reconciles on object o with site y, we show that the following holds: $\forall q : a_q^o \in H_x \Leftrightarrow \mathrm{time}(a_q^o) \leq RV_x^o[q]$. If x did not commit the reconciliation, no changes were made to the object, its reception vector, and the history log, and hence the invariant holds. If x committed the reconciliation, we will now prove that the invariant still holds.
($\Leftarrow$) Consider an action a_q^o, such that $time(a_q^o) \leq RV_x^o[q]$. After reconciliation, we know that $RV_x^o[q] = \max(RV_x^o[q], RV_y^o[q])$ holds, and before reconciliation either $RV_x^o[q] \geq time(a_q^o)$ or $RV_y^o[q] \geq time(a_q^o)$ holds. Therefore, before reconciliation either $a_q^o \in H_x$ or $a_q^o \in H_y$. If $a_q^o \in H_x$ the implication obviously holds. If $a_q^o \in H_y$, by Theorem 2, $a_q^o \in H_x$ at the end of reconciliation.
($\Rightarrow$) Consider an action $a_q^o \in H_x$. Since $a_q^o \in H_x$ after reconciliation, either $a_q^o \in H_x$ before reconciliation, or $a_q^o \in \Delta_{y,x}^o$. If $a_q^o \in H_x$ before reconciliation, $time(a_q^o) \leq RV_x^o[q]$ before reconciliation. Hence, after reconciliation $RV_x^o[q] = \max(RV_x^o[q], RV_y^o[q])$ and thus $RV_x^o[q] \geq time(a_q^o)$ and the implication holds. If $a_q^o \in \Delta_{y,x}^o$, then $a_q^o \in H_y$ before the reconciliation and via an analogous reasoning the implication holds. □

4.4. Global agreement

Theorem 4 *If* Rec_Info *is empty at each site, all sites have identical values, history log, and reception vectors, and the values reflect the timestamp-ordered execution of all actions originated at any site.*

Proof Theorem 2 ensures that no action is executed more than once at any site and the execution order is equivalent to the timestamp execution order. We now prove that if all *Rec_Info* tables are empty, all actions have been executed at every site. We will prove it by negation.

Suppose that all *Rec_Info* tables are empty and there exist a site x and an action a_q^o such that $a_q^o \notin H_x$. Since $a_q^o \notin H_x$, site x did not commit the transaction in which a_q^o was executed. Hence, site y did not receive an acknowledgement from site x, and the tuple (o, x) was inserted in *Rec_Info* at site y. This entry could only have been deleted upon a reconciliation between sites y and x on object o, where site y commits the reconciliation. Since all the *Rec_Info* tables are empty, such a reconciliation took place. If the reconciliation was also committed at site x, then $a_q^o \in H_x$ from Theorem 2, and we have a contradiction. If site x did not commit the reconciliation, the tuple (o, y) was inserted in *Rec_Info*$_x$. This entry could only have been deleted upon commit of a reconciliation at x on o with y. Again, since from the hypothesis all the logs are empty such a reconciliation must have taken place. Then, from Theorem 2, it must be $a_q^o \in H_x$ and we have derived a contradiction.

In the beginning, all sites had the same data values, history log, and reception vectors. Since the same actions have been executed at each site and all the actions are commutative, we know that the data values and reception vectors are the same and the history logs contain the same information at every site. □

5. Example

In this section we illustrate through a simple example our algorithms. We consider three sites, x, y, and z, and an object o and item i replicated at every site. We assume that initially all copies of the item at each site have value 0, all history logs and *Rec_Info* tables are empty, and reception vector entries are all equal to t_0.

Fig. 5 illustrates the sequence of events (committed transactions or reconciliations) that occurs at each site. Each event is denoted by a transaction-id or reconciliation-id, the timestamp of the transaction/reconciliation, the value of item i, history log H, reception vector RV, and *Rec_Info* table. Fig. 6 illustrates the actions that are performed during the observed time interval on object o; each entry contains a transaction-id, a timestamp, a site-id, an object-id, and the action performed on the object.

We described the following sequence of events. Initially, transaction T_1 is started at site x and executed at all sites. Then, a network partition occurs such that sites x and y cannot communicate with site z. During the partition transaction T_2 originates at site x and is executed at site x and y and transaction T_3 originates and executes at site z. Upon execution of T_2, site x inserts tuple (o, z) in its *Rec_Info* table indicating the need for reconciliating with z on object o. Analogously, upon execution of T_3, site z inserts tuples (o, x) and (o, y) in its *Rec_Info* table indicating the need of reconciling with x and y on object o. Then, site y fails and the partition between sites x and z is repaired. A reconciliation is called between x and z, and the corresponding need for reconciliation is deleted from the *Rec_Info* tables at these sites. Then, transaction T_4 originates at site x and commits at site x and z. Upon execution of the transaction, site x inserts tuple (o, y) in its *Rec_Info* table site x

T-id / Rec-id	timestamp	i	H	RV^o	*Rec_Info*
T_1	t_1	1000	a	$[t_1\ t_0\ t_0]$	
T_2	t_2	1500	ab	$[t_2\ t_0\ t_0]$	$(o,\ z)$
$rec_{x,z}$	t_4	1300	abc	$[t_2\ t_0\ t_3]$	
T_4	t_5	1100	abcd	$[t_5\ t_0\ t_3]$	$(o,\ y)$
$rec_{x,y}$	t_6	1100	abcd	$[t_5\ t_0\ t_3]$	

site y

T-id / Rec-id	timestamp	i	H	RV^o	*Rec_Info*
T_1	t_1	1000	a	$[t_1\ t_0\ t_0]$	
T_2	t_2	1500	ab	$[t_2\ t_0\ t_0]$	
$rec_{x,y}$	t_6	1100	abcd	$[t_5\ t_0\ t_3]$	
$rec_{z,y}$	t_7	1100	abcd	$[t_5\ t_0\ t_3]$	

site z

T-id / Rec-id	timestamp	i	H	RV^o	*Rec_Info*
T_1	t_1	1000	a	$[t_1\ t_0\ t_0]$	
T_3	t_3	800		$[t_1\ t_0\ t_3]$	$(o,\ x),\ (o,\ y)$
$rec_{x,z}$	t_4	1300	acb	$[t_2\ t_0\ t_3]$	$(o,\ y)$
T_4	t_5	1100	acbd	$[t_5\ t_0\ t_3]$	$(o,\ y)$
$rec_{z,y}$	t_7	1100	acbd	$[t_5\ t_0\ t_3]$	

Figure 5. Sequence of events at sites x, y, z

$$\begin{array}{ll} a: & \langle T_1, t_1, X, o, \mathit{credit}(i, 1000)\rangle \\ b: & \langle T_2, t_2, X, o, \mathit{credit}(i, 500)\rangle \\ c: & \langle T_3, t_3, Z, o, \mathit{debit}(i, 200)\rangle \\ d: & \langle T_4, t_4, Y, o, \mathit{debit}(i, 200)\rangle \end{array}$$

Figure 6. Actions executed during observed time interval

indicating the need of reconciling with y on object o. Later on, site y recovers. A reconciliation is executed between x and y, deleting the need for reconciliation stored at site x. Finally a reconciliation is performed between z and y. Notice that this reconciliation does not bring any new information for any of the two sites. The *Rec_Info* tables at all sites are now empty, all sites agree on their status and all the actions that have been executed are reflected in the database.

6. Reconciliation algorithm for non-commutative actions

We now drop the assumption that all actions are commutative, and assume we can have non-commutative actions as well. Commutativity implies that correctness is independent of the order of action execution. Non-commutative actions, however, have to be executed in timestamp order. Therefore, when actions arrive out-of-order, previously executed actions may have to be undone. Hence, we need to add the assumption that every action has an inverse which undoes its effect.

The reconciliation algorithm for non-commutative actions is similar to the algorithm for commutative actions (see Sec. 4.1). In particular, message exchanges and behavior upon failures are the same. The only difference is in the determination of the new value for an object. Instead of directly applying the actions communicated by the other site (in $\Delta^o_{x,y}$), the reconciling site now must ensure that the actions are applied in the correct order, as follows. First it determines the minimum timestamp of all actions it received from the other site. Then it has to undo in reverse timestamp order all actions with timestamp greater than this minimum timestamp. Finally, it merges the actions that were undone and the actions it received, in timestamp order, and applies them. The *Reconcile* procedure of the reconciliation algorithm is presented in Fig. 7; this change is the only difference with the algorithm in Fig. 3.

6.1. Correctness

In Sec. 4.3 the reconciliation algorithm for commutative actions was proven correct. The only place where we used the commutativity of actions, was in the proof of Theorem 2. We now restate that theorem, and prove it for the case of non-commutative actions.

Theorem 5 *If a site x commits a reconciliation on object o with site y, the value of o at x after reconciliation is the value obtained by timestamp-ordered execution of all the distinct actions executed at x and y before reconciliation.*

Proof The first two steps of the proof remain the same (all actions in H_y and not in H_x are communicated to x, and no action which is already in H_x is communicated to x). What remains to be proved, is that after reconciliation the value of o reflects the timestamp-ordered execution of the actions at a site. By assumption, this invariant held before reconciliation; undoing actions at the site x does not affect this invariant. All actions that were undone and the actions communicated by y were merged and sorted in timestamp-order. Since all actions that had a timestamp bigger than the minimum timestamp of the actions communicated by y were undone, after applying the merged and sorted actions, the value of o reflects the timestamp-ordered execution of the actions in H_x and H_y. □

```
**Reconcile** $(x, y, o, RV_y^o)$
$\Delta_{x,y}^o := \{a_q^o \in H_x \mid time(a_q^o) > RV_y^o[q]\}$;
/* all actions executed at site $x$
        and not executed at site $y$ */
Send $\Delta_{x,y}^o$ to site $y$;
Wait for $\Delta_{y,x}^o$ from site $y$;
$t := min(\{time(a_q^o) \mid a_q^o \in \Delta_{y,x}^o\})$
        /*time of the earliest action in $\Delta_{y,x}^o$;*/
$U_x^o := \{a_q^o \in H_x | time(a_q^o) > t\}$;
        /*actions to be undone*/
Sort $U_x^o$ in reverse timestamp order;
Undo all actions in $U_x^o$;
Sort $\Delta_{y,x}^o$ in timestamp order;
$R_x^o$ := sorted merge of $U_x^o$ and $\Delta_{y,x}^o$;
        /*actions to be executed*/
Execute actions in $R_x^o$;
        /* update object value */
$H_x := H_x \cup \Delta_{y,x}^o$;
        /* append actions to the history */
$\forall q : RV_x^o[q] := \max(RV_x^o[q], RV_y^o[q])$;
        /* update reception vector */
```

Figure 7. Reconciliation (non-commutative actions)

7. Log maintenance

To avoid that the history log becomes too large, we can archive the oldest part of it. However, we have to avoid disposing that part of the log that might still be needed for pending reconciliations. The log can be managed as follows.

At each site q and for each object o, an auxiliary structure, called *propagation vector* PV_q^o is kept. There is an entry in PV_q^o for each site of the system. The semantics of the propagation vector is illustrated by the following invariant condition:

Invariant 2 *For all sites x, y, q, objects o, and actions a_q^o:* $\mathrm{time}(a_q^o) \leq PV_x^o[y] \Rightarrow a_q^o \in H_y$.

Thus, entry $PV_x^o[y]$ indicates the time of the latest action executed at site y, such that site x is certain that all actions with a smaller timestamp have already been executed at y. Intuitively, as a reception vector gives information about actions executed at a site itself, a propagation vector gives information about actions executed at all the other sites. Communication of the respective propagation vectors PV^o is included in the message exchange at the beginning of the reconciliation between two sites.

Propagation vectors are updated in the following way. At initialization, all entries are assigned time t_0. At commit at site x of a reconciliation on object o with site y, PV_x^o is updated as follows: $PV_x^o[x] := \min_q(RV_x^o[q])$, and $PV_x^o[q] := \max(PV_x^o[q], PV_y^o[q]) \quad \forall q \neq x$. From Invariant 1 and the way propagation vectors are updated, it is trivial to prove that Invariant 2 holds.

Theorem 6 *The reconciliation algorithm preserves Invariant 2.*

Proof By assumption the invariant held before the execution of the reconciliation. Consider site x which reconciles on object o with site y, we show that the following holds: $\mathrm{time}(a_q^o) \leq PV_x^o[y] \Rightarrow a_q^o \in H_y$ at the end of reconciliation. Consider an action a_q^o such that $time(a_q^o) \leq PV_x^o[y]$. If $y = x$, we have $PV_x^o[x] = \min_q(RV_x^o[q]) \leq RV_x^o[x]$. Hence, from Invariant 1, $a_q^o \in H_x$ and the implication holds. If $y \neq x$, $PV_x^o[q] = \max(PV_x^o[y], PV_y^o[y])$. Hence, before reconciliation either $time(a_q^o) \leq PV_x^o[y]$ or $time(a_q^o) \leq PV_y^o[y]$. Since Invariant 2 held before reconciliation, $a_q^o \in H_y$ and hence the implication is satisfied. $\square$

A cleaning procedure archives part of the log off-line. This procedure is called periodically at each site and removes all the information that is not required anymore, by removing all actions with a timestamp smaller than the minimum entry of the site's propagation vector, as shown in Fig. 8.

Theorem 7 *Algorithm cleanup never removes an action which may be used in a reconciliation.*

Proof To prove that all the actions removed by the cleanup process at site x are no longer needed, we have to prove that they have already been executed at all sites, and that they will not need to be undone.

```
**Cleanup** $(x)$

For all actions $a^o_q \in H_x$
**If** $time(a^o_q) \leq \min_z(PV^o_x[z])$
**then** delete $a^o_q$ from $H_x$
```

Figure 8. Algorithm for cleaning the log

We first prove that all actions that were removed have already been executed at any site. That is, for all sites x, y, and q, objects o and actions a^o_q the following holds: $time(a^o_q) \leq \min_z(PV^o_x[z]) \Rightarrow a^o_q \in H_y$. Consider action a^o_q such that $time(a^o_q) \leq \min_z(PV^o_x[z])$; thus, $time(a^o_q) \leq PV^o_x[y]$. Hence, from Invariant 2 $a^o_q \in H_y$ and the implication is satisfied.

We now prove that the removed actions will not need to be undone at site x. Suppose we remove an action a^o_q at site x that needs to be undone, i.e., not all actions with a smaller timestamp have been executed yet. Therefore, there is some other action a^o_z such that $time(a^o_z) < time(a^o_q)$ and $a^o_z \notin H_x$. Clearly, this situation contradicts the implication we just proved. □

8. Application of reconciliation

In the previous section, we described the basic step of the reconciliation mechanism: reconciliation between two sites on an object o. For application of this basic step, there are several options.

Immediate The basic step of reconciliation is applied immediately after sites that participate in a transaction discover that a reconciliation is required, because the comparison on the reception vector entries fails. In such a case, a reconciliation can be called for the object on which the comparison failed.

Periodically Reconciliation is called periodically, at a given point in time (for instance at midnight) or after fixed time interval (for instance every hour), by applying the basic step to all objects for all pairs of communicating sites. This option is viable in an environment where, e.g., the night is used to bring all sites to agreement, whereas during the day they may "drift apart."

Upon demand Reconciliation is called for all objects and all pairs of communicating sites once a user demands it. This option is appropriate before running a transaction that requires full consistency.

At full connectivity Reconciliation is performed only when the full connectivity of all sites in the system is established. Then we may iterate over all sites and objects and apply the basic step of the reconciliation mechanism.

Note that a full reconciliation involves a quadratic number of binary reconciliations. However, at each binary reconciliation, two reception vectors are updated. This reduces the work in some of the subsequent binary reconciliations and, therefore, the workload of the last reconciliations.

An optimization reduces the number of binary reconciliations to $o(n)$ under the assumption that no failures or partitions of the reconciling sites occur during these reconciliations. All n sites are ordered in a linear chain using their site number. First reconciliations are performed "forward," between sites 1 and site 2, 2 and 3, and so on, until the binary reconciliation between site $n-1$ and site n. At that point, sites n and $n-1$ have all actions executed by all sites in their history log. A subsequent "backwards" execution of reconciliations, starting with sites $n-1$ and $n-2$ and ending with sites 2 and 1, brings all sites to agreement. As all sites now reflect all actions, the local tables *Rec_Info* can be emptied since they call for reconciliations that are not needed.

9. Conclusions

In this paper we described an approach to update propagation in replicated databases that is progressive (the transaction's coordinator always commits) and non-blocking (each site may make unilateral decisions).

In our approach each site s maintains two vectors. The first one, called reception vector, indicates how up to date the site is. The other one, called propogation vector, indicates how up to date other sites are, according to the information avaliable at site s; the information in this vector is used by every site to prune the log and avoid it to grow indefinitely. We then described a reconciliation protocol that sites can follow to update their databases and arrive to an agreement. The reconciliation is preceded by a phase where sites exchange their vectors to determine which part of the log should be exchanged during the reconciliation procedure. This phase avoids unnecessary communication. We have illustrated the behaviour of sites, during reconciliation, during normal operation, and at failures.

Our approach is a viable alternative for those applications that cannot tolerate the overhead and performance degradation induced by synchronous update propagation and atomic transactions (which are implemented in many replicated databases). Examples of such applications include automated teller machine networks, airline reservations, and part inventory control.

Acknowledgments

The work reported in this paper was performed as part of the Fauve project and started while some of the authors were visiting Stanford, and was partially supported by NSF grant IRI-9007753, by the LOGIDATA+ project of Italian CNR, and by the Center for Integrated Systems at Stanford University. The research of

Maurice Houtsma has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences.

References

1. D. Agrawal and A. El Abbadi, "The Tree Quorum Protocol: an Efficient Approach for Managing Replicated Data," in *Proc. of the 16th Int. Conf. on Very Large Data Bases*, Brisbane, Aug. 1990, pp. 243–254.
2. R. Alonso, D. Barbara. H. Garcia Molina, and S. Abad, "Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems," in *Advances in Database Technology–EDBT'88*, J.W. Schmidt, S. Ceri, and M. Missikoff (Eds.), LNCS **303**, 1988.
3. P.M.G. Apers and G. Wiederhold, "Transaction Classification to Survive a Network Partition," Technical report STAN-CS-85-1053, Stanford University, Aug. 1984.
4. D. Barbara and H. Garcia-Molina, "The Case for Controlled Inconsistency in Replicated Data," *Proc. of the Workshop on Management of Replicated Data*, Houston, TX, Nov. 1990.
5. D. Barbara and H. Garcia-Molina, *The Demarcation Protocol: a Technique for Maintaining Arithmetic Constraints in Distributed Database Systems*, in Advances in Database Technology–EDBT'92, LNCS **580**, 1992, pp. 373–397.
6. P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
7. P.A. Bernstein, M. Hsu, and B. Mann, "Implementing Recoverable Requests Using Queues," in *Proc. ACM SIGMOD'90*, Atlantic City, pp. 112–122.
8. S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarati, "A Classification of Update Methods for Replicated Databases," Technical Report STAN-CS-91-1932, Stanford University, October 1991.
9. S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarati, "The case for independent updates," in *Proc. 2nd Workshop on Replicated Data Management*, Monterey, CA, Nov. 1992.
10. S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarati, "Achieving Incremental Consistency among Autonomous Replicated Databases," in *Proc. DS-5, "Semantic Interoperability,"* Lorne, Australia, Nov. 1992.
11. S. Ceri and G. Pelagatti, *Distributed Database Systems*, McGraw-Hill, 1984.
12. M. Colten, "The Sybase Approach to Replicated Data," Oral Presentation, CS347 Course on Distributed Databases, Stanford University, March 1992.
13. A.R. Downing, I.B. Greenberg, and J.M. Peh, "Oscar: an Architecture for Weak-Consistency Replication," in *Proc. Parbase Conference*, 1990.
14. A. El Abbadi, D. Skeen, and F. Christian, "An Efficient Fault-Tolerant Protocol for Replicated Data Management," *Proc. 4th ACM SIGACTSIGMOD Symp. on Principles of Database Systems*, Portland, OR, March 1985, pp. 215–228.
15. C.A. Ellis and S.J. Gibbs, "Concurrency Control in Groupware Systems," *Proc. ACM SIGMOD'89*, Portland, OR, May 1989, pp. 399–407.
16. M.J. Fischer and A. Michael, "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," *ACM SIGACTSIGMOD Symposium on Principles of Database Systems*, 1982, pp. 70–75.
17. H. Garcia-Molina and K. Salem, "Sagas," *Proc. ACM SIGMOD'87*, May 1987.
18. D.K. Gifford, "Weighted Voting for Replicated Data," *Proc. 7th ACM-SIGOPS Symp. on Operating Systems Principles*, Pacific Grove, CA, Dec. 1979, pp. 150–159.
19. R.A. Golding, "Weak-Consistency Group communication and membership," PhD Thesis, University of Santa Cruz, 1992.
20. J.N. Gray and M. Anderton, "Distributed Computer Systems: Four Case Studies," *Proc. of the IEEE*, Vol. 75, No. 5, May 1987.
21. J.N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, August 1992.
22. A. Heddaya, M. Hsu, and W.E. Weihl, "Two Phase Gossip: Managing Distributed Event Histories," in *Information Sciences*, 49(1), 1989, pp. 35–57.

23. M. Hsu and A. Silberschatz, “Unilateral Commit: a New Paradigm for Reliable Distributed Transaction Processing,” in *Proc. 7th Int. Conf. on Data Engineering*, 1991, pp. 286–293.
24. B. Kahler and O. Risnes, “Extending Logging for Database Snapshot Refresh,” in *Proc. 13th Int. Conf. on Very Large Data Bases*, Brighton, England, 1987, pp. 389–398.
25. N. Krishnakumar and A.J. Bernstein, “Bounded Ignorance in Replicated Systems,” in *Proc. ACM-PODS'91*, Denver, CO, May 1991.
26. A. Kumar and A. Segev, “Optimizing Voting-Type Algorithms for Replicated Data,” in *Advances in Database Technology–EDBT'88*, J.W. Schmidt, S. Ceri, and M. Missikoff (Eds.), LNCS **303**, 1988, pp. 428–442.
27. R. Ladin, B. Liskov, and L. Shira, “Lazy Replication: Exploiting the Semantics of Distributed Services,” *Proc. 1st Workshop on Replicated Data*, Houston, TX, Nov. 1990, pp. 31–34.
28. L. Lamport, “Time, Clocks, and Ordering of Events in a Distributed System,” *CACM*, Vol. 21, No.7, July 1978.
29. T. Mostardi and C. Siciliano, “Bitransactions, Relay Races, and their Applications to the Management of Replicated Data,” CRAI Internal Report, S. Stefano di Rende (CS), Italy, Nov. 1990.
30. N. Natarajan and T.V. Laksman, “An Open Architecture Facilitating Semantics Based Transaction Management for Telecommunications Applications,” in *Proc. Int. Workshop on High-Performance Transaction Systems*, Asilomar, Sep. 1991.
31. D.S. Parker, et al., “Detection of Mutual Inconsistency in Distributed Systems,” *IEEE T-SE*, May 1983.
32. C. Pu and A. Leff, “Epsilon-Serializability,” Technical Report No. CUCS-054-90, Columbia University, Jan. 1990.
33. C. Pu and A. Leff, “Replica Control in Distributed Systems: an Asynchronous Approach,” *Proc. ACM SIGMOD'91*, Denver, CO, May 1991.
34. K.V.S. Ramarao, “Transaction Atomicity in the Presence of Network Partition,” in *Proc. 4th Int. Conf. on Data Engineering*, Feb. 1988, Los Angeles, CA, pp. 512–519.
35. A. Reuter and H. Wächter, “The Contract Model,” *IEEE Database Engineering Bulletin* Vol. 14, No. 1, March 1991.
36. S.K. Sarin, C.W. Kaufman, and J.E. Somers, “Using History Information to Process Delayed Database Updates,” *Proc. 12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, Aug. 1986, pp. 71–78.
37. G.T.J. Wuu and A. Bernstein, “Efficient Solutions to the Replicated Log and Dictionary Problems,” in *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984.