Achieving Incremental Consistency among Autonomous Replicated Databases*

Stefano Ceri†
Politecnico di Milano

Maurice A.W. Houtsma‡
University of Twente

Arthur M. Keller§
Stanford University

Pierangela Samarati¶
Universita' di Milano

Abstract

In this paper, we present methods for supporting autonomous updates in replicated databases. Autonomous updates are of particular importance to applications that cannot tolerate the delay and vulnerability due to synchronous update methods (2PC). We separate the notion of replication consistency, meaning that all copies have the same value and reflect the same update transactions, from behavior consistency, meaning that transaction execution reflects all integrity constraints. The method proposed in this paper supports independent updates during network partitioning, and achieves a consistent final database state on recovery of partitions that reflects all actions that were executed during network partitioning. To this purpose, we describe a *reconciliation procedure* that applies all actions to each updated data item in the order in which they were originally performed, possibly independently; therefore, reconciliation may require the undo and redo of actions. We formally define the properties that need to hold for our approach to work, and we prove that our reconciliation procedure respects these properties.

Our approach is incremental, as it can be applied to any sequence of partitionings and recoveries; reconciliation occurs whenever possible or at the user's desire. However, we trade consistent behavior for update availability: in general, there is no guarantee that the execution will reflect all global consistency constraints. Localization techniques for constraints can be used to support consistent behavior for

*This work was performed in the context of the Fauve-project and started while some of the authors were visiting Stanford, and partially supported by NSF grant IRI-9007753

†Stefano Ceri is partially supported by the LOGIDATA+ project of CNR Italy.

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

§Arthur Keller is partially supported by the Center for Integrated Systems at Stanford University.

¶Pierangela Samarati was partially supported by a scholarship from the Rotary Foundation.

particular constraint classes.

Keyword Codes: H.2.7; H.2.8; H.2.m
Keywords: Database Management, Database Administration; Database applications; Miscellaneous

1 Introduction

Replicated databases are becoming more and more of interest lately. There are several reasons for that, such as performance, availability, and autonomy. In distributed database systems, the availability of replicas at each site increases read-availability and thus greatly improves read-performance. Many distributed databases are indeed created as a federation of autonomous and possibly heterogeneous databases, where only a portion of data need to be shared[14]. In such systems it is unacceptable that transactions be blocked when some of the sites fail or become unreachable; however, propagation of updates among the participating sites frequently leads to such blocking.

Many strategies have been developed for propagating updates in replicated databases [6]. An overview of this topic has been given by us in [7], and for reasons of brevity we will not dwell on this topic here but refer the interested reader to [7].

Basically, we can state that atomic updates in replicated databases form a major obstacle to the spreading of distributed database applications. Although many commercial distributed databases support atomic updates through two-phase commit, they have intrinsic disadvantages such as cost, delay, and reduced availability [10]. Moreover, many applications do not require atomic updates [4]. For instance, in airline reservation systems it is often unacceptable that a replica be unavailable while another replica is being updated, or during a network partitioning. As another example, consider a replicated inventory control system, where part descriptions are stored at the various sites of a company producing or selling those parts [14]; this kind of application does not necessarily require immediate propagation of updates, but is inherently discrete and batch-processing oriented. Even some banking applications do not need atomic updates of replicas. Therefore, several protocols have been developed for updating replicated data without the requirement of atomic and synchronous update to each and every replica [1, 8, 11, 13, 16, 17, 20, 21]. Some of these protocols work by transforming global constraints on the data into local constraints that should hold on the replicas; each replica may then independently be updated as long as its local constraints are satisfied [5]. Other protocols recover from violations of global constraints by means of compensating actions [12, 22].

From our classification [7], we noticed that most protocols do not deal successfully with network partition. 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].

In this paper, we develop a strategy for achieving incremental consistency by allowing updates on arbitrary sites during network partitioning; updates may take place on replicas as if the system were in a normal mode of operation. We say that a group of sites is consistent if they have identical database states and action histories. In the environment we envision, with a large number of sites, the complete set of sites might hardly ever be consistent. Partitions and other failures may occur regularly, leading to groups of sites that are group-wise consistent, but not globally consistent.

In this paper we describe mechanisms that allow reconciliation of groups, regardless of the sequence of partitions and reconciliations that have constructed these groups. These mechanisms allow for partial reconciliation: not all groups have to reconcile into one big group. Global consistency is only achieved asymptotically, when a global reconciliation occurs involving all sites. The mechanisms presented in this paper require that applications be action-based (see Section 2.2), and that all sites maintain a complete copy of the history log (which contains the actions that have led to the current database state)[1]. Similar assumptions are made in [23], which uses timestamp-based concurrency control and proposes to immediately apply updates to replicated data in their arrival order, yet possibly restoring inconsistencies when such arrivals violate the timestamp ordering of transactions. The mechanism of [23] does not consider network partitions; it 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. The use of the history log for propagation of updates was suggested in [15],

The paper is organised as followed. In Section 2 we extensively discuss our model of a replicated database and its behavior; this includes discussing partitions, transactions, reconciliation, and their formal properties. In Section 3 we discuss the normal execution of transactions. In Section 4 we give an algorithm for the reconciliation of updates which have independently been executed during a partitions, and prove its correctness. In Section 5 we outline some application scenarios for the proposed approach; in Section 6 we discuss further research issues and draw some conclusions.

2 Model of Replicated Database and its Behavior

The database consists of S sites, each storing a complete replica of the database state; this assumption is merely for convenience and may be relaxed. We assume the existence of a Lamport-style timestamping mechanism that allows us to produce a global ordering of actions executed at different sites, reflecting all the observable precedences between actions [18].

2.1 Model of partitions

Initially, all sites are connected and have the same database state. During operation, failures may occur in the system and one or more sites may become disconnected. In

[1]However we can design extensions to the basic mechanism presented in this paper that enable the deletion of log records; see Section 6.

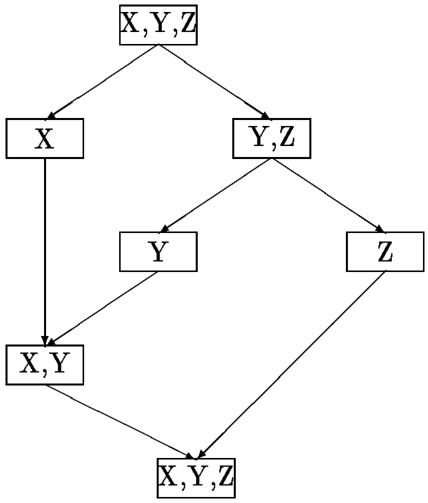

Figure 1: Partition graph

this way *partitions* come into existence; we assume that splitting and merging of partitions are detected by the underlying communication software, thereby creating groups of communicating sites. We further assume that, for any two sites S_1 and S_2, if site S_1 cannot communicate with site S_2 then also site S_2 cannot communicate with site S_1[2]. Consistency is maintained within each group of communicating sites (i.e., they agree on their database state).

We model the behavior of a database system, in terms of partitions and reconciliations, by means of a directed *partition graph*. Nodes of the graph represent groups of communicating sites and edges represent splitting and merging of groups of sites. Nodes are labeled by the site-ids of the sites in the group. Of course, each site-id appearing in a label should also appear in exactly one predecessor node and one successor node.

An example of a partition graph is shown in Figure 1, for a system consisting of three sites: X, Y, and Z. Initially, they can all communicate, but then they partition into two groups, $\{X\}$ and $\{Y, Z\}$. Later on, the group $\{Y, Z\}$ splits. Then groups $\{X\}$ and $\{Y\}$ start communicating again and reconcile. Finally, all sites reconcile and the network becomes fully connected.

2.2 Model of transactions

We envision an environment where copies are available for reading and writing, even in presence of a partition; in that case, it is not possible to guarantee that global constraints always be satisfied. For instance, a local ATM will use its replica to check if a withdrawal of money is allowed, but because of network partitioning its replica may be out-of-date. Only if global constraints can be split into an equivalent system of local constraints,

[2]Though these assumptions are usually supported by communication software, we are currently working on protocols that integrate the detection of splitting and merging of nodes within the normal behavior of transactions, by making use of specific messages; see Section 6.

global constraints may be strictly enforced, as guaranteed for specific replica updates by the Demarcation Protocol [5]; however, truly independent behavior is not achieved anymore. For simplicity, we assume that the ROWA (read one, write all) protocol is used to maintain consistency within a group of sites. This is not affecting write availability: a partition may occur in the context of a write operation, in which case the current partition is split in two new ones and the write operation is executed in one of the new partitions.

We assume that all database processing be described by means of actions; actions are unary operations on a single data item. They may include a parameter, for example, *debit*(*account_no*, 100), and they compute the new value of the data item based on its current value and the parameter. For each action, we assume there exists an inverse operation (for instance, *credit*(*account_no*, 100)). Note that this assumption does not allow for direct assignment of values to items; assignments should be modeled using actions (for instance, instead of assigning the value 0 to an account balance, a withdrawal of all available money on the account should be used).

At all times, each site keeps a *history log* of the actions that have been applied to the initial database state in order to reach its current state. The structure of the log records is as follows:

$$\langle \; \textit{timestamp, item\_id, action, inverse, parameter} \; \rangle$$

As effect on the proposed mechanisms, no duplicate records are kept in the history log of a site.

2.3 Reconciliation after partitioning

When a reconciliation is performed, *representative sites* from merging groups exchange part of their history log. The considered actions are ordered and possible duplicates are removed. Then, the database state is rolled back to an appropriate state and an appropriate sequence of actions is applied to this state, producing a new database state which takes into consideration all actions know by any of the reconciling groups. The history log is then updated to include all the actions which produced the new database state.

To determine the time at which the database states have to be rolled back to during reconciliation, an additional data structure is kept at each site. This data structure is called *reception vector*; it records the extent up to which a site is informed about the other sites. More precisely, given the reception vector R_s for site s, the entry $R_s[p]$ denotes the the time of execution of the last action at site p of which site s is informed. (The idea of using a vector for detecting inconsistency among sites, was proposed before in [19].) Reconciliation is viewed as any action w.r.t. the reception vector, and will therefore be recorded as such in the reception vector. When an action is executed in a partition, all participating sites will update the entries for the participating sites in their reception vector with the time of execution. This time is exactly the same for all sites, and is determined by the originating site for the action; it is propagated to all sites by the "write all" protocol, and installed on each log.

To illustrate the notions of history log and reception vector, we will use Figure 1. We have three sites X, Y, and Z. At each of those sites we have a copy of bank account A

with value 1000, all history logs are empty, and all the entries of the reception vectors are 0. Initially all sites communicate; suppose they executed a deposit of \$500 on account A at time t_0. The values of the data item are then $A_X = A_Y = A_Z = 1500$, the reception vectors are $R_x = R_y = R_z = [t_0, t_0, t_0]$, and the history logs are $H_x = H_y = H_z = \langle t_0, A, \textit{deposit}, \textit{withdrawal}, \$500\rangle$. Then, there is a partition and sites split into the two groups $\{X\}$ and $\{Y, Z\}$. At site X we execute at time t_1 a deposit of \$1000 on account A. The value of the item becomes $A_X = 2500$, the reception vector $R_x = [t_1, t_0, t_0]$, and the history log $H_x = \langle t_0, A, \textit{deposit}, \textit{withdrawal}, \$500\rangle, \langle t_1, A, \textit{deposit}, \textit{withdrawal}, \$1000\rangle$. At time t_2, 6% of interest is posted on the account at site Y and Z, yielding to the item values $A_Y = A_Z = 1590$, the reception vectors $R_y = R_z = [t_0, t_2, t_2]$, and the history logs $H_y = H_z = \langle t_0, A_1, \textit{deposit}, \textit{withdrawal}, \$500\rangle, \langle t_2, A, \textit{post} - \textit{interest}, \textit{undo} - \textit{interest}, 6\%\rangle$. Then, Y and Z partition, and at some later time t_3 sites X and Y reconcile. Upon reconciliation, new actions at sites X and Y are performed so that both sites reflect both updates (according to a mechanism that will be explained in Section 4). The item value becomes $A_X = A_Y = 2650$, the reception vectors become $R_x = R_y = [t_3, t_3, t_2]$, and the logs $H_X = H_Y = \langle t_0, A,$ *deposit, withdrawal, \$500* $\rangle, \langle t_1, A,$ *deposit, withdrawal, \$1000* $\rangle, \langle t_2, A, \textit{post} - \textit{interest}, \textit{undo} - \textit{interest}, 6\%\rangle$,. The final reconciliation between $\{X, Y\}$ and $\{Z\}$ at time t_4 will produce reception vectors $R_X = R_Y = R_Z = [t_4, t_4, t_4]$ and no change to X and Y's database states and logs, while at site Z an update will be performed to reflect the update that was done on site X at time t_1. The log and database state at site Z will thus become equal to those at site X and Y.

Note that although the actions are repeated at reconciliation time, the decision process that led to them is not. Instead, the effect of the original decision that was taken at the first time of execution is adhered too. For instance, if during a partition a customer is allowed to withdraw money because locally the account balance is positive, the effect of this decision cannot be undone. At reconciliation time this effect is recorded but not changed, eventhough the account might globally have been overdrawn at the time of the withdrawal.

2.4 Notation and formal properties

In the remainder we will use the following notation. At each site s we have a status $S_s = \langle R_s, H_s, D_s\rangle$ where:

- R_s is the reception vector; each entry $R_s[p]$ indicates the time of the last action executed at site p that site s is aware of (i.e., it is reflected in the history of s, or it is the time of the last reconciliation with p).

- H_s is the history, ordered according to timestamp, of all the actions known to site s. We indicate with $H_s^{(t_1,t_2]}$ the sequence of all the actions in H_s with timestamp included in the interval $(t_1, t_2]$, from t_1 (excluded) to t_2 (included).

- D_s is the state of the database at site s; i.e., the value of each data item.

There are also two important properties which must be achieved by the algorithms performing reconciliation after partitioning, based on the definition of consistent database state:

Definition 1 *A status $S_s = \langle R_s, H_s, D_s \rangle$ is called consistent iff the following conditions are satisfied:*

1. *The history log H_s of site s is consistent with respect to the reception vector R_s; i.e., it contains all and only those actions that should be known to the site according to the entries in its reception vector. Formally: $\forall p: \quad R_s[p] = t \Rightarrow$ every action executed at site p at a time t_p with $t_p \leq t$ is included in H_s and no action executed at site p at a time t'_p with $t'_p > t$ is included in H_s.*
2. *The database reflects the timestamp-ordered execution of all the actions contained in the history. This is written $D_s = H_s(D_0)$, where D_0 is the initial database state, and $H(D)$ indicates the application of a history H to a database state D.*

We require that at all times the following two properties hold:

Property 1 Status consistency *All sites of the system have a consistent status.*

Property 2 Group agreement *All sites in the same group have the same status, i.e., $\forall p, q \in P : S_p = S_q$*

We will use these properties later on, in Sec.4.1.

3 Transaction execution

In this section, we discuss the execution of transactions within a group of sites in a partition. Then, we will describe how the occurrence of a partition may be detected, and what happens to the execution of transactions in such a case. As indicated in Section 2.2, we assume the ROWA (read one, write all) protocol to maintain consistency within a group of sites (though, as we will see in Section 5, other scenarios are possible).

During normal execution, a transaction T is started at a particular site, which belongs to a group. Necessary locks are obtained at all sites in the group, the transaction is executed, the reception vector is updated with the time of execution, and then the locks are released. This is sketched by the following algorithm.

Algorithm 1 Execution of a transaction within a group

/* Input: Transaction T, current group P, and status S_P. */
/* Output: When 2PC succeeds, an updated status $\bar{S}_{P'}$ reflecting the actions executed at a group of sites $P' \subseteq P$.*/

1. Obtain necessary locks at each site $s \in P$
2. Execute T at each site $s \in P$, update database accordingly, and record actions in history H_s.
3. Update reception vector R_s at each site with the timestamp t_l of the latest action ($\forall p \in P : R_s[p] := t_l$).

4. Perform the Two-Phase Commit protocol with all the sites of the group P. If it is not possible to commit, abort the transaction and rollback the status on each site, including the reception vectors. Then, retry the transaction in a smaller group $P' \subseteq P$ of sites which can communicate with the coordinator. Disconnected sites are autonomously rolled back (with no blocking).

Following this algorithm, upon successful completion of the 2PC-protocol, a transaction is reflected at all sites of a group.

4 Reconciliation algorithms

In this section, we present an algorithm for reconciling partitions, and prove it to be correct. We consider the binary reconciliation of two groups; reconciliation of n groups is achieved as a sequence of binary reconciliations. However, the algorithm may easily be extended to allow for reconciliation of multiple partitions. The algorithm we present is a centralized one: one of the sites behaves as a coordinator, determines the new status and communicates it to the other sites. In [9] we describe several other reconciliation algorithms that are more distributed in nature. If special knowledge is available about the applications, then the algorithm uses such knowledge; for instance, if all the actions are commutative (like deposit and withdrawal in banking applications), then our algorithm may be simplified by omitting to reorder the actions in the history.

Algorithm 2 Centralized Reconciliation of two groups $P = \{p_1, \ldots p_m\}$ and $Q = \{q_1, \ldots, q_n\}$ at time t_r.

1. For each partition choose an arbitrary site as its representative. Let these representatives be p and q respectively.

2. Choose one representative to be the coordinator of the reconciliation process; let this site be p. Let us refer to the other site q as participant.

3. Send the reception vector R_q of q to site p.

4. If $R_p = R_q$ then update the reception vector as in step 7e) and stop the reconciliation process[3].

5. At site p consider all values $R_p[s]$ where $R_p[s] < R_q[s]$; assign to t the minimum of these values. If the above disequation does not hold for any site s, then assign t_r to t. Communicate the value t thus obtained to site q.

6. At the participant site q, assign to T the sequence of actions contained in the history log H_q with a timestamp greater than t; $T = H_q^{(t,t_r]}$. Communicate T to site p.

7. Upon reception of T at site p determine the new status as follows:

[3]If $R_p = R_q$, then the two sites have the same status and no update of history and database is necessary.

(a) Determine the sequence U of actions to be undone as the sequence of actions in the log of p with timestamp greater than t; $U = H_p^{(t,t_r]}$.

(b) Determine the sequence N of transitions to be redone by merging the sequences U and T, in timestamp order and by removing any duplicates that might be present.

(c) Determine the new state of the database $\bar{D}$ by undoing the transition sequence U in reverse order, and then redoing the transition sequence N; $\bar{D} = N(U^{-1}(D_p))$.

(d) Determine the new log $\bar{H}$ by removing U from the history log of p and replacing it with N; $\bar{H} = H_p^{(t_0,t]} \cup N$.

(e) Determine the new reception vector $\bar{R}$ by taking for each entry the maximum value of the reception vectors R_p and R_q; $\forall s : \bar{R}[s] = \max(R_p[s], R_q[s])$ if $s \notin \; P \cup Q$, t_r otherwise.

8. Propagate the new status $\bar{S} =< \bar{R}, \bar{H}, \bar{D} >$ to all the sites participating in the reconciliation and atomically update the status of all the sites taking part in the reconciliation.

The presented algorithm makes no attempt to avoid unnecessary operations; however in an actual implementation, several optimizations are possible that limit the amount of required log scannings and undo/redo of actions. For example, consider step 7a): in an actual implementation, we would only include actions in U with a timestamp greater than the first action communicated by q to p, thus reducing the size of U.

4.1 Correctness of reconciliation

We assume that before reconciliation Properties 1 and 2 hold (i.e., all sites are consistent, and all partitions agree). The correctness of reconciliation is approached gradually: first we will show that the history is treated correctly, then we will show that the produced database state is correct, and finally we will show that the reconciliation vector is updated in the appropriate way. The final theorem is then a trivial composition of these components.

To simplify the proofs of the theorems, we first prove the following lemma from Def. 1:

Lemma 1 *Assume that H_s is consistent w.r.t. R_s; then for every site p and for the minimum value of the reception vector $t = R_s[r]$ it holds that $H_s^{(t_0,t]} \supseteq H_p^{(t_0,t]}$.*

PROOF: Because H_s is consistent w.r.t. R_s, every action executed in 2PC at an arbitrary site p with $t_a \leq t$ is included in $H_s^{(t_0,t]}$. Now suppose there exists an action a executed in 2PC on a site q with $t_a \leq t$ and $a \notin H_s^{(t_0,t]}$. Then $R_s[q] < t$ which is a contradiction. As the only actions included in the history log are the ones executed in 2PC, and the history log contains no duplicates, the lemma is obviously true. $\square$

Theorem 1 History correctness *Given a database where Properties 1 and 2 hold, after reconciliation between two arbitrary partitions P and Q, the history log H_p for each site $p \in P \cup Q$ is the timestamp-ordered sequence of all the actions contained in the history logs H_s of the individual sites $s \in P \cup Q$.*

PROOF: We will prove this theorem in three steps as follows.

1. The resulting history log $\bar{H}$ which is assigned to every site contains all and only those actions a which were contained in the history logs of the sites taking part in the reconciliation: $\forall a : a \in \bar{H} \Leftrightarrow a \in \bigcup_{s \in P \cup Q} H_s$.

2. $\bar{H}$ does not contain any duplicates.

3. $\bar{H}$ is in timestamp order.

Let us now prove the consecutive steps.

1. ($\Rightarrow$) $\forall a : a \in \bar{H} \Rightarrow a \in \bigcup_{s \in P \cup Q} H_s$. In step 7d) of the reconciliation algorithm, the new history $\bar{H}$ is assembled as the merge of $H_p^{(0,t]}$ and N. If a is part of $H_p^{(0,t]}$ the implication is proved. Let us therefore assume that a was part of N. In step 7c) of the reconciliation algorithm, N was assembled by merging $H_p^{(t,t_r]}$ and $H_q^{(t,t_r]}$. Therefore, $a \in H_p^{(t,t_r]} \vee a \in H_q^{(t,t_r]}$ which proves the implication.

   ($\Leftarrow$) $\forall a : a \in \bar{H} \Leftarrow a \in \bigcup_{s \in P \cup Q} H_s$ This implication expresses the fact that the resulting history is complete. Because all the sites in the same group in the partition have exactly the same history according to Property 2, this implication may be simplified into $\forall a : a \in H_p \cup H_q \Rightarrow a \in \bar{H}$, where p and q are the representatives of P and Q respectively.

   Let us suppose that there exists an action a such that $a \in H_p \cup H_q$ and $a \notin \bar{H}$. From step 7) of the reconciliation algorithm we have that $\bar{H}$ is computed as the concatenation of $H_p^{(0,t]}$, $H_p^{(t,t_r]}$, and $H_q^{(t,t_r]}$. If $a \in H_p$ then either $a \in H_p^{(0,t]}$ or $a \in H_p^{(t,t_r]}$ and we have a contradiction. Thus we have $a \in H_q \wedge a \notin \bar{H}$, and we also have $a \notin H_q^{(t,t_r]}$. Therefore, $a \in H_q^{(t_o,t]}$ and thus $t_a \leq t$.

   Now let us suppose that action a originally executed at some site s. As $a \notin H_p$, we know that $R_p[s] < t_a$ (using Property 1 and Def. 1). As $a \in H_q$, we know that $t_a \leq R_q[s]$. We now have derived that $R_p[s] < t_a \leq R_q[s]$. But from the computation of t in step 5 of the reconciliation algorithm we know that $t \leq R_p[s]$, and therefore $t < t_a$. This means we have derived a contradiction, which proves this part of the theorem.

2. Before the start of the reconciliation algorithm, the histories H_p and H_q do not contain any duplicates. The new history $\bar{H}$ is the concatenation of $H_p^{(t_o,t]}$ and N. Obviously, $H_p^{(0,t]}$ does not contain duplicates. In step 7b) of the reconciliation algorithm, N is explicitly assembled to contain no duplicate actions. As for every action $a \in N$ its time $t_a > t$, $\bar{H}$ is guaranteed not to contain any duplicate actions.

3. Again, both $H_p^{(t_o,t]}$ and N are ordered by timestamp, the first by definition, the second because of the algorithm. As all actions in N have an associated time $t_a > t$ the concatenation of $H_p^{(t_o,t]}$ and N is ordered by timestamp. $\square$

Theorem 2 Database correctness *Given a database where properties 1 and 2 hold, after reconciliation between two arbitrary partitions P and Q the new database status produced ($\bar{D}$) is equal to the database status ($\hat{D}$) produced by a timestamp-ordered execution of the actions contained in the history logs of all sites in $P \cup Q$ on the initial database state (D_0).*

PROOF. The new database state is produced by the algorithm as $\bar{D} = N(U^{-1}(D)) = N(H_p^{(0,t]}(D_0))$, where N is the ordered merge of $H_p^{(t,t_r]}$ and $H_q^{(t,t_r]}$. We have to prove that $\bar{D} = \hat{D}$, where $\hat{D} = S(D_0)$ and S is the ordered merge of $\bigcup_s H_s (s \in P \cup Q)$.

Using Property 2, we may rewrite this last expression using the representatives of each partition such that S is now the ordered merge of H_p and H_q. Let us now determine t_i such that $H_p^{(0,t_i]} \supseteq H_q^{(0,t_i]}$ and there does not exist an t_j such that $t_j > t_i \wedge H_p^{(0,t_j]} \supseteq H_q^{(0,t_j]}$. This means that $S = H_p^{(0,t_i]} \cup S'$, where S' is the ordered merge of $H_p^{(t_i,t_r]}$ and $H_q^{(t_i,t_r]}$. If $t_i > t$, we may definitely write S as $H_p^{(0,t]} \cup S'$ and S' is the ordered merge of $H_p^{(t,t_r]}$ and $H_q^{(t,t_r]}$. Therefore, $t_i \leq t$, but if $t_i < t$ Lemma 1 would be violated and thus $t_i = t$. Then, since S' is referred to a time period preceding t, we can write $\hat{D} = S(D_0)$ as $\hat{D} = S'(H_p^{(0,t]}(D_0)) = N(H_p^{(0,t]}(D_0)) = \bar{D}$. This proves the theorem. $\square$

Theorem 3 Reception Vector correctness *Given a database where Properties 1 and 2 hold, after reconciliation between two arbitrary partitions P and Q the history log $\bar{H}$ is consistent w.r.t. $\bar{R}$.*

PROOF: To prove the correctness of reception vector $\bar{R}$ with respect to history $\bar{H}$ we have to prove that for each site s, all actions executed at s before time $\bar{R}[s]$ are contained in $\bar{H}$ and no action executed at s after time $\bar{R}[s]$ is contained in $\bar{H}$. Formally: $\forall s: \ \bar{R}[s] = t \Rightarrow$ every action executed at site s at a time t_s with $t_s \leq t$ is included in $\bar{H}$ and no action executed at site s at a time t'_s with $t'_s > t$ is contained in $\bar{H}$.

Let us prove the first part of the implication. We assume there exists an action a executed at a site s at time t_a and $a \notin \bar{H}$, such that $\bar{R}[s] = t$ and $t \geq t_a$.. Let us suppose $s \in P \cup Q$, then, from Theorem 1, $a \notin \bar{H} \Rightarrow a \notin H_s$ which is a contradiction since a was executed at s and then is contained in the log of s. Let us now suppose $s \notin P \cup Q$. Since $a \notin \bar{H}$, then, from Theorem 1, $a \notin H_p$ and $a \notin H_q$. Then, since the statuses of these sites were consistent before the reconciliation, $R_p[s] < t_a$ and $R_q[s] < t_a$. As in step 7e) of the reconciliation algorithm the maximum of $R_p[s]$ and $R_q[s]$ is taken as entry for $\bar{R}[s]$, $\bar{R}[s] = \max(R_p[s], R_q[s]) < t_a$ and we have a contradiction. From this we have the satisfaction of the first part of the theorem.

Let us now prove last part, i.e, no action a executed at site s at time t_a with $t_a > t$ is in $\bar{H}$. First of all, if $s \in P \cup Q$, then t is the time of reconciliation and it is impossible that $t_a > t$. Let us then consider a site $s \notin P \cup Q$ and suppose that there exists an action a such that $t_a > t$. Since $a \in \bar{H}$, from Theorem 1 either $a \in H_p$ or $a \in H_q$. Then, since the sites were in a consistent state before the reconciliation, either $R_p[s] > t_a$ or $R_q[s] > t_a$.

Then $t = \bar{R}[s] = \max(R_p[s], R_q[s]) > t_a$ which leads to a contradiction. Therefore, such an action cannot exist. This concludes the proof that the reception vector and the history produced by the reconciliation process are consistent. □

Theorem 4 Global correctness *Given a database where Properties 1 and 2 hold, after reconciliation between two arbitrary partitions P and Q properties 1 and 2 still hold.*

PROOF. The proof follows directly from theorems 1, 2, and 3, and from the fact that in step 8 of the reconciliation algorithm the database status obtained at site p is copied to all sites participating in the reconciliation. □

5 Applicability scenarios

The concept of independent updates, and the reconciliation mechanism sketched, can be applied in several ways. Two important characteristics of a system supporting independent updates are: degree of independence, and options for application of reconciliation mechanisms. Let us discuss these in some more detail.

The degree of independence that is allowed to the sites may vary along a gliding scale. An obvious option is not to allow independence at all, requiring all updates to execute in two-phase commit. A second option is to have updates executed in two-phase commit as long as no failures (such as network partition) occur, but allow independent updates once failures occur. A third option is to group sites in a number of more or less autonomous groups; within a group all updates execute in two-phase commit, but groups themselves behave in an autonomous way. A fourth option is to give full autonomy to all sites; updates will then, in general, not execute in two-phase commit but be applied locally.

In each of the cases described above, there are several 'levels of asynchrony'. Application of the reconciliation mechanism may vary, according to the application's need. Let us sketch a few options:

Time-based Reconciliation may be done at regular time-intervals. For instance, reconciliation could be done every night, every hour, etc.

Network driven Reconciliation may be driven by the network; as soon as low-level primitives detect that two or more network partitions are able to communicate, reconciliation may commence. Alternatively, reconciliation may be deferred for some time period. We may wish to keep two groups of sites independent even if they can communicate. One reason is to reduce the number of reconciliations, especially with highly dynamic merging and splitting of partitions; an explicit decision of performing reconciliations may be appropriate for several applications where reconciliation is postponed in order to meet favourable conditions (e.g., reduced computing load in certain times of the day). In this latter case, we simulate a partition graph that has a slower dynamic of splits and merges compared to the partition graph which is based on the ability to communicate.

Operation-based Reconciliation may be done as soon as an operation (application) requires to operate on a consistent state; for instance, a non-commutative operation is requested in the context of systems accepting both commutative and noncommutative operations. An example of this is given by banking applications, where deposits and withdrawals are commutative operations, but posting interest requires a consistent global state.

User demand Reconciliation may be done as soon as a user explicitly demands it, for instance, for infrequent but important applications that require a global consistent state.

6 Further research issues and conclusion

In this paper, we described the support of autonomous updates in replicated databases. Autonomous updates are of particular importance to applications which use replicated data and cannot tolerate the reduced availability due to site failures and network partitions. Indeed, many distributed databases are actually created as a federation of autonomous and possibly heterogeneous databases. We defined the notion of consistent state, meaning that the three components of a state (history, reception vector, and database) are mutually consistent. We then introduced a *reconciliation method* that integrates updates that have taken place independently in such a way that the same consistent state is reached by all sites taking part in the reconciliation. We formulated theorems and proofs, showing that our reconciliation method indeed achieves such a consistent state.

The described reconciliation method allows reconciliation of groups of sites, regardless of the sequence of partitions and reconciliations that constructed these groups; reconciliation can be partial and delayed.

The work reported here gives the foundation to a family of methods for independent updates that we are currently investigating. Other aspects of this research are reported in [9]. Several variations of the reconciliation algorithm described in the paper are possible; each of them leads to a different amount of distributed processing during the reconciliation phase and is justified by some applications. Also, several optimizations on the algorithms are possible, depending on the type of actions encountered; an obvious optimization can be implemented, for instance, for commutative actions, which need not to be undone and redone. A further issue is the minimization of the amount of log information to be kept at each site. By introducing auxiliary data structures, we may able to discard parts of the log that are unnecessary for reconciliation (because this information is already reflected at all sites).

This paper focused on the specification and proofs of correctness of the reconciliation algorithm. In a separate paper, we plan to describe the (lenghty) protocols that are needed by the reconciliation mechanism at a lower level of abstraction, describing the messages exchanges in order to implement it and discussing the details of recovery procedures at all stages of the algorithm, as well as its full integration with the normal transaction execution.

References

[1] D. Agrawal and A. El Abbadi, "The tree quorum protocol: an efficient approach for managing replicated data," in *Proc. 16th Int. Conf. on VLDB*, Brisbane, Aug. 1990, pp. 243–254.

[2] R. Alonso, D. Barbara. H. Garcia Molina, S. Abad, "Quasi-copies: efficient data sharing for information retrieval systems," *Proc. of the Int. Conf. on Extending Data Base Technology, EDBT'88.*

[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*, CS-TR-320-91, Princeton University, April 1991.

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

[7] S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarati, "A Classification of Update Methods for Replicated Databases," STAN-CS-91-1932, Stanford University, October 1991.

[8] S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarati, "The case for independent updates," in *Proc. 2nd Workshop on the Management of Replicated Data,* Monterey, CA, Nov. 1992.

[9] S. Ceri, M.A.W. Houtsma, A.M. Keller, and P. Samarati, "Independent updates and incremental consistency in replicated databases," in preparation.

[10] S. Ceri and G. Pelegatti, *Distributed database systems*, McGraw-Hill.

[11] A. El Abbadi, D. Skeen, F. Christian, "An efficient fault-tolerant protocol for replicated data management," *Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems,* Portland, OR, March 1985, pp. 215–228.

[12] H. Garcia-Molina and K. Salem, "Sagas," *Proc. ACM SIGMOD'87*, May 1987.

[13] 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.

[14] J.N. Gray and M. Anderton, "Distributed computer systems: four case studies", *Proc. of the IEEE*, Vol. 75, No. 5, May 1987.

[15] 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.

[16] N. KRISHNAKUMAR AND A.J. BERNSTEIN, "Bounded ignorance in replicated systems," in *Proc. ACM-PODS'91*, Denver, CO, May 1991.

[17] 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.

[18] L. LAMPORT, "Time, clocks, and ordering of events in a distributed system", CACM, Vol. 21, No.7, July 1978.

[19] D.S. PARKER, ET AL., "Detection of mutual inconsistency in distributed systems," *IEEE T-SE*, May 1983.

[20] C. PU AND A. LEFF, "Epsilon-Serializability," Technical report No. CUCS-054-90, Columbia University, Jan. 1990.

[21] C. PU AND A. LEFF, "Replica control in distributed systems: an asynchronous approach,", *Proc. ACM SIGMOD'91,* Denver, CO, May 1991.

[22] A. REUTER AND H. WÄCHTER, "The contract model," *IEEE Database Engineering bulletin* Vol. 14, No. 1, March 1991.

[23] 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.