Multiversion Reconciliation for Mobile Databases*

Shirish Hemant Phatak
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
e-mail:phatak@cs.rutgers.edu

B. R. Badrinath
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
e-mail:badri@cs.rutgers.edu

Abstract

As mobile computing devices become more and more popular, mobile databases have started gaining popularity. An important feature of these database systems is their ability to allow optimistic replication of data by permitting disconnected mobile devices to perform local updates on replicated data. The fundamental problem in this approach is the reconciliation problem, i.e. the problem of serializing potentially conflicting updates performed by local transactions on disconnected clients on all copies of the database. In this paper we introduce a new algorithm that combines multiversion concurrency control schemes on a server with reconciliation of updates from disconnected clients. The scheme generalizes to multiversion systems, the single version optimistic method of reconciliation, in which client transactions are allowed to commit on the server iff data items in their read sets are not updated on the server after replication.

1. Introduction

Mobile databases are gaining popularity and are likely to do so well into the future as portable devices become more and more popular and common. One key feature of these database systems is their ability to deal with disconnection. Disconnection refers to the condition when a mobile system is unable to communicate with some or all of its peers. In such a situation the mobile no longer has access to shared data. To deal with the disconnection problem, optimistic replication approaches have become exceedingly common. In such approaches, the mobile unit is allowed to locally replicate shared data and to operate on this data while it is disconnected. The local updates can be propagated to the rest of the system on reconnection. However, since the local updates potentially conflict with other updates in the system, schemes to detect and resolve such conflicts are required. This paper focuses on one such scheme.

The architecture we consider is an extended client server architecture. The primary copies of all data items are stored on the server. All transactions must commit on the server to be "globally" committed. The clients are allowed to locally replicate a subset of a database state (as defined in section 2). Local transactions on the client can operate on this local replica and perform updates. As long as the client is connected to the server, each local transaction is automatically serialized on the server before it is allowed to commit. However, if the client is disconnected and cannot access the server, local transactions are allowed to "locally" commit in the sense that their updates are made available to other local transactions after they have locally committed. The client, on reconnection, propagates all local transactions to the server for globally serializability testing. A transaction that can't be serialized due to irresolvable conflicts must be aborted. If serialization succeeds, the transaction is globally committed and its updates applied to the shared database. We refer to the process of testing, conflict resolution and serialization as reconciliation.

Most of the current approaches to the reconciliation problem assume a lot of specialized knowledge on the part of the application and even the user. Furthermore, all such approaches are highly tied to specific applications and are difficult to generalize. In this paper, we attempt to strike a balance between requiring highly specialized knowledge on part of the applications and providing generic consistency management. The cornerstone of our approach is using multiversioning. This allows us to reconcile transactions in the "past", that is on versions of data older than the current version, improving the probability of reconciliation. Multiversioning also allows us to use snapshot isolation (see Berenson et. al. in [4]), which provides consistency that is almost as strong as ANSI read committed, but which is weaker than full serializability. Snapshot isolation allows a multiversion transaction to commit as long as the data items in its write

*This research work was supported in part by DARPA under contract numbers DAAH04-95-1-0596 and DAAG55-97-1-0322, NSF grant numbers CCR 95-09620, IRIS 95-09816 and Sponsors of WINLAB.

set were not overwritten after these data items were read by this transaction (i.e. there was no intervening write on these data items after the transaction read them). Using snapshot isolation additionally improves the probability of reconciliation, without significant loss of consistency. We present a reconciliation scheme which integrates incoming client transactions and provides snapshot isolation for such transactions. The server must provide at least snapshot isolation to server transactions. Finally, we validate our algorithm using the phenomena and the anomalies presented by Berenson et. al. in [4]. The scheme is a generalization of the single version optimistic concurrency control (see [14]), where a client transaction is deemed globally serializable iff no item in its read set was written on the server after the client downloaded its local replica.

1.1. Benefits of Multiversioning

To see how multiversioning helps in the reconciliation/reintegration process, consider the following example history (for the single version system, we ignore the subscripts on x and y):

Server: $w_0[x_0] = 1, w_0[y_0] = 1, c_0, r_1[x_0] = 1, r_1[y_0] = 1, w_1[x_1] = 2, c_1$
Client: downloaded $x = 1, y = 1$: $r_{1'}[x] = 1, r_{1'}[y] = 1, w_{1'}[y] = 3, c_{1'}$, reconcile $T_{1'}$

Let us assume that no conflict resolution protocols have been defined. In a single version system, at the time of reintegration, the database has the following snapshot: $\{x = 2, y = 1\}$. Since this snapshot is not consistent with the read snapshot of $T_{1'}$, which is $\{x = 1, y = 1\}$, $T_{1'}$ must be rejected by the server to maintain consistency.

Now let us consider a multiversion system. In this case there are two distinct snapshots in the database: $\{x_0 = 1, y_0 = 1\}$ and $\{x_1 = 2, y_0 = 1\}$. The first snapshot is in the past and is consistent with the read set of $T_{1'}$. Thus, $T_{1'}$ can now be potentially serialized on this snapshot. A sample resulting history is as follows:

Server: $w_0[x_0] = 1, w_0[y_0] = 1, c_0, r_{1'}[x_0] = 1, r_{1'}[y_0] = 1, r_1[x_0] = 1, r_1[y_0] = 1, w_{1'}[y_1'] = 3, c_{1'}, w_1[x_1] = 2, c_1$

Note that this history is not serializable, but snapshot isolation holds. Thus, by using snapshot isolation, rather than full serializability, we can improve the probability of reconciliation, even when conflict resolution protocols are not defined. Note that the reason this works for our example is that transaction T_1 on the server does not write y. If it does, snapshot isolation would no longer hold.

1.2. Related work

Recently lot of research has been directed at optimistic replication schemes and at mobile databases and reconciliation. Some early work can be found in [6, 3, 10, 11, 12]. Recently, Gray et. al. in [8] present a system architecture and a replication model for mobile databases. The database here is a collection of replicated objects with primary copies at certain sites known as object masters. The model uses a two tier replication scheme, with one tier on the mobile (disconnected) nodes and the other on the base (connected) nodes. All transactions operating on objects on the first tier are considered tentative and must be reapplied to the object master and the second tier, using a set acceptance criteria, whenever the mobile reconnects.

In BAYOU [7], the replicas are local copies of an *entire* data repository. Bayou requires applications to specify both the conflict detection and resolution functions with the updates. Updates are reconciled whenever two data repositories get connected. Reconciliation takes place by rolling back all updates made by both connecting servers and replaying both sets of updates together (including conflict detection and resolution functions) in timestamp order. Note that the data repository here is not a database.

An approach to the reconciliation problem that relies on application semantics is fragmentable objects [16]. Here the designer can exploit application semantics to split large and complex objects into smaller fragments. The mobile can then independently operate on an object partition consisting on one or more object fragments with certain constraints. On receiving a request from the mobile for caching an object, the server attempts to split the object into fragments so that it can satisfy the request with an object partition. The mobile can then operate on this partition in disconnected mode.

Note that most of the above work does not focus on the conflict resolution problem. On the other hand, approaches like BAYOU and fragmentable objects which do focus on reconciliation require specialized knowledge of the system for all transactions and special concurrency control models. We attempt to reach a balance between these solutions by creating a system that provides consistency management and that can use but does not need specialized knowledge to provide conflict resolution.

1.3. Organization of the paper

We have organized the remainder of this paper as follows: section 2 describes our model of the database; section 3 describes the multiversion reconciliation algorithm and discusses consistency properties; section 4 explores some practical concerns with the implementation of the algorithm and section 5 briefly lists our conclusions.

2. The model

We model the database as a collection of data items (e.g. tuples or objects) along with their versions. The data items are drawn from a universe X. The database at any given instant exists in a particular state. This state reflects the updates due to all committed transactions upto this instant. In practice, the database would also consist of data written by active transactions. However, we ignore this for the purposes of our algorithm, which only attempts to serialize transactions against committed versions. Each (globally) committed transaction that operated on the data is provided a unique nonnegative integral commit timestamp on (globally) committing on the server, which can be different from a possibly non-unique start timestamp, if such a start timestamp is provided. The timestamps reflect the order in which the transactions commit. The version number of a version of a data item is the commit timestamp of the transaction that wrote it. Note that as each transaction commits on the server (which means it globally commits), the database state D "expands" to include the updates performed by the transaction. Since this also includes reconciling client transactions, D progressively changes through the entire reconciliation process as individual client transactions globally commit.

Definition 1 (Database State and Value Function) *The state of the database on the server at any instant of time consists of a set of elements $D \subset X \times \mathbb{Z}^+$, a collection of a finite number of versions of data items drawn from X and a* value $: D \rightarrow \bigcup_{x \in X} \text{domain}(x)$ *function that maps data items and versions to the actual values of the data items. The domain of the value function at any instant of time is the set D at that instant of time, and the co-domain is the union of the domains of all the data items in X.*

Our characterization of snapshots is using snapshot functions. There are two kinds of snapshots: version snapshots consisting of data items and their versions (and hence implicitly their values using the *value* function), and value snapshots consisting purely of the data items and their values. Note that snapshots seen on a disconnected client may have no correspondence to snapshots on the server and hence a value snapshot from a client may not correspond to any version snapshot on the server. This is because transactions executing on the client are completely independent of transactions executing on the server as long as the client is disconnected. Hence the need for two separate definitions.

Definition 2 (Version Snapshot Function) *A version snapshot function $S : \mathbb{Z}^+ \rightarrow D$ is a function that maps a nonnegative integral timestamp into a snapshot of the database. For each timestamp value it yields a collection of versions of data items with the following properties:*

1. $S(v) \subseteq D$. *Every snapshot is a subset of the database state.*
2. $\langle x, v\rangle \in D \implies \langle x, v\rangle \in S(v)$. *For any given snapshot $S(v)$, if the data item was written by transaction with timestamp v, then version v of the data item is in $S(v)$.*
3. $\langle x, v'\rangle \in S(v) \implies v \geq v'$. *No data item in $S(v)$ can have version number greater than v.*
4. $\forall v, v' : \langle x, v'\rangle \in S(v) \wedge \langle x, v''\rangle \in S(v) \implies v' = v''$. *Only one version of a data item can be present in a snapshot.*
5. $\forall v', v'' : v' > v'' \wedge \langle x, v'\rangle \in S(v) \wedge \langle y, v''\rangle \in S(v) \implies [\forall v''' \in \mathbb{Z} : \langle y, v'''\rangle \in D \Rightarrow v''' < v'' \vee v''' > v']$. *A snapshot $S(v)$ has the latest versions of the data items that are less than or equal to v. Note that if a data item x was introduced into the database by a transaction with timestamp greater than v (i.e. x was inserted into the database by the transaction), then x can not be in $S(v)$.*

From a computational perspective, $S(v)$ is defined as follows (lub is the least upper bound and is equivalent here to the maximum function): $S(v) = \{\langle x, v'\rangle \mid \langle x, v'\rangle \in D \wedge v' = lub\{v''|\langle x, v''\rangle \in D \wedge v'' \leq v\}\}$. $S(v)$ is in fact the snapshot of the database that a read-only transaction with (start) timestamp v would see. Additionally, we also define value snapshots S^v, which consists data items along with their values (instead of their versions). We use the superscript v to distinguish value snapshots from version snapshots. Note that the value snapshots need not have corresponding version snapshots (especially if the value snapshots originate from a disconnected client). However, corresponding to every version snapshot $S(v)$ there exists a value snapshot $S^v(v) = \{\langle x, i\rangle \mid \langle x, v'\rangle \in S(v) \wedge i = \textit{value}(x, v')\}$. In order to simplify our exposition, we also extend the *value* function to version snapshots as follows: $\textit{value}(S) = \{\langle x, i\rangle \mid \langle x, v'\rangle \in S \wedge i = \textit{value}(x, v')\}$. Note that only one value for each data item can be present in the value snapshot.

As defined, a snapshot $S(v)$ must contain all data items x, such that $\langle x, v'\rangle \in D \wedge v' \leq v$. However, we might be interested in subsets of such snapshots. Therefore, we also allow partial snapshots where this requirement is relaxed. Instead of requiring all data items in the data base to be tested, we can restrict our attention to any fixed subset of X. This is useful in characterizing the readsets and writesets of transactions which contain subsets of data items from the database.

The concurrency control model on the server is assumed to be modular as described by Agrawal and Sengupta in [2] and Bernstein and Goodman in [5]. Here read only (query) transactions run in a nonblocking fashion using old versions of the data, while read write (update) transactions always operate on the latest snapshot in existence when they start executing. Any optimistic multiversion concurrency control protocol that guarantees at least snapshot isolation can be used for the update transactions on the server. Requiring that the protocol be optimistic simplifies our description of the algorithm. With a little care, however, our algorithm can easily be re-engineered to work with pessimistic mul-

tiversion concurrency control protocols. In that latter case we have to be careful while serializing client transactions on snapshots that are in "use" by active transactions on the server.

The mobile clients can use any concurrency control protocol (multiversion or otherwise) as long as the read and write sets (and the values read and written) of all locally committed client transactions are available during reconciliation.

Definition 3 (Read, Write and Read-Write Sets) *For each transaction T we define three partial value snapshots: the* $\text{RSET}^v(T)$*, consisting of all data items which were read but not written by T; the* $\text{RWSET}^v(T)$ *consisting of all data items read and then written by the transaction; and the* $\text{WSET}^v(T)$*, consisting of all the data items that are written by T before they are read (i.e. T blind writes all the data items in this set).*[1] *We also define* $\text{READSET}^v(T) = \text{RSET}^v(T) \cup \text{RWSET}^v(T)$ *and* $\text{WRITESET}^v(T) = \text{RWSET}^v(T) \cup \text{WSET}^v(T)$.

To deal with conflicts we require conflict resolution functions. For each client transaction T_c, the client can optionally define a conflict resolution function CR_{T_c}. This function can be provided by the client whenever it seeks reconciliation of a transaction. If this function is not provided by the client the server uses a default. CR_{T_c} takes as input three value snapshots: the readset consisting of data items and values read by T_c, the writeset and values generated by T_c and a new values snapshot against which T_c needs to be serialized. The CR_{T_c} function returns a new writeset for T_c, i.e. a value snapshot of the data items along with new values, after conflicts have been resolved. Thus, $CR_{T_c}(\mathit{READSET}^v(T_c), \mathit{WRITESET}^v(T_c), S^v_{in}) = \mathit{NEWWRITESET}^v$, wherever the function is defined. Note that $\mathit{NEWWRITESET}^v$ only consists of data items that would actually be written if conflict resolution took place for T_c against S^v_{in}. If the conflict resolution function is defined, the client must also define a cost function C_{T_c} that takes the same inputs as the conflict resolution function and returns an integral cost value that indicates the cost of resolving conflicts for that set of inputs. An example cost function is:

$$C_{T_c}(\mathit{READSET}^v(T_c), \mathit{WRITESET}^v(T_c), S^v_{in}) = |\mathit{READSET}^v(T_c) - \mathit{SREADSET}^v(T_c)|$$

where $\mathit{SREADSET}^v(T_c) = \{\langle x, i\rangle \mid \langle x, i\rangle \in S^v_{in} \wedge \exists i' : \langle x, i'\rangle \in \mathit{READSET}^v(T_c)\}$. Here, the cost of resolving against a snapshot is equal to the number of data items which have different values in the snapshot and in the readset of T_c. If the client does not specify these functions, the server uses defaults defined as follows:

[1] The concept of a WSET^v comes largely from the work by Agrawal and Krishnamurthy [1], which adapts multiversion concurrency control to write-only transactions.

Definition 4 (Default Cost Function)

$$C_{T_c}(\text{READSET}^v(T_c), \text{WRITESET}^v(T_c), S^v_{in}) = \begin{cases} 0 & \textit{if } \text{READSET}^v(T_c) \subseteq S^v_{in} \\ \infty & \textit{otherwise} \end{cases}$$

Definition 5 (Default Conflict Resolution Function)

$$\text{CR}_{T_c}(\text{READSET}^v(T_c), \text{WRITESET}^v(T_c), S^v_{in}) = \begin{cases} \text{WRITESET}^v(T_c) & \textit{if } \text{READSET}^v(T_c) \subseteq S^v_{in} \\ \text{undefined} & \textit{otherwise} \end{cases}$$

These functions simply allow serialization of a transaction iff the readset seen by the transaction from the snapshot it read from and the snapshot S^v_{in} against which it is being serialized are identical. The cost function will usually be infinity whenever the new input snapshot S^v_{in} is empty. This, however, would not be the case for write-only transactions [1]. These default functions are used by the system whenever no explicit conflict resolution is requested by the client. Note that our choice of inputs for the conflict resolution function closely parallels the framework defined by BAYOU. In essence, this function can be thought of as re-executing the client transaction T_c on the new snapshot S^v_{in}.

3. The multiversion reconciliation algorithm

We now describe the multiversion reconciliation algorithm. The basic idea is to consider each client transaction in turn and compute a snapshot which is consistent and leads to least cost reconciliation. The snapshots are computed by considering each timestamp that could be provided to the client transaction in increasing order. Furthermore, after conflict resolution the resulting writeset should not affect any snapshot already present in the database. This is done by ensuring that all writes in the new writeset are always serialized after the latest version of the data item or just before a blind write to the data item. A blind write to a data item occurs when a transaction writes the data item without first reading it. The algorithm proceeds by progressively computing snapshots and attempting to serialize client transactions against these snapshots. To achieve these two goals (i.e. least cost reconciliation and snapshot isolation), we define two snapshot functions, the backward or the normal snapshot function $S^{\downarrow}(v)$ and the forward or reverse snapshot function $S^{\uparrow}(v)$. The domain of these functions is the integral domain $\mathbb{Z}^+$ extended by a special element Δ to form a new domain $\mathbb{Z}^\Delta$ with the following properties (note that we could instead work with real timestamps):

1. $\forall i \in \mathbb{Z}^+ : i - \Delta < i$
   $\forall i \in \mathbb{Z}^\Delta : i - \Delta < i$
2. $\forall i \in \mathbb{Z}^+ : j < i \Longrightarrow j < i - \Delta$
   $\forall i \in \mathbb{Z}^\Delta : j < i \Longrightarrow j \leq i - \Delta$
3. $\forall i \in \mathbb{Z}^\Delta, \forall k \in \mathbb{Z}^+ : i - \frac{\Delta}{k} < i - \frac{\Delta}{k+1}$

Note that we assume that $\frac{\Delta}{1} = \Delta$. We also extend the set of allowable timestamps and versions to the domain $\mathbb{Z}^{\Delta}$. The two snapshot functions can now be defined as follows (glb is the greatest lower bound and is equivalent to the minimum function in the integral domain):

- $S^{\downarrow}(k) = \{\langle x, v' \rangle \mid \langle x, v' \rangle \in D \wedge v' = lub\{v'' | \langle x, v'' \rangle \in D \wedge v'' \leq k\}\}$
- $S^{\uparrow}(k) = \{\langle x, v' \rangle \mid \langle x, v' \rangle \in D \wedge v' = glb\{v'' \mid \langle x, v'' \rangle \in D \wedge v'' \geq k\}\}$

The algorithm uses the normal snapshot to determine the input snapshot of the transaction. The reverse snapshot is used to determine whether the transaction's updates can be serialized against the normal snapshot. In some sense $S^{\uparrow}(k)$ can be considered a reverse time snapshot, i.e. a snapshot of the database, if time were reversed. Also note that $S^{\downarrow}(k)$ and $S(k)$ are identical in the integral domain. We also define a version set V that consists of all versions in use in the database when the client reconciles its local replica. Thus, $V = \{v | \exists x : \langle x, v \rangle \in D\}$. Note that $V \subset \mathbb{Z}^{\Delta}$.

The basic building block of our algorithm is a procedure to reconcile the updates of a single client transaction. This procedure is illustrated in algorithm 1. We assume that this algorithm executes atomically[2]. (We show how to relax the atomicity requirement in section 4.2.) The algorithm can then be used as a subroutine to reconcile the updates of all reconciling client transactions on the server as illustrated in Algorithm 2. Note that all snapshot and read/writeset computations are performed with reference to the current transaction being reconciled, i.e. T_c and the current state of the database D. In a fashion analogous to Gray et. al.'s model in [8], the client may specify the C_{T_c} and the CR_{T_c} functions for each client transaction T_c.

There are a few things to note about the algorithm. As long as the conflict resolution and cost functions cover most cases, T_c would rarely be aborted. This is because in our model $S^{\uparrow}(1 + lub(V) - \frac{\Delta}{i})$ is always an empty set for any integral i. Thus, the conditions in the inner for loop would always be satisfied for this snapshot. This situation corresponds to serializing the transaction against the current (latest) snapshot of committed versions. The current snapshot is defined to be $S^{\downarrow}(\infty) = S^{\downarrow}(1 + lub(V))$. Also note that the variable $iter$ is static and its value is retained across invocations of the algorithm. It allows the algorithm to find "gaps" in the timestamp sequence.

The serialization operation at line 30 corresponds to introducing elements in *NEWWRITESET*^{v} into the database with timestamp $opt - \frac{\Delta}{iter}$. Thus, the operation replaces D with

[2]Actually, read-only transactions can continue executing. This is because any snapshot on the server is guaranteed to be "undisturbed" due to the monotonicity properties that must be guaranteed by the algorithm for any set of snapshots of the current database state.

Algorithm 1 Multiversion Reconciliation Algorithm

Ensure: *// Inputs: Database state D, Transaction T_c, Functions* CR_{T_c}, C_{T_c}
Ensure: $opt = -1$
Ensure: $cost = \infty$
Ensure: static $iter = 1$
1: *// Main Loop:*
2: **for all** $v \in V \cup \{1 + lub(V)\}$ in increasing order **do**
3: $S^{\downarrow} = S^{\downarrow}(v - \frac{\Delta}{iter})$
4: $S^{\uparrow} = S^{\uparrow}(v - \frac{\Delta}{iter})$
5: *// Now find if this snapshot is best cost and use-able*
6: **if** $C_{T_c}(READSET^v(T_c), WRITESET^v(T_c), value(S^{\downarrow})) < cost$ **then**
7: *NEWWRITESET*^{v} $= CR_{T_c}(READSET^v(T_c), WRITESET^v(T_c), value(S^{\downarrow}))$
8: *// NEWWRITESET should not conflict with any existing snapshot in D*
9: *// Inner Loop:*
10: **for all** $\langle x, i \rangle \in$ *NEWWRITESET*^{v} **do**
11: *// Is this element* $\in S^{\uparrow}$ *? If yes was it produced by a blind write?*
12: **if** $\exists v' : \langle x, v' \rangle \in S^{\uparrow}$ **then**
13: Let T be the transaction with timestamp v'
14: **if** $\exists i :< x, i > \in$ *WSET*$^v(T)$ **then**
15: *// $\langle x, v' \rangle$ is the result of a blind write*
16: continue Inner For Loop
17: **else** *// This snapshot cannot be used, continue with the main loop*
18: continue Main Loop
19: **end if**
20: **end if**
21: **end for**
22: $cost = C_{T_c}(READSET^v(T_c), WRITESET^v(T_c), value(S^{\downarrow}))$
23: $opt = v$
24: **end if**
25: **end for**
26: **if** $cost = \infty$ **then**
27: *// unable to reconcile updates*
28: $abort\ T_c$
29: **else** *// Reconciliation successful*
30: $serialize\ T_c$ with timestamp $opt - \frac{\Delta}{iter}$
31: $iter = iter + 1$
32: **end if**

Algorithm 2 The Reintegration Algorithm

// Called for each reconciling client C
// Note that the database state D changes for every invocation of Algorithm 1
for all Transactions T_c on C **do**
// The transactions would usually be scanned in the order in which they committed on the client, but this is not a requirement of the algorithm per se.
Begin-Atomic
Invoke Algorithm 1 with D, T_C, CR_{T_c} and C_{T_c}
End-Atomic
end for

$D \cup \{\langle x, opt - \frac{\Delta}{iter}\rangle \mid \exists i : \langle x, i\rangle \in \textit{NEWWRITESET}^v\}$
and *value* with
$\textit{value} \cup \{\langle x, opt - \frac{\Delta}{iter}\rangle \rightarrow i \mid \langle x, i\rangle \in \textit{NEWWRITESET}^v\}$
(we use the $\rightarrow$ instead of an ordered triple for clarity). Also note that the value of $iter$ is remembered between subsequent invocations on algorithm 1.

Even if a client transaction T_c is aborted, client transactions that read dirty data from it need not be. Consider the following example history on a single data item database:

Server: $w_0[x_0] = 1, c_0, r_1[x_0] = 1, w_1[x_1] = 2, c_1, r_2[x_1] = 2, w_2[x_2] = 3, c_2$
Client: downloaded $x_0 = 1$: $r_{1'}[x] = 1, w_{1'}[x] = 3, c_{1'}, r_{2'}[x] = 3, c_{2'}$

Note that, in this case, the client is not following a multiversion scheme. In the absence of explicit conflict resolution, transaction $T_{1'}$ on the client must be rejected on reintegration. This is because the $T_{1'}$ can only be serialized before T_1 on the server but the *NEWWRITESET*^{v} that the default (trivial) conflict resolution handler produces is $\{\langle x, 3\rangle\}$. However, the snapshot $S^{\uparrow}(1-\Delta)$ is $\{\langle x, 1\rangle\}$ in which $\langle x, 1\rangle$ is not produced by a blind write[3].

Even if $T_{1'}$ is aborted, however, $T_{2'}$ can be serialized with timestamp 3 (actually $3 - \Delta$) giving a serial history (we do not assume any intertransactional dependencies):

Server: $w_0[x_0] = 1, c_0, r_1[x_0] = 1, w_1[x_1] = 2, c_1, r_2[x_1] = 2, w_2[x_2] = 3, c_2, r_{2'}[x_3] = 3, c_{2'}$

Note that this is because $T_{2'}$ sees the same value snapshot on the server that it sees on the client. Thus, cascaded aborts would normally not be forced on the client.

3.1. The phenomena and the anomalies

We now validate algorithm 1 using the phenomena and anomalies as specified by Berenson et. al. in [4]. We do this to show that snapshot isolation is indeed provided by the overall system. We briefly recapitulate the relevant phenomena and anomalies analyzed in that paper in terms of histories allowed by the phenomenon or the anomaly:

- **Phenomena P0 (Dirty Write)**
  $w_1[x] \ldots w_2[x] \ldots (c_1 \vee a_1)$
- **Phenomena P1 (Dirty Read)**
  $w_1[x] \ldots r_2[x] \ldots (c_1 \vee a_1)$
- **Phenomena P2 (Fuzzy Read)**
  $r_1[x] \ldots w_2[x] \ldots (c_1 \vee a_1)$
- **Phenomena P3 (Phantom)**
  $r_1[P] \ldots w_2[y : P(y)] \ldots (c_1 \vee a_1)$
- **Anomaly A3 (Phantom)**
  $r_1[P] \ldots w_2[y : P(y)] \ldots c_2 \ldots r1[P] \ldots c_1$
- **Phenomena P4 (Lost Update)**
  $r_1[x] \ldots w_2[x] \ldots w_1[x] \ldots (c_1 \vee a_1)$
- **Anomalies A5 (Data Item Constraint Violation)**
  - **Anomaly A5A (Read Skew)**
    $r_1[x] \ldots w_2[x] \ldots w_2[y] \ldots c_2 \ldots r_1[y] \ldots (c_1 \vee a_1)$
  - **Anomaly A5B (Write Skew)**
    $r_1[x] \ldots r_2[y] \ldots w_1[y] \ldots w_2[x] \ldots (c_1 \wedge c_2)$

Here the $o_i \vee o_j$ implies one or both of o_i and o_j occur in any order, while $o_i \wedge o_j$ implies both of o_i and o_j occur in any order.

In order to provide snapshot isolation the system must disallow histories with phenomena and anomalies **P0, P1, P3, A3, A5A** and **P4**. Since we assume that the system inherently guarantees at least snapshot isolation for server transactions, we need to concentrate only on reconciling client transactions.

Since we only operate on committed data on the server it is obvious that the reconciliation algorithm does not allow phenomena **P1** or **P2**. Interestingly enough in the form in which we have presented the algorithm, it does allow anomaly **A3**. To see this consider a client transaction that used a predicate P to read data on the client. Suppose this read produces 3 data items x, y and z. Thus, in effect the readset of the transaction is $\{x, y, z\}$. Furthermore, suppose that no conflict resolution is specified and the algorithm decides to serialize the transaction against a snapshot which has identical values of x, y and z, but also has an additional item a which also satisfies P. This item would never get reflected in the client transaction's readset, which is a clear violation of **A3** and hence **P3**. (This is because reconciliation can be considered as a second instance of $r[P]$. Furthermore, if a transaction is serialized against a snapshot, it implies that the transaction should have executed as if it had been reading data from the snapshot in the first place.) A solution to this problem is to provide read predicates rather than readsets to the algorithm. Another possibility is for the client to build the predicates into the conflict resolution functions.

Since the client transaction always gets reconciled against a snapshot of the database, anomaly **A5A** cannot occur. Note that as long as a client transaction reads data (or rather appears to read data) from a single snapshot, it can not be reading data from an intermediate update. Also note that uncommitted updates do not appear as a part of the database's state. This also precludes phenomenon **P0** since the client transaction will only "overwrite" committed writes (i.e. its writes will be serialized after committed writes).

In order to disallow **P4**, client updates must neither overwrite nor be overwritten by committed updates. Another way of looking at this requirement, is that the set of consistent snapshots on the server must be monotonic, i.e. a reconciling transaction must add snapshots to this set, but may

[3] Note that *NEWWRITESET*^{v} is a value snapshot while $S^{\uparrow}(1-\Delta)$ is a version snapshot. In $\{\langle x, 3\rangle\}$ the 3 is the value of x, but in $\{\langle x, 1\rangle\}$, 1 refers to the version of x.

not delete any other snapshot. Furthermore, every committing client transaction must introduce exactly one additional snapshot in the database. Note that in our case the snapshots are tied to versions (see the definition of $S(v)$).

To avoid **P4**, new writes to a data item can be serialized either before a blind write, where no read to the data item has taken place; or after the last write to the data item in the current database state (i.e. the current or latest snapshot). Since our algorithm always serializes new writes either against the latest (current) snapshot or before a blind write, **P4** can not occur. The former condition is precisely the reason why we need a separate *WSET*^{v} to pinpoint the blind writes.

The system obviously does not provide serializability since it allows **A3**. It also allows **A5B**. To see this consider the following example:

Server:$w_0[x_0] = 1, w_0[y_0] = 1, c_0, r_1[x_0] = 1, r_1[y_0] = 1, w_1[y_1] = 2, c_1$
Client:downloaded $x_0 = 1, y_0 = 1$: $r_{1'}[x] = 1, r_{1'}[y] = 1, w_{1'}[x] = 3, c_{1'}$

On reconciliation of the client transaction $1'$ the algorithm yields the following history:

Server:$w_0[x_0] = 1, w_0[y_0] = 1, c_0, r_{1'}[x] = 1, r_{1'}[y] = 1, r_1[x_0] = 1, r_1[y_0] = 1, w_{1'}[x] = 3, c_{1'}, w_1[y_1] = 2, c_1$

Note that the snapshot $S^{\downarrow}(1-\Delta) = \{\langle x, 0\rangle, \langle y, 0\rangle\}$ and *value*$(S^{\downarrow}(1-\Delta)) = \{\langle x, 1\rangle, \langle y, 1\rangle\}$ which is identical to the value snapshot seen by client transaction $1'$ giving a cost of 0. Furthermore, $S^{\uparrow}(1-\Delta) = \{\langle y, 2\rangle\}$. For the client transaction $1'$, *NEWWRITESET*^{v} = *WRITESET*$^v(T_{1'}) = \{\langle x, 3\rangle\}$ which does not "conflict" (i.e. have common data items) with $S^{\uparrow}(1-\Delta)$. Thus, transaction $1'$ is serialized with timestamp $1-\Delta$, which means it is serialized after the reads of transaction 1, but before its write.

It is easy to see that this history satisfies **A5B**. Thus, we conclude that our algorithm provides a weakened form snapshot isolation, i.e. snapshot isolation with phenomenon **A3**.

3.2. Providing serializability

As mentioned in the last section algorithm 1 does not guarantee serializability. We do not believe that this would be a problem for most systems. However, there might be some systems where serializability is desired or necessary. In this section, we show how to strengthen the algorithm to provide serializability. Note that to guarantee serializability on the system as a whole the server must have a multiversion optimistic concurrency control model that guarantees serializability for server transactions.

The key problems with providing serializability are avoiding phenomenon **P3** and **A5B**. As mentioned earlier both **A3** and **P3** can both be avoided by using read predicates to extract the read set from the snapshot, rather than using an absolute read set. The extracted set can then be tested against the actual read set of the transaction. The definition of C_{T_c} and CR_{T_c} functions must be extended to allow for the read predicates of the transaction. The default conflict resolution and cost functions then become

$$C_{T_c}(\textit{READSET}^v(T_c), \textit{WRITESET}^v(T_c), S^v_{in}) = \begin{cases} 0 & \text{if } \textit{READSET}^v(T_c) = P_{T_c}(\textit{value}(S^v_{in})) \\ \infty & \text{otherwise} \end{cases}$$

and

$$CR_{T_c}(\textit{READSET}^v(T_c), \textit{WRITESET}^v(T_c), S^v_{in}) = \begin{cases} \textit{WRITESET}^v(T_c) & \text{if } \textit{READSET}^v(T_c) \\ & = P_{T_c}(\textit{value}(S^v_{in})) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

where P_{T_c} is the read predicate of transaction T_c and $P(S)$ extracts from the set S a set of all elements $e \in S$ which satisfy P. Thus, $P(S) = \{e \mid e \in S \wedge P(e)\}$.

We believe that handling **P3** is a primarily the function of conflict resolution routines. Whenever a client wants to explicitly prevent **P3**, it simply needs to modify the functions it transmits with the client transactions. Thus, normally the server should not even care about read predicates.

To eliminate **A5B**, we note that the only reason histories satisfying **A5B** are allowed by algorithm 1 is that only the elements in the writeset of client transactions are inspected for conflicts against the reverse snapshot $S^{\uparrow}$. Instead, if both the read and writesets are inspected, then the resulting histories do not allow **A5B**. Thus, the for loop in line 10 of algorithm 1 must be changed to read:

for all $\langle x, i\rangle \in$ *NEWWRITESET*$^v \cup$ *READSET*$^v(T_c)$ **do**
  ⋮
end for

This modification can increase the collision cross section of the client transaction on any snapshot. This is due to the fact that there are additional elements to test in line 14 of algorithm 1, which means that there are additional chances of failure. Note that though most client transactions may have small single element writesets, they can still have relatively larger readsets.

4. Some practical considerations

4.1 Timestamp Maintenance

Algorithm 1 requires that its "input" transaction T_c be serialized with some timestamp $opt - \frac{\Delta}{iter}$. This implies that a new timestamp that is less than opt but greater than $lub\{v \mid v \in V \wedge v < opt\}$ must be given to the reconciling client transaction. In practice, though, we need to have integral values for timestamps. The most direct but expensive solution to this problem is to "renumber" the timestamps

each time a client transaction commits. An alternative is to use commit lists as described in [9]. Instead of storing version numbers with data items, we would store transaction ids and version numbers would correspond to the transaction's index in the commit list. Whenever a new transaction commits with "timestamp" $opt - \frac{\Delta}{k}$ it is inserted into the commit list just before the transaction with index opt. Another way of achieving integral timestamps is to leave "gaps" in the sequence of timestamps provided by the server, e.g. by splitting the timestamp into a most significant for committing server transactions and a least significant part for committing client transactions.

4.2. Increasing parallelism

The algorithm as described in the section 3 has one problem. It requires that the reconciliation of each client transaction be an atomic operation. Unfortunately, such reconciliation is likely to be a time consuming process since multiple snapshots must be computed and tested. Thus, the system throughput as a whole is likely to suffer during reconciliation due to loss of parallelism. We present a preliminary technique to improve parallelism in the system.

We can do this by exploiting one key property of the algorithm and our model of the database. The active transactions in our model always commit against the current snapshot. (Note that though these transactions might be reading data from an older snapshot, they must always be serialized after the current snapshot, which means that they must be serialized after the transaction that wrote the current snapshot.) Furthermore, the algorithm always scans the V in increasing order. Thus, a tree locking protocol with access intention locks on the elements of V can be used to improve parallelism in the reconciliation process. As long as the last element of V is not locked, server transactions can continue executing, since in our model, they must always commit against the latest snapshot (i.e. after the last committed transaction).

5. Conclusions

We have presented an algorithm that provides multiversion reconciliation. The algorithm is unique in that conflict resolution and detection are integrated with global serializability testing. We have also presented some practical considerations for implementation of the algorithm.

A key concept in our algorithm is that conflict resolution and detection are decoupled. The responsibility for detecting conflicts lies with the server. This is done by performing serializability testing on locally committed transactions on reconciling clients. On the other hand, conflict resolution is the responsibility of the client. The client manages this by providing the conflict resolution and cost functions for each transaction. In the absence of these functions, the server assumes a default, which guarantees snapshot isolation to "unmodified" client transactions.

References

[1] D. Agrawal and V. Krishnamurthy. Using multiversion data for non-interfering execution of write-only transactions. *Proceeding of the ACM SIGMOD conference*, pages 98–107, 1991.

[2] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. *Proceedings of ACM SIGMOD Conference*, pages 408–417, 1989.

[3] R. Alonso and H. F. Korth. Database system issues in nomadic computing. *Proceedings of the ACM SIGMOD*, pages 388–392, June 1993.

[4] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. *Proceedings of ACM SIGMOD Conference*, pages 1–10, 1995.

[5] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[6] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM TODS*, 9(3):456–481, Sept. 1984.

[7] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The BAYOU architecture: Support for data sharing among mobile users. *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–7, Dec. 1994.

[8] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. *Proceedings of ACM SIGMOD*, pages 173–182, June 1996.

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

[10] T. Imieliński and B. R. Badrinath. Mobile wireless computing: Challenges in data management. *Communications of the ACM*, 37(10):18–28, 1994.

[11] R. Katz and S. Weiss. Design transaction management. *Proceedings of the 21st Design Automation Conference*, pages 692–693, 1984.

[12] N. Krishnakumar and R. Jain. Mobile support for sales and inventory applications.

[13] G. Kuenning, G. J. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. *Proceedings of the USENIX Summer Conference*, pages 291–303, 1994.

[14] H. T. Kung and J. T. Robinson. On optimistic methods of concurrency control. *ACM TODS*, 6(2):213–226, June 1981.

[15] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, pages 447–459, Sept. 1989.

[16] G. Walborn and P. Chrysanthis. Supporting semantics-based transaction processing in mobile database systems. *Proceedings of the 14th Symposium on Reliable Database Systems*, Sept. 1995.