Incorporating Transaction Semantics to Reduce Reprocessing Overhead in Replicated Mobile Data Applications [*]

Peng Liu Paul Ammann Sushil Jajodia

Center for Secure Information Systems
George Mason University
{*pliu,pammann,jajodia*}*@isse.gmu.edu*

Abstract

Update anywhere-anytime-anyway transactional replication has unstable behavior as the workload scales up. To reduce this problem, a two-tier replication algorithm is proposed in [GHOS96] that allows mobile applications to propose tentative transactions that are later applied to a master copy. However, it can suffer from heavy reprocessing overhead in many circumstances. In this paper, we present the method of merging histories instead of reprocessing to reduce the overhead of two-tier replication. The basic idea is when a mobile node connects to the base nodes merging the tentative history into the base history so that substantial work of tentative transactions could be saved. As a result, a set of undesirable transactions (denoted $\mathbf{B}$*) have to be backed out to resolve the conflicts between the two histories. Desirable transactions that are affected, directly or indirectly, by the transactions in* $\mathbf{B}$ *complicate the process of backing out* $\mathbf{B}$*. We present a family of novel rewriting algorithms for the purpose of backing out* $\mathbf{B}$*. By incorporating transaction semantics, our rewriting methods are strictly better at saving desirable tentative transactions than the traditional reads-from transitive-closure based approach. And in most cases our rewriting methods are better at saving desirable tentative transactions than an approach which is based only on commutativity.*

Key Words: Mobile Databases, Data Replication, Transaction Processing.

[*]This effort was sponsored by Rome Laboratory, Air Force Material Command, USAF, under agreement number F30602-97-1-0139. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation herein. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Rome Laboratory or the U.S. Government.

1 Introduction

Data is replicated at multiple network nodes for performance and availability. There are typically two ways to propagate updates to replicas: *eager replication* keeps all replicas exactly synchronized at all nodes by updating all the replicas as part of one atomic transaction; *lazy replication* asynchronously propagate replica updates to other nodes after the updating transaction. Moreover, there are typically two ways to regulate replica updates: *group replication* where any node with a copy of a data item can update it (this is often called *update anywhere*), and *master replication* where each object has a master node; only the master can update the *primary copy* of the object; all other replicas are read-only; other nodes wanting to update the object request the master do the update.

It is shown in [GHOS96] that update anywhere-anytime-anyway transactional replication has unstable behavior as the workload scales up: (1) A ten-fold increase in nodes and traffic gives a thousand fold increase in deadlocks or reconciliations. (2) Disconnected operation and message delays mean lazy replication has more frequent reconciliation.

Since eager replication is not an option for mobile applications where most nodes are normally disconnected, the unstable behavior of replicated mobile applications can be more serious. To solve this problem, a modified lazy-master replication scheme, namely, *two-tier replication*, is proposed in [GHOS96]. Assume there are two kinds of nodes in a mobile database where each data item is replicated on every node: *mobile nodes* are disconnected much of the time. *Base nodes* are always connected. A mobile node may be the master of some data items. Most items are mastered at base nodes. The basic idea of *two-tier replication* is first allowing users to run tentative transactions on a mobile node, later when the mobile node connects to the base nodes, these tentative transactions will be transformed to corresponding base transactions and then reexecuted. Failed reexecutions will be informed to the users together with the corresponding reasons.

Although *two-tier replication* can achieve the goals such as *availability*, *scalability*, *mobility*, *serializability*, and *convergence* [GHOS96], it can suffer from heavy reprocessing overhead in many circumstances. For example, when the number of mobile nodes are much larger than that of base nodes, even if the transaction processing on each mobile node is not busy, the reprocessing on the base nodes can be very busy since the number of the accumulated tentative transactions waiting for reexecuting at the base nodes can be huge.

In this paper, we present the method of merging histories instead of reprocessing to reduce the overhead of two-tier replication. The basic idea is when a mobile node connects to the base nodes merging the tentative history into the base history so that substantial work of tentative transactions could be saved. Since the two histories may conflict with each other, we build the *precedence graph* which is proposed in [Dav84] to detect the inconsistency and to compute the set of undesirable transactions (denoted $\mathbf{B}$) whose backing out can resolve the conflicts. Desirable transactions that are affected, directly or indirectly, by the transactions in $\mathbf{B}$ complicate the process of backing out $\mathbf{B}$. We present a family of novel rewriting algorithms for the purpose of backing out $\mathbf{B}$. By incorporating transaction semantics, our rewriting methods are strictly better at saving desirable tentative transactions than the back-out method used in [Dav84][‡]. The approach of merging histories still keeps the above desirable features of two-tier replication. And it will not violate the durability property of base transactions.

The paper is organized as follows. In Section 2, we present the merging protocol. We give our model for rewriting histories in Section 3. In Section 4 and 5, we give our rewriting algorithms. In Section 6, we show how to prune a rewritten history so that a repaired history can be generated. And we conclude in Section 7.

2 The Proposed Replication Scheme

2.1 The Protocol

In two-tier replication, there are two kinds of transactions: *base transactions* work only on master data since lazy master replication where reads go to the master gives ACID serializability, and they produce new master data. They involve at most one connected-mobile node and may involve several base nodes. *Tentative transactions* work on local tentative data. They produce new tentative versions.

[‡]This implys that our rewriting methods can also be used to improve the performance of optimistic replication protocols in distributed database systems.

In our merging protocol, there are two kinds of serializable histories: (1) **Base History:** A base history is an execution history of base transactions that read and write only master data. (2) **Tentative History:** A tentative history is an execution history of tentative transactions originated and executed on a mobile node.

The merging protocol works as follows:

1. When a mobile node connects to the base nodes, we construct a precedence graph similar to [Dav84], denoted $G(H_m, H_b)$, using both the tentative history (denoted H_m) and the base history (denoted H_b) as follows [§]:
   - Let T_i and T_j be two tentative transactions that perform conflicting operations on a data item [¶]. There is a directed edge $T_i \rightarrow T_j$ if T_i precedes T_j.
   - Let T_i and T_j be two base transactions that perform conflicting operations on a data item. There is a directed edge $T_i \rightarrow T_j$ if T_i precedes T_j.
   - Let T_m be a tentative transaction, T_b be a base transaction. If T_m read a data item that has been updated by T_b, then there is a directed edge $T_m \rightarrow T_b$. If T_b read a data item that has been updated by T_m, then there is a directed edge $T_b \rightarrow T_m$.
2. Detect if the precedence graph has any cycles. If so, compute the set of undesirable transactions $\mathbf{B}$ whose backing out can break these cycles. In order to ensure the durability of base transactions, only tentative transactions can be put into $\mathbf{B}$. Several back-out strategies for getting $\mathbf{B}$ are proposed in [Dav84].
3. Rewrite H_m so that transactions in $\mathbf{B}$ are moved to the end of H_m, and a prefix of the rewritten history is composed of only desirable transactions (the prefix is referred to as the *repaired history*). The rewriting model and algorithms are presented in Section 3, 4, and 5.
4. Prune the rewritten history using the methods proposed in Section 6 so that the repaired history can be extracted.
5. Forward updates from tentative transactions in the repaired history to the base nodes. Note that for each data item d modified by a transaction in the repaired history, we only need the value of d in the final state of the repaired history to merge H_m and H_b.

[§]H_m and H_b must start with the same database state because otherwise the correctness of the merger can not be ensured.

[¶]Two operations *conflict* if one is write.

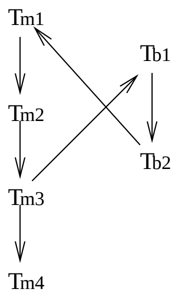

Figure 1. The Precedence Graph for the History in Example 1

6. Try to reexecute each backed-out tentative transaction. Failed reexecutions will be informed to the users together with the corresponding reasons.

It should be noticed that here we assume that the differences between the result of a tentative transaction in H_m and that in the merged history are acceptable. Handling the differences that are unacceptable is out of the scope of the paper.

Example 1 *Consider the six transactions given below:* $READSET(T_{m1}) = WRITESET(T_{m1}) = \{d_1, d_2\}$, $READSET(T_{m2}) = \{d_2, d_3\}$, $WRITESET(T_{m2}) = \{d_3\}$, $READSET(T_{m3}) = \{d_3, d_4, d_5, d_6\}$, $WRITESET(T_{m3}) = \{d_4, d_6\}$, $READSET(T_{m4}) = WRITESET(T_{m4}) = \{d_6\}$, $READSET(T_{b1}) = WRITESET(T_{b1}) = \{d_5\}$, $READSET(T_{b2}) = \{d_1, d_5\}$, $WRITESET(T_{b2}) = \{\}$.

Assume $H_m = T_{m1}\ T_{m2}\ T_{m3}\ T_{m4}$, $H_b = T_{b1}\ T_{b2}$. The precedence graph G is shown in Figure 1. Since the graph has a cycle, conflict exists among the transactions. Note that since T_{b2} read the item d_1 which is then updated by T_{m1}, T_{b2} should precede T_{m1}; since T_{m1} should precede T_{m3}, T_{b2} should precede T_{m3}; however, since T_{m3} read the item d_5 which is then updated by T_{b1}, T_{m3} should precede T_{b1}, since T_{b1} should precede T_{b2}, T_{m3} should precede T_{b2}, yielding a contradiction.

However, after T_{m3} and T_{m4} are backed out, it is clear that the reconstructed precedence graph is acyclic. And clearly the history $H = T_{b1}\ T_{b2}\ T_{m1}\ T_{m2}$ is an equivalent merged history which can generate the same database state as the state generated by forwarding the updates of T_{m1} and T_{m2} to the base nodes.

The correctness of the precedence-graph based approach is shown in the following theorem, which is similar to Theorem 2.2.2 in [Dav84].

Theorem 1 Given H_m and H_b, the precedence graph $G(H_m, H_b)$ is acyclic if and only if H_m and H_b are serializable (i.e., equivalent to some merged history H).

Cycles in $G(H_m, H_b)$ indicate that there are conflicts between H_m and H_b. In order to resolve these conflicts, we have to back out enough tentative transactions. According to [Dav84], there are two kinds of tentative transactions that have to be backed out: *undesirable* transactions are the set of transactions whose absence from the precedence graph can break all the cycles (Note that the set is $\mathbf{B}$). For example, in Example 1 $\mathbf{B} = \{T_{m3}\}$. *Affected* transactions are the set of transactions that are in $\mathbf{B}$'s transitive-closure of the *reads-from* relation [‖]. For example, in Example 1 T_{m4} is an affected transaction since it reads d_6 from T_{m4}, thus is in the reads-from transitive-closure.

Minimizing the number of transactions in $\mathbf{B}$ is NP-complete although in our protocol only tentative transactions can be backed out. The reason is similar to [Dav84]. Although this result discourages attempts to minimize the total back-out cost, the simulation results of [Dav84] show that in many situations several back-out strategies, in particular *breaking two-cycles optimally*, can still achieve good performance. Interested readers can refer to [Dav84] for more details on these back-out strategies.

After $\mathbf{B}$ is computed, [Dav84] backs out every affected transaction. This approach, however, can still result in a big set of affected transactions in many cases. For example, in Example 1 if T_{m4} and T_{m3} commute, then the work of T_{m4} can be saved by compensating T_{m3} at the end of H_m. Our goal is to save the work of affected transactions as much as possible.

The rewriting approach, enforced in step 3 and step 4 and proposed in Section 4, 5 and 6, has the following desirable features which can be exploited to save more transactions, ensure the correctness of the merging protocol, and achieve better performance. For lack of space, the proofs of all the Lemmas and Theorems about the rewriting approach except Theorem 4 are omitted. Interested readers can refer to [LAJ99] for the proofs.

- It can save every affected transaction.
- In most situations, it can save more transactions than the approaches which are based only on commutativity.
- Every rewritten history is equivalent to the original history in that they always generate the same final state.
- Every repaired history generated by the rewriting approach is consistent.

[‖]A transaction T_i reads x from T_j if T_j reads x after T_i has updated x and there are no transactions that update x between the time T_i updates x and T_j reads x.

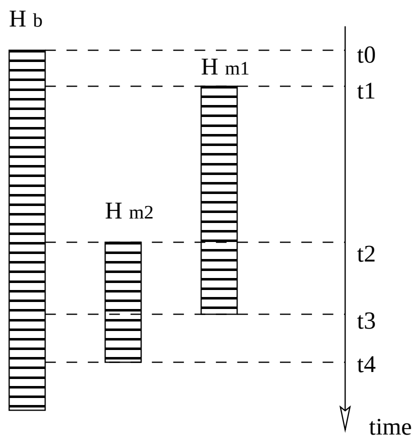

Figure 2. The Relation Between Multiple Tentative Histories

- After a rewritten history is pruned, we can always get the same state as that generated by re-executing the repaired history.

2.2 Synchronizing Multiple Tentative Histories

The synchronization among multiple tentative histories when there are several mobile nodes is a critical issue. Assume there are n active tentative histories at time t, namely, H_{m1}, ..., H_{mn}. These histories might start to be merged at different points of time, for example (See Figure 2), the base history H_b began at time t_0, tentative history H_{m1} began and ended at time t_1 and t_3 respectively, tentative history H_{m2} began and ended at time t_2 and t_4 respectively, both H_{m1} and H_{m2} were active during the period of time from t_2 to t_3.

Two possible isolation strategies may be applied here. Strategy 1 is to let tentative histories, i.e., H_{m1} and H_{m2}, take different database states as their original states. For example, H_{m1} takes the database state at time t_1, and H_{m2} takes that at time t_2. Since the value of some data item x may change during the period of time from t_1 to t_2, different values of x can be given to H_{m1} and H_{m2}.

Strategy 1 can cause serious problems. In Figure 2, suppose data item x was updated by H_{m1} and read by H_{m2}, suppose further that x had not been updated by H_b from t_1 to t_2, then after H_{m1} was merged into H_g at time t_3, the transaction in H_{m1} which updated x could be merged into H_b in a position between $p(t_1)$ and $p(t_2)$ ($p(t)$ denotes the position of the latest committed transaction in H_b at time t). In this situation, the value of x in the database state at $p(t_2)$ would be changed, and the new value of x should have been given to H_{m2} when it read x. Since H_{m2} might read x before the merge, when H_{m2} is going to be merged into H_b, we may fail to find a subhistory of H_b into which can H_{m2} be merged.

Strategy 2 is to let every tentative history take the same base database state as its original database state. Strategy 2 avoids the above problem and allows merging multiple tentative histories individually into the base history, because no matter how other mergers will take place, a tentative history H_{mi} can always find a proper base history to merge into, that is, the base sub-history which begins with the same database state as H_{mi}. This is also the reason why we took this strategy in our merging protocol. However, we need to reset the original state periodically for all the tentative histories because otherwise the back-out cost of mergers will increase substantially as the base history grows longer and longer. The resynchronization can work as follows: assume there is a time window within which each mobile node will connect to the base nodes at least once, then we can reset the original state at the beginning of each time window. As a result, all of the mobile nodes that connect to the base nodes in a time window will take the same state, i.e., the state of the base history at the beginning of the time window, as the original states for their new tentative histories after the mergers in the time window. In addition, when a mobile node connects to the base nodes too late, i.e., the next time window is already open, its tentative history will not be merged. Instead, its transactions will be reexecuted.

3 The Rewriting Model

Given a tentative history H to be merged, after $\mathbf{B} = \{B_{i1}, B_{i2}, ..., B_{im}\}$ is computed in step 2, we denote other transactions in H as $\mathbf{G} = \{G_{j1}, G_{j2}, ..., G_{jn}\}$. We denote the set of affected transactions as $\mathbf{AG}$. We assume H is serializable and there is an explicit serial history H^s of H. We assume that transactions do not issue blind writes. That is, if a transaction writes some data, the transaction is assumed to read the value first. Although the rewriting approach can be adapted to blind writes, doing so complicates the presentation.

For a serial tentative history H^s, we *augment* H^s with explicit database states so that the result is a sequence of interleaved transactions and database states. The sequence begins and ends with a state. The state that immediately precedes a transaction in H^s is called the *before state*; the state that immediately follows a transaction in H^s is called the *after state*. For an example, consider the augmented history $H^s_1 = s_0\ B_1\ s_1\ G_2\ s_2$ where

B_1 : `if x > 0 then y := y + z + 3`
G_2 : `x := x - 1`

The states associated with H_1^s are:

$$s_0 = \{\texttt{x = 1; y = 7; z = 2}\}$$
$$s_1 = \{\texttt{x = 1; y = 12; z = 2}\}$$
$$s_2 = \{\texttt{x = 0; y = 12; z = 2}\}$$

In rewriting histories, the general goal is either to move transactions in $\mathbf{B}$ towards the end of a history or to move transactions in $\mathbf{G}$ towards the beginning of a history. It turns out that rewriting histories for recovery purposes requires some care with respect to state equivalence of histories. Two augmented histories H_1^s and H_2^s are *final state equivalent* if they are over the same set of transactions and the final states are identical. Note that two final state equivalent histories might not be *conflict equivalent*, or *view equivalent* [BHG87].

The example above helps to clarify this point. After we make the transformation of exchanging the order of G_2 and B_1, H_1^s is clearly not final state equivalent to the serial history G_2B_1 since they result in different final states. At this situation, if H_1^s has more transactions following B_1G_2, i.e., $G_3G_4...G_n$, then this transformation will change the before state of G_3. As a result, after the transformation, the rewritten history may not be consistent any longer because the precondition of some G_i, $3 \leq i \leq n$, may not be satisfied any more. Even if the rewritten history is still consistent, the rewritten history usually can not result in the same final state, and the new final state is usually very difficult to get, thus semantics-based compensation is disabled. Therefore, keeping the final state equivalence of rewritten histories during a rewrite is essential to the success of the rewrite.

We approach this problem by decorating each transaction T in an augmented history H^s with special values for read purposes by T. The decoration is facilitated by the notation *fix* which is specified below.

Definition 1 A *fix* for transaction T_i in history H^s, denoted F_i, is a set of variables read by T given values as in the original position of T in H^s. That is, $F_i = \{(x_1, v_1), ..., (x_n, v_n)\}$, and v_i is what T_i read for x_i in the original history.

The notation ${T_i}^{F_i}$ indicates that the values read by T_i for variables in F_i should not come from the before state of T_i, but from F_i.

To reduce notational clutter, we show just the variable names in F_i and omit the associated values.

Consider the augmented history $H_1^s = s_0\ B_1\ s_1\ G_2\ s_2$ above. As discussed, the history $H_2^s = s_0\ G_2\ s_3\ B_1\ s_3$ with $s_3 = \{\texttt{x = 0; y = 7; z = 2}\}$ results in a different value of y in the final state, but the history $H_3^s = s_0\ G_2\ s_3\ B_1^{F_1}\ s_2$ ends in final state s_2 for $F_1 = \{x\}$. States s_1 and s_3 differ in the value of x; this discrepancy is captured by F_1, where x is associated with the value 1, which is the value B_1 read for x in the original history H_1^s.

In what follows, each transaction T_i is assumed to have an associated fix F_i. For ordinary serializable execution histories, each such fix $F_i = \emptyset$, the empty fix. In the example above, the two histories

$$H_1^s = s_0\ B_1^{\emptyset}\ s_1\ G_2^{\emptyset}\ s_2$$
$$H_3^s = s_0\ G_2^{\emptyset}\ s_3\ B_1^{\{x\}}\ s_2$$

are final state equivalent.

Definition 2 Given a history H^s over $\mathbf{B} \cup \mathbf{G}$, H_r^s is a *repaired* history of H^s if (1) H_r^s is over some subset of $\mathbf{G}$, and (2) There exists some history H_e^s over $\mathbf{B} \cup \mathbf{G}$ such that (a) H_r^s is a prefix of H_e^s and (b) H_e^s and H^s are final state equivalent.

Our notion of a repaired history is that only desirable transactions remain (condition 1) and further that there is some extension to the repair that captures exactly the same transformation to the database state as the original history (condition 2). Note that the reads-from transitive-closure based approach satisfies the first part of the definition of a repaired history where the subset of $\mathbf{G}$ is $\mathbf{G} - \mathbf{AG}$.

Armed with a definition of repairs to histories, we are now ready to consider algorithms to construct them.

4 Can-follow Rewriting

We denote the set of items read or written by a transaction T as $T.readset$ or $T.writeset$, and the set of items read or written by a sequence of transactions $R = T_1T_2...T_n$ as $R.readset$ or $R.writeset$. Due to our assumption of no blind writes, $R.writeset \subseteq R.readset$.

Definition 3 Transaction T *can follow* a sequence of transactions R if $T.writeset \cap R.readset = \emptyset$.

There are some properties of can follow: (1)If $T_i.writeset$ is not empty, then transaction T_i can not follow itself. (2)The fact that T_i can follow transaction T_j and T_j can follow transaction T_k does not imply that T_i can follow T_k. (3) Read-only transactions can follow any transaction. (4) Transaction T can follow a sequence of transactions R iff T can follow every transaction in R.

The can follow relation captures the notion that a transaction T can be moved to the right past a sequence of transactions R if no transaction in R reads from T.

The can follow relation can be used to rewrite a history to move transactions in $\mathbf{G} - \mathbf{AG}$ to the beginning of the history, namely, move transactions in $\mathbf{B} \cup \mathbf{AG}$ backwards.

```
**Algorithm 1** Can-Follow Rewriting
**Input:** the serial history $H^s$ to be rewritten and the set $\mathbf{B}$ of bad transactions[**].
**Output:** a rewritten history with transactions in $\mathbf{G} - \mathbf{AG}$ preceding transactions in $\mathbf{B} \cup \mathbf{AG}$.
**Method:** Scan forward from the first good transaction after $B_1$ until the end of $H^s$, for each transaction $T$
    case $T \in \mathbf{B}$   skip it;
    case $T \in \mathbf{G}$
        if each transaction between $B_1$ and $T$ (including $B_1$) can follow $T$, then move $T$ to the position immediately preceding $B_1$.
```

Algorithm 1 does not describe how to compute the *fix* with any transaction which has some transaction being moved to the left of it. The reason is that repair can simply be accomplished by undo. However, if we want to save some of the transactions in **AG** then we need to maintain the *fix* information for these transactions. Fixes are computed as follows:

Lemma 1 Suppose transaction T can follow sequence R in history $H_1^s = s_0\ T^{F_1}\ s_1\ R\ s_2$ Then for fix $F_2 = F_1 \cup (T.readset \cap R.writeset)$ history $H_2^s = s_0\ R\ s_3\ T^{F_2}\ s_2$ is final state equivalent to H_1^s. The values associated with each data item in the fixes are those originally read by T.

The correctness of Algorithm 1 is specified as follows.

Theorem 2 Given a history H^s, Algorithm 1 produces a history H_e^s with a prefix H_r^s such that:

1. All and only transactions in $\mathbf{G} - \mathbf{AG}$ appear in H_r^s.
2. H_e^s and H^s order transactions in $\mathbf{G} - \mathbf{AG}$ identically. And they order transactions in $\mathbf{B} \cup \mathbf{AG}$ identically.
3. The fix associated with each transaction in H_r^s is empty.
4. H^s and H_e^s are final state equivalent. And H_r^s is a repaired history of H^s.

In realistic applications, although Lemma 1 gives users a sound approach to capture fixes in Algorithm 1, it is not efficient in many cases since whenever a transaction T_i is moved to the left of another transaction T_j, F_j may need be augmented. A better way to compute fixes is as follows:

Lemma 2 For any history H^s, assume rewriting H^s using Algorithm 1 generates a history H_e^s with a prefix H_r^s (H_e^s typically looks like: $G_{j1} ... G_{jn}\ B_{i1}^{F_{i1}}\ AG_{k1}^{F_{k1}} ... B_{im}^{F_{im}} ... AG_{kp}^{F_{kp}}$. The subhistory before $B_{i1}^{F_{i1}}$ is H_r^s), and assume all the fixes are computed according to lemma 1 during the rewriting, then the history $H_e^{s\prime}$, generated by replacing each non-empty fix F_i in H_e^s with $F_i' = T_i.readset - T_i.writeset$, is final state equivalent to H_e^s.

[**] In the rest of the paper, we use B_1 to denote the first bad transaction in H^s

According to Lemma 2, there can be two methods to get $T_i.readset - T_i.writeset$ for a transaction T_i: one is to first get the readset and writeset of T_i after a history is generated using the approaches proposed in Section 7, then compute $T_i.readset - T_i.writeset$; the other is to let each transaction T_i write the set $T_i.readset - T_i.writeset$ as a record to the database when it is executed, then when we rewrite H^s all the fixes can be directly got from the database.

The major result of this section is an equivalence theorem between the effect of a reads-from transitive-closure based algorithm and the history produced by Algorithm 1. The reads-from transitive-closure based algorithm restores the values of all elements updated by transactions in $\mathbf{B} \cup \mathbf{AG}$. In particular, the theorem shows that the optimizations in the following section are strict improvements over the reads-from transitive-closure based algorithm.

Theorem 3 Given H^s, let H_r^s be the serial history produced by eliminating all transactions in $\mathbf{B} \cup \mathbf{AG}$ as in the dependency-graph based algorithm. Given H^s, let H_e^s be the result of Algorithm 1. Then H_r^s is a prefix of H_e^s.

5 Saving Additional Desirable Transactions

In this section, we show how to integrate the notion of commutativity with Algorithm 1 to save not only the transactions in $\mathbf{G} - \mathbf{AG}$, but potentially transactions in **AG** as well.

5.1 Motivating Example

Consider the following history:

$H_4 : B_1 G_2 G_3$
B_1: if $u > 10$ then $x := x + 100, y := y - 20$
G_2: $u := u - 20$
G_3: $x := x + 10, z := z + 30$

The result of Algorithm 1 is the history $H_e^s = G_2 B_1^{\{u\}} G_3$, thus G_3 need to be undone. Note that G_3 *commutes backward through* $B_1^{\{u\}}$ for any value of u[††], and so a final

[††] We adapt the notation of commutativity from [LMWF94, Wei88]. Transaction T_2 *commutes backward through* transaction T_1 if for any state s on which $T_1 T_2$ is defined, $T_2(T_1(s)) = T_1(T_2(s))$; T_1 and T_2 *commute* if each commutes backward through the other. Note that one-sided commutativity (i.e., commutes backward through) is enough for our purpose.

state equivalent history is $G_2G_3B_1^{\{u\}}$. Compensation for $B_1^{\{u\}}$ can be applied directly to this history, but an undo approach requires more care. Suppose we decide to undo B_1 by restoring the before values for x and y from the log entries for B. After B is undone the value of u is unchanged because only G_1 updates u. The value of z is unchanged because only G_3 updates z. The effect of G_3 on x is wiped out because both G_3 and B update x, and after B is undone x no longer reflects the effects of G_3. However x can be repaired by re-executing the corresponding part of G_3's code, that is, $x = x + 10$, and the cumulative effect is that of history G_2G_3. We call this last step an *undo-repair* action. Both the undo approach and the compensation approach to repair are discussed in detail in section 6.

The presence of fixes for transactions limits the extent to which commutativity can be applied. Consider the following history:

$H_5 : \; s_0 \; T_1 \; s_1 \; T_2 \; s_2 \; T_3 \; s_3$
T_1: if $y > 200$ then $x := x + 100$ else $x := x * 2$
T_2: $y := y + 100$
T_3: if $y > 200$ then $x := x - 10$ else $x := x/2$

T_1 can follow T_2 with fix $F_1 = \{y\}$ for T_1. Although T_3 commutes backward through T_1, T_3 does not commute backward through $T_1^{F_1}$, because the value of x produced by $T_1^{F_1}$ depends on the value of y in the fix F_1. For example, if the initial value of x is 100 and fix value of y is 150, then the final value of x in history $T_2T_1^{F_1}T_3$ is 190, but the final value of x in history $T_2T_3T_1^{F_1}$ is 180.

The example shows that sometimes a fix can interfere with the commutativity of transactions. This motivates our definition of *can precede*:

Definition 4 A transaction T_2 *can precede* a transaction T_1 for fix F if for any assignment of values to the variables in F and for any state $s_0 \in \mathbf{S}$ on which $T_1^FT_2$ is defined,

1. $T_2T_1^F$ is defined on s_0, and
2. The same final state is produced by $T_1^FT_2$ and $T_2T_1^F$.

Similar to commutativity [LMWF94, Wei88], can precede relation can be detected by analyzing the semantics of transaction profiles (or codes). For example, in H_4, G_3 can precede $B_1^{\{u\}}$ because the operations of G_3 and B_1 on data item x, respectively, commute no matter to which value u is assigned. For canned systems which are widely used in real applications such as banking systems and airline ticket reservation systems, since transactions are of limited number of types and the code of each transaction type is available, so the can precede relation between two transactions can be pre-detected by detecting the relation between the corresponding two transaction types in advance. For some non-canned systems where codes of transactions are recorded when they are executed, the can-precede relation can be detected at the time of repair. However, the cost of detection can be untoleratable if there are so many transactions in the history and few transactions are of the same type because automatic approaches are usually limited in doing this thus some amount of manual work is often required. For some non-canned systems where codes of transactions are not recorded, the can precede relation usually can not be detected, thus only can follow rewriting can be enforced.

5.2 Can-Follow and Can-Precede Rewriting

We present a repair algorithm which integrates both can follow and can precede. For brevity, only the modifications to Algorithm 1 are specified.

Algorithm 2 Can-Follow and Can-Precede Rewriting
Method:

```
    case $T \in \mathbf{G}$
        if for each transaction $T'$ between $B_1$ and $T$(including $B_1$),
either $T'$ can follow $T$ or $T$ can precede $T'$, then move $T$ to the position
immediately preceding $B_1$. As $T$ is pushed through each such $T'$ between
$B_1$ and $T$ to the left of $B_1$
            if $T'$ can follow $T$, then push $T$ to the left of $T'$ and
modulate the fix of $T'$ correspondingly according to Lemma 1;
            else push $T$ to the left of $T'$.
```

In Algorithm 1, Lemma 2 provides an efficient way to compute fixes. However, Lemma 2 may not hold for Algorithm 2 if the system does not have the following property.

Property 1 Transaction T_j can precede transaction T_i for a fix F_i only if $(T_i.readset - T_i.writeset - F_i) \cap T_j.writeset = \emptyset$ and $(T_j.readset - T_j.writeset) \cap T_i.writeset = \emptyset$.

It should be noticed that Property 1 is not strict since if Property 1 does not hold, $T_i^{F_i}T_j$ and $T_jT_i^{F_i}$ usually can not result in the same final state.

Lemma 3 Lemma 2 holds for Algorithm 2 if the system has Property 1.

Commutativity can be directly used to rewrite histories. The rewriting algorithm based on *commutes backward through* can be adapted from Algorithm 1 by replacing the word *can-follow* with the word *commutes backward through*. We found that in most cases Algorithm 2 can save more transactions than algorithms based on commutativity.

Theorem 4 Let FPR(H^s) and CBTR(H^s) represent the sets of saved transactions after H^s is rewritten by Algorithm 2 and the rewriting algorithm based on *commutes backward through*, respectively. If the system has Property 1, then $\forall$ H^s, CBTR(H^s) $\subseteq$ FPR(H^s).

The proof of Theorem 4 is in Appendix A.

6 Pruning Rewritten Histories

After a rewritten history H_e^s, is generated from H^s, we need to prune H_e^s such that the effect of H_r^s can be generated. If H_e^s is produced by Algorithm 1, then the pruning can be easily done by undoing each transaction in $H_e^s - H_r^s$. However, if H_e^s is produced by Algorithm 2, undo does not give the pruning in most cases.

In this section, two pruning approaches are presented: the compensation approach is more direct, but compensating transactions may not be specified in some systems. The undo approach is a syntactic approach, but it imposes some restrictions on transaction programs.

6.1 The Compensation Approach

We denote the compensating transaction of transaction T_i as T_i^{-1} [GM83, GMS87, KLS90]. T_i^{-1} semantically undoes the effect of T_i. It is reasonable to assume that $T_i^{-1}.writeset \subseteq T_i.writeset$, and for simplicity we further assume that every transaction T_i has a compensating transaction. T_i^{-1} is usually not enough to compensate $T_i^{F_i}$ where F_i is not empty. Fixes must be taken into account for the compensation to be correct.

Definition 5 The *fixed compensating transaction* of $T_i^{F_i}$, denoted $T_i^{(-1,F_i)}$, is the regular compensating transaction of T_i (denoted T_i^{-1}) associated with the same fix F_i.

The effects of $T_i^{F_i}$ can be removed by executing $T_i^{(-1,F_i)}$, this is justified by the following lemma.

Lemma 4 Transaction $T_i^{F_i}$ can be *fix compensated*, that is, for every consistent state s_1 on which $T_i^{F_i}$ is defined, $T_i^{(-1,F_i)}(T_i^{F_i}(s_1)) = s_1$, if $F_i \cap T_i.writeset = \emptyset$.

Lemma 4 shows that every H_e^s produced by Algorithm 2 can be fix compensated because for each transaction T_i in H_e^s which is associated with a non-empty fix F_i, $F_i \cap T_i.writeset = \emptyset$ always holds. The pruning algorithm by compensation therefore is straightforward: based on the final state of H^s, executing the fixed compensating transaction for each transaction in $H_e^s - H_r^s$ in the reverse order as they are in H^s.

6.2 The Undo Approach

Given a rewritten history H_e^s, the undo approach first undo all transactions in $H_e^s - H_r^s$, then execute the undo-repair actions for the transactions in both $\mathbf{AG}$ and H_r^s in the same order as they are in H_r^s.

Our algorithm described below is based on the following assumptions about transactions:

- a transaction is composed of a sequence of statements, each of which is either: (1) An operation, or (2) A conditional statement of the form: **if** c **then** $SS1$ **else** $SS2$, where $SS1$ and $SS2$ are sequences of statements, and c is a predicate;
- each statement can update at most one data item;
- each data item is updated only once in a transaction;

```
Algorithm 3 Build Undo-repair Actions
Input: an affected transaction $AG_k$.
Output: the undo-repair action $URA_k$ for $AG_k$.
Method:
1. Copy the codes of $AG_k$ to $URA_k$. Assign $URA_k$ with the same input
   parameters and the same values associated with them as $AG_k$.
2. Parse $URA_k$. For each statement to be scanned
      case it is a read statement, keep it;
      case it is an update statement of the form: $x := f(x, y_1, y_2, ...y_n)$ where $f$ specifies
         the function of the statement, $y_1, ..., y_n$ are the data items used in the statement.
         Some input parameters may also be used in the statement, but they are not explicitly stated here.
            if $x$ has not been updated by any other transaction in $\mathbf{B} \cup \mathbf{AG}$
               Remove the statement from $URA_k$;
            elseif $x$ has not been updated by any transaction in $\mathbf{B} \cup \mathbf{AG}$ which precedes $AG_k$ in $H^s$
               Replace the statement with: $x := AG_k.afterstate.x$,
               that is, get the value of $x$ from the after state of $AG_k$ in $H^s$;
            else for each $y_i$ (including $x$)
               if $y_i$ has not been updated by any preceding statement and has not been updated
               by any transaction in $\mathbf{B} \cup \mathbf{AG}$ which precedes $AG_k$ in $H^s$
                  Bind $y_i$ with $AG_k.beforestate.y$;
3. Reparse $URA_k$. Remove every read statement which reads some item never used in an update
   statement of $URA_k$, or reads some item $y$ used in one or more update statements but $y$ is
   bound with a value in these statements.
```

The correctness of the undo approach is specified as follows.

Theorem 5 For any rewritten history H_e^s generated by Algorithm 2, after all transactions in $H_e^s - H_r^s$ are undone, executing the undo-repair actions which are generated by Algorithm 3 for the affected transactions in H_r^s, in the same order as their corresponding affected transactions are in H_r^s, produces the same effect of H_r^s.

7 Discussion and Conclusion

7.1 Discussion

One important question we need to answer is: 'Is the merging protocol always more efficient than the original two-tier replication protocol? If not, in which situations will the merging protocol win?'

We break down the costs of either the merging protocol or the two-tier replication protocol into three parts: (1) the communication costs between mobile nodes and the base node, (2) the computing costs at the mobile codes, and (3) the computing costs at the base node. It is clear that the above question can be roughly answered by making a comparison between the total cost paid by the merging protocol to save a set of tentative transactions (denoted $\mathbf{SAV}$), and the total cost paid by the two-tier replication protocol to reprocess each transaction in $\mathbf{SAV}$.

Consider a scenario where there is only one mobile node (similar analysis can be applied to scenarios with multiple mobile nodes), in the two-tier replication protocol, we need to transmit the code and input arguments of each transaction in $\mathbf{SAV}$ to the base node for reprocessing[‡‡], and send the execution result back. In contrast, in the merging protocol we need to first transmit the readset and writeset of each transaction in the tentative history (denoted H_m), and the precedence graph of H_m (denoted $G(H_m)$) to the base node for building the precedence graph $G(H_m, H_b)$ (H_b denotes the base history), then send the set $\mathbf{B}$ back to the mobile node, then transmit the updates of $\mathbf{SAV}$ to the base node after the rewrite for merging. Therefore, we can see that if transaction codes need be transmitted by the two-tier replication protocol, then the communication costs of these two protocols are comparable if many transactions are saved, otherwise, the two-tier replication protocol usually costs less.

At the base node, the costs of the two-tier replication protocol include: (1) the cost of transforming tentative transactions to base transactions. (2) the cost of query processing. Each transaction is usually composed of several queries. The processing of each query includes parsing, query validation, view resolution, optimization, plan compilation and execution. (3) the cost of concurrency control. Transactions in $\mathbf{SAV}$ are usually reexecuted concurrently in order to enhance the system throughput. (4) the cost of I/O. The updates of each tentative transaction must be forced to durable logs when it commits. In addition, query processing can cause more I/O operations. In contrast, the costs of the merging protocol include: (1) the cost of constructing $G(H_m, H_b)$, which can be built by parsing the log for H_m and the log for H_b only once if read operations (or read sets) are recorded in the log. [AJL98] shows that for canned systems read-set information can be extracted from transaction profiles by offline analysis, thus the performance penalty of logging reads is avoided. (2) the cost of computing $\mathbf{B}$. [Dav84] shows that each of the proposed back-out strategies can compute $\mathbf{B}$ in polynomial time. (3) the cost of forwarding updates. Forwarding the updates of $\mathbf{SAV}$ can be done within one transaction. So all the updates need be forced to durable logs only once.

[‡‡]Note that in canned systems transaction type information can be sent instead of codes.

At the mobile node, in the two-tier replication protocol we need to inform users the results of the reprocessed transactions in $\mathbf{SAV}$. In contrast, the costs of the merging protocol include: (1) the cost of constructing $G(H_m)$. (2) the cost of backing out $\mathbf{B}$. The reads-from transitive-closure can be extracted from the graph within $O(m)$ time where m is the size of the closure. Given the can-follow and the can-precede relation between transactions, both Algorithm 1 and Algorithm 2 can be done within $O(n^2)$ time where n is the length of H_m. The can-follow relation can be determined during the construction of $G(H_m)$. The detection of can precede relation is mentioned in section 5. The reads-from transitive-closure based approach may save more time, however, it may back out much more transactions than the rewriting approach in a system where many transactions commute with each other. (3) the cost of pruning. If the number of backed-out transactions is much smaller than n, then the cost of compensation or the undo approach is relatively very small.

In summary, the result of the comparison depends mainly on (1) how many transactions can be saved (the size of $\mathbf{SAV}$), (2) the characteristics of transactions (i.e., size, read-set, write-set, semantics), and the characteristics of the system (i.e., canned or non-canned). We can see that when the size of $\mathbf{SAV}$ is big enough the two-tier replication protocol can cause more I/O and CPU costs although it may cause less communication costs, thus the merging protocol can win. On the contrary, when the size of $\mathbf{SAV}$ is very small the merging protocol will probably lose.

7.2 Conclusion

In this paper, we present the method of merging histories instead of reprocessing to reduce the overhead of *two-tier replication*, a replication protocol to reduce the problem that update anywhere replication has unstable behavior as the workload scales up. When a tentative history is merged into the base history, a set of undesirable transactions (denoted $\mathbf{B}$) have to be backed out to resolve the conflicts between the two histories. Desirable transactions that are affected, directly or indirectly, by the transactions in $\mathbf{B}$ complicate the process of backing out $\mathbf{B}$. We present a family of novel rewriting algorithms for the purpose of backing out $\mathbf{B}$. By incorporating transaction semantics, our rewriting methods are strictly better at saving desirable tentative transactions than the traditional reads-from transitive-closure based approach. And in most cases our rewriting methods are better at saving desirable tentative transactions than an approach which is based only on commutativity.

References

[AJL98] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. Technical report, George Mason University, 1998. http://isse.gmu.edu/∼pliu/papers/dynamic.ps.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[Dav84] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):456–581, September 1984.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, 1996.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 249–259, San Francisco, CA, 1987.

[KLS90] H.F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, 1990.

[LAJ99] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovering from malicious transactions. *Distributed and Parallel Databases*, 1999. To appear.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[Wei88] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

A Proof Sketch of Theorem 4

Proof Sketch: Given a history H^s, showing that $T_i \in$ FPR(H^s) holds for each transaction $T_i \in$ CBTR(H^s) gives the proof. We prove this by induction on k where T_k is the kst transaction moved into CBTR(H^s).

Induction base: ($k = 1$) We want to show that $T_1 \in$ FPR(H^s). If there are no transactions between B_1 and T_1 which are in FPR(H^s), then T_1 will be moved into FPR(H^s) according to Algorithm 2 because T_1 can precede every transaction $T_j^{\emptyset}$ between B_1 and T_1. Otherwise, there must be some transaction T_j with a non-empty fix F_j staying between B_1 and T_1 (including B_1) in the rewritten history when T_1 is scanned in Algorithm 2. Here we assume that F_j is captured by Lemma 1. At this point, assume T_1 cannot precede $T_j^{F_j}$, then $F_j \cap (T_1.readset - T_1.writeset) \neq \emptyset$ because otherwise T_1 can precede $T_j^{F_j}$ (The reason is: for every state s_0 on which $T_j^{F_j} T_1$ is defined, replacing s_0 with another state s_1 where the value of each item x in $s_0 \cap F_j$ is replaced with x's value in F_j, then $T_j^{\emptyset} T_1$ is defined on s_1. According to Property 1, since T_1 can precede $T_j^{\emptyset}$, so $(T_j.readset - T_j.writeset) \cap T_1.writeset = \emptyset$. Since $F_j \subseteq (T_j.readset - T_j.writeset)$ according to Lemma 1, so $F_j \cap T_1.writeset = \emptyset$. So T_1 will not read or update any item in F_j. Therefore, $T_j^{F_j} T_1(s_0) = T_j^{\emptyset} T_1(s_1)$, and $T_1 T_j^{F_j}(s_0) = T_1 T_j^{\emptyset}(s_1)$. Since T_1 commutes backward through T_j, so $T_j^{\emptyset} T_1(s_1) = T_1 T_j^{\emptyset}(s_1)$. Therefore, $T_j^{F_j} T_1(s_0) = T_1 T_j^{F_j}(s_0)$, so T_1 can precede $T_j^{F_j}$). Therefore, $\exists x$, such that, $x \in F_j \cap (T_1.readset - T_1.writeset)$. Since $x \in F_j$, so according to Algorithm 2 there must be a transaction T_p, such that T_p is now in FPR(H^s), and $x \in T_p.writeset$. Otherwise, x will not be put into F_j by Lemma 1. Hence $T_p.writeset \cap (T_1.readset - T_1.writeset) \neq \emptyset$. This conflicts with Property 1 since T_1 can precede $T_p^{\emptyset}$. Therefore, T_1 can precede $T_j^{F_j}$. So T_1 can precede every transaction between B_1 and T_1, so T_1 will be moved into FPR(H^s).

Induction hypothesis: for each $1 \leq k \leq n$, if $T_k \in$ CBTR(H^s), then $T_k \in$ FPR(H^s).

Induction Step: Let $k = n + 1$, then when T_k is scanned in both algorithms, every transaction T_j, which is between B_1 and T_k in the rewritten history generated by Algorithm 2, is between B_1 and T_k in the rewritten history generated by the commutes-backward-through rewriting algorithm. Therefore, T_k commutes backward through every such T_j. For the same reason as in the *induction base* step, we know that T_k will be moved into FPR(H^s).

This completes the proof.