Lazy Database Replication with Ordering Guarantees

Khuzaima Daudjee and Kenneth Salem
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{kdaudjee, kmsalem}@db.uwaterloo.ca

Abstract

Lazy replication is a popular technique for improving the performance and availability of database systems. Although there are concurrency control techniques which guarantee serializability in lazy replication systems, these techniques may result in undesirable transaction orderings. Since transactions may see stale data, they may be serialized in an order different from the one in which they were submitted. Strong serializability avoids such problems, but it is very costly to implement. In this paper, we propose a generalized form of strong serializability that is suitable for use with lazy replication. In addition to having many of the advantages of strong serializability, it can be implemented more efficiently. We show how generalized strong serializability can be implemented in a lazy replication system, and we present the results of a simulation study that quantifies the strengths and limitations of the approach.

1 Introduction

Replication is a popular technique for improving the performance and availability of a database system. In distributed database systems, replication can be used to bring more computational resources into play, or to move data closer to where it is needed.

A key issue in replicated systems is synchronization. Synchronization schemes can be classified as eager or lazy [7]. Eager systems propagate updates to replicas within the scope of the original updating transaction. This makes it relatively easy to guarantee transactional properties, such as serializability. However, since such transactions are distributed and relatively long-lived, the approach does not scale well. Lazy schemes, on the other hand, update replicas using separate transactions.

In this paper we address a problem related to the perceived ordering of execution of transactions in lazy replicated systems. To understand this problem, consider a customer of an electronic ticket reservation service. This customer issues two transactions[1] sequentially against the ticket reservation database. The first transaction, $T_{reserve}$, books tickets for a concert and records the bookings in the customer's virtual shopping bag . The second transaction, T_{status}, reports the contents of the customer's shopping bag.

Since the customer issues T_{status} after $T_{reserve}$ has finished, it is reasonable for him to expect that T_{status} will "see" the effects of $T_{reserve}$. That is, T_{status} should report that the shopping bag contains the concert tickets. This is, in fact, the behavior that many database systems will exhibit. For example, if the database system uses eager replication or no replication, and it uses two-phase locking, T_{status} will appear to follow $T_{reserve}$. However, if the ticketing database is replicated and lazily synchronized, this may not be the case. The $T_{reserve}$ and T_{status} transactions may run against different copies of the data and, in particular, T_{status} may run against a copy that does not yet reflect the updates made by $T_{reserve}$.

There are several concurrency control algorithms that guarantee global serializability in lazy replicated systems [5, 4, 16, 13]. However, serializability is not enough to solve this transaction ordering problem. Serializability ensures that transactions will appear to have executed sequentially in *some* order. However, it does not guarantee that the serialization order will be consistent with the order in which the transactions are submitted to the system. A concurrency control algorithm that guarantees only serializability may serialize T_{status} before $T_{reserve}$ even if $T_{reserve}$ is submitted first. This happens when T_{status} is allowed to see a stale copy, as described above.

Our transaction ordering problem could be solved by using a concurrency control that guarantees *strong serializability* [3], rather than plain serializability. Informally, strong serializability means that sequential transactions must be serialized in the order in which they are sub-

[1] Perhaps this customer issues these transactions indirectly, by interacting with a web service provided by the e-ticketer.

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

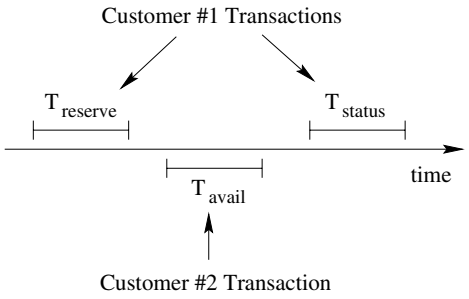

Figure 1. Execution Example with Two Customers

mitted. In our example, T_{status} would have to be serialized after $T_{reserve}$. However, none of the proposed concurrency controls for lazily replicated systems guarantee strong serializability, and there is good reason for this. Although strong serializability would address our transaction ordering problem, it is *too* strong. In a strongly serializable system, once one copy of the data has been updated, all other copies are effectively rendered useless until they, too, have been updated. Showing a stale copy to a subsequent transaction will violate the strong serializability guarantee. This effectively neutralizes many of the advantages of using lazy replication in the first place.

In this paper, we address this transaction ordering problem in lazy replicated systems. We make two contributions to the body of work in this area. First, since serializability is not strong enough to address the transaction ordering problem and strong serializability is too strong, we propose a new criterion called *strong session serializability*. The idea behind strong session serializability can be illustrated with the help of the diagram in Figure 1. The figure shows the transactions $T_{reserve}$ and T_{status} that have already been described. In addition, the figure shows a transaction T_{avail}, which is issued by a different customer. T_{avail} simply checks ticket availability for the concert.

With strong session serializability, we preserve the ordering of $T_{reserve}$ and T_{status}. However, we do not (necessarily) preserve the ordering of T_{avail} with respect to $T_{reserve}$ and T_{status}. The intuition is that the ordering of $T_{reserve}$ and T_{status} is important, in this case because the sequential nature of those transactions is observed by Customer 1. However, the ordering of T_{avail} relative to the other two transactions is not directly observed by either customer. Indeed, customers do not observe other customers' requests at all, except indirectly through their effects on the database. In our example, strong session serializability frees the system to serialize T_{avail} before or after $T_{reserve}$, and before or after T_{status}.

In general, strong session serializability allows us to specify which transaction orderings are important and which are not. As we describe in Section 2, this is accomplished by grouping transactions into sessions. Transaction ordering is preserved between transactions in the same session, but not between transactions in different sessions.

By ignoring unimportant transaction orderings, we expect to be able to implement strong session serializability efficiently. Our second contribution is a pair of algorithms for implementing strong session serializability in lazy replicated systems. These algorithms are presented in Section 3.2. We have used simulation to compare the cost of providing strong session serializability using these algorithms with the cost of providing plain serializability, and with the cost of providing strong serializability. The results of these experiments are presented in Section 5. The experiments show that, under the right conditions, strong session serializability can be provided almost as efficiently as plain serializability, and at a much lower cost than strong serializability.

Strong session serializability provides a basis for capturing applications' transaction ordering constraints, while simultaneously providing sufficient flexibility to allow those constraints to be enforced in lazy replicated systems.

2 Sessions and Strong Session Serializability

Breitbart, Garcia-Molina and Silberschatz [3] used the term *strong serializability* to mean serializability in which the ordering of non-concurrent transactions is preserved in the serialization order. Since we are concerned in this paper with replicated databases, we present here a slightly modified version of their definition that is appropriate when there is replication.

Definition 2.1 Strong 1SR: *A transaction execution history H is strongly serializable (Strong 1SR) iff it is 1SR and, for every pair of committed transactions T_i and T_j in H such that T_i's commit precedes the first operation of T_j, there is some serial one-copy history equivalent to H in which T_i precedes T_j.*

A transaction execution history over a replicated database is one-copy serializable (1SR) if it is equivalent to a serial history over a single copy of the database [2].

As discussed in Section 1, strong 1SR can be too strong: it requires the enforcement of transaction ordering constraints that may be unnecessary and costly. For example, strong 1SR requires T_{avail} to be serialized after $T_{reserve}$ in the example of Figure 1. In a replicated system, this means that T_{avail} cannot run at a site until $T_{reserve}$'s updates have been propagated there.

Strong session serializability captures the idea that ordering constraints may be necessary between some pairs of transactions, but not between others. Abstractly, we use *sessions* as a means of specifying which ordering constraints

are important and which are not. A session is simply a set of transactions. The transactions in an execution history H are partitioned into one or more sessions. Ordering constraints will be enforced among transactions in a single session, but not between transactions from different sessions.

We use a *session labeling* to identify which transactions are assigned to each session:

Definition 2.2 Session Labeling: *A session labeling L_H of an execution history H assigns a session label (identifier) to each transaction in H.*

We use the notation $L_H(T)$ to refer to the session label of transaction T. Given an execution history H and a labeling L_H, we define our correctness criterion as follows:

Definition 2.3 Strong Session 1SR: *A transaction execution history H is strong session 1SR under labeling L_H iff it is 1SR and, for every pair of committed transactions T_i and T_j in H such that $L_H(T_i) = L_H(T_j)$ and T_i's commit precedes the first operation of T_j, there is some serial one-copy history equivalent to H in which T_i precedes T_j.*

To use Definition 2.3, we must specify a session labeling for the transactions. The appropriate choice depends on the requirements of the application. For example, one natural choice might be to associate one session with each client application connection to a database server. This would have the effect of ordering the transactions over a single connection, but not across connections. In the case of the example in Figure 1, we wish to enforce ordering constraints among the transactions generated by a single customer, but not between transactions generated by different customers. Therefore, it is natural to use a distinct session label for each customer's interactions with the reservation system. In a three-tier web services environment, the customer sessions may be tracked by the application server or web server using cookies or a similar mechanism. In this case, the upper tiers can create session labels and pass them to the database system to inform it of the ordering constraints.

Strong session 1SR is stronger than 1SR, but weaker than strong 1SR. If L_H assigns the same label to all transactions in H, then strong session 1SR reduces to strong 1SR. Conversely, if L_H assigns a distinct label to each transaction in H, then no ordering constraints are important, and strong session 1SR reduces to 1SR. Otherwise, strong session 1SR will be somewhere in between. In the example of Figure 1, suppose that the first customer's transactions ($T_{reserve}$ and T_{status}) get one session label, and the second customer's transaction (T_{avail}) gets a different session label. Table 1 summarizes the transaction serialization orders that would be allowed under 1SR, strong 1SR and strong session 1SR under this labeling. As the example shows, strong session 1SR allows more flexibility than strong 1SR, while preserving the important ordering constraints.

	Strong 1SR	Strong Session 1SR	1SR
$T_{reserve} < T_{avail} < T_{status}$	ok	ok	ok
$T_{reserve} < T_{status} < T_{avail}$	-	ok	ok
$T_{avail} < T_{reserve} < T_{status}$	-	ok	ok
$T_{avail} < T_{status} < T_{reserve}$	-	-	ok
$T_{status} < T_{reserve} < T_{avail}$	-	-	ok
$T_{status} < T_{avail} < T_{reserve}$	-	-	ok

Table 1. Serialization Orders Permitted by Various Correctness Criteria

3 Enforcing Strong Session Serializability

In this section we present two algorithms for enforcing strong session serializability in a replicated database system with lazy update propagation. Our presentation will proceed in two steps. First, we will describe a basic replicated database system. The basic system uses well-known transaction execution and update propagation mechanisms to guarantee global one-copy serializability (1SR), but not strong session 1SR. Next, we will present a *session manager* that extends the basic system. With the addition of the session manager, the system guarantees strong session 1SR.

3.1 The Basic System

Figure 2 illustrates the architecture of the basic system. A primary site holds the primary copy of the database, and one or more secondary sites hold secondary copies of the database. Each site consists of an autonomous database management system with a local concurrency controller. For the purposes of this paper, we will assume that the database is fully replicated at the secondary sites. This is not necessary for the concurrency controls that we will describe, but it simplifies the presentation. If the database were not fully replicated, an additional mechanism would be needed to route transactions to the sites that hold the particular data they require.

Clients connect to the secondary sites and submit streams of transactional database operation requests. We assume that read-only transactions are distinguished from update transactions in the request streams. Each transaction is executed at a single site. Read-only transactions are executed at the secondary site to which they are submitted. Update transactions are forwarded by the secondaries to the primary and executed there. According to the classification proposed by Gray and colleagues [7], this is a lazy master architecture.

A desirable feature of this architecture is that an arbitrary number of secondary sites can be added to scale the system

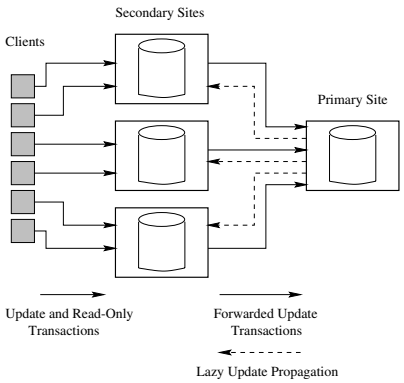

Figure 2. Lazy Master System Architecture

with increasing read-only transaction workloads. The architecture is best suited to read-mostly workloads since the primary site is a potential update bottleneck. Many common workloads in application areas such as decision support and web commerce are read-mostly [19, 11, 20]. However, if the update load at the primary does become an issue, data partitioning can be used to distribute the update load over multiple primary sites. As discussed in Section 3.1.1, the important property of the primary site, for our purposes, is that it should be capable of generating transaction sequence numbers and propagating updates in serialization order.

The basic system uses the following general approach to ensure global serializability. All update transactions are serialized by the local concurrency control at the primary site. Updates are propagated lazily to the secondary sites, and are installed there in the same order in which they were serialized at the primary. Thus, at any time, each secondary site will hold a snapshot, probably stale, of the primary database. Read-only transactions run against these snapshots. The secondary sites are not synchronized with each other, so at any given time some secondary sites may be more or less fresh than others. There are several specific issues that must be addressed in order to make this general approach work. We discuss these in the next two subsections.

3.1.1 The Primary Site

We assume that the primary site is capable of associating a *sequence number* with each update transaction committed there. We will use $seq(T)$ to denote the sequence number of transaction T. We also assume that these sequence numbers are consistent with the local transaction serialization order at the primary site. That is, $seq(T_1) < seq(T_2)$ if and only if T_1 precedes T_2 in the local serialization order. For our purposes it is not important how the sequence numbers are generated as long as they are available. However, one simple way to implement such sequence numbers is to use the log sequence numbers of the transactions' commit records.[2]

Updates made by committed transactions, together with the transactions' sequence numbers, are propagated lazily from the primary site to the secondary sites. Lazy propagation means that propagation occurs some time after the update transaction has been committed. We assume that the propagation mechanism propagates updates in serialization order, and that updates are not lost or reordered during propagation. If the primary database resides at a single site as shown in Figure 2, this is relatively easy to achieve using log sniffers, triggers, or other mechanisms. If, for performance reasons, the primary database is partitioned over several sites, it is necessary for these sites to generate a single, unified stream of updates (consistent with the primary serialization order) for propagation to the secondary sites. This is more difficult in the multi-site case than it is in the single site case, but it is certainly possible. For example, Liu and colleagues have described a log-sniffing mechanism that merges updates from the logs of a partitioned database system into a single stream [10]. We will not be concerned in this paper with the exact format of the change log or the update propagation messages.

3.1.2 The Secondary Sites

At each secondary site, propagated updates are placed in a FIFO update queue. A *refresh* process removes propagated updates from the update queue and applies them to the local database copy. Note that there is an independent refresh process at each secondary site. For each propagated update transaction, the refresher uses a local transaction to remove the transaction's updates from the update queue and then apply them to the local secondary database copy. These local transactions are called *refresh transactions*. Thus, for every update transaction at the primary there is eventually one corresponding refresh transaction at each secondary site.

Refresh transactions are also used to maintain a database sequence number, $seq(DB)$, at each secondary site. Each secondary site has its own $seq(DB)$, which is independent of the sequence numbers at the other secondary sites. When a refresh transaction applies updates at a secondary site, it also sets that site's $seq(DB)$ to match the transaction sequence number that was propagated with those updates. These sequence numbers are not needed by the basic system to ensure serializability. However, as described in Section 3.2, they are used by the session managers to enforce strong session serializability.

It is important that refresh transactions at each secondary site be serialized in the order in which their correspond-

[2] This approach also requires that the local concurrency control at the primary site guarantees commitment ordering [15]. Commitment ordering is a stronger property than strong serializability, which is required at the primary site if Theorems 2 and 3 are to hold.

ing update transactions were serialized at the primary site. We use a simple mechanism to achieve this. First, the refresh process performs the refresh transactions sequentially in the order in which updates are retrieved from the local update queue. Second, the local concurrency controller at the secondary is required to guarantee the strong serializability property. This property ensures that sequentially executed transactions are serialized in the order in which they are executed.[3] Note that it is possible to use other techniques, e.g. ticketing or transaction conflict analysis [6, 8], to order the refresh transactions. However, since our approach depends only on the proper serialization of the refresh transactions and not on the particular mechanism used to achieve it, we will stick with the simple sequential execution mechanism in this paper.

Theorem 1 *The basic system guarantees 1SR.*

Proof: Let T_i represent the ith transaction in the local serialization order at the primary, and let S_i represent the database state at the primary site that results from T_i's execution. Without loss of generality (since secondary sites are independent of one another), suppose there is a single secondary site, and let R_i represent the ith refresh transaction to commit at that site. Because (1) the primary site propagates updates in serialization order, (2) updates are not reordered during propagation, (3) the refresh process applies update transactions in the order received, and (4) the local concurrency control at the secondary site ensures strong serializability, R_i applies T_i's updates to the database at the secondary site. Since this is true for all i, and since refresh transactions are the only update transactions at the secondary site, the secondary database will be in state S_i as a result of R_i's execution.[4] Now consider an aribtrary read-only transaction T_r that is serialized after R_i and before R_{i+1} by the local concurrency control at the secondary site. T_r sees state S_i. Thus, it can be serialized after T_i and before T_{i+1} in an equivalent one-copy serial schedule. All other read-only transactions at the secondary can be serialized the same way.□

3.2 The Session Manager

In this section, we show how to augment the basic system so that it will guarantee strong session 1SR. Previously, we assumed that each client submits a sequence of transactions to a secondary site. We will now make the additional assumption that each client's transactions constitute a session. That is, we wish to enforce ordering constraints among the transactions submitted by a single client, but not between transactions submitted by different clients. Each transaction has associated with it, either explicitly or implicitly, a session label, so that the secondary site can tell which transactions belong to which session.

This simple session model has two properties which are significant to the algorithms that we will describe. The first property is that the transactions within a session are submitted sequentially, not concurrently. The second property is that each session is associated with a single secondary site. A relaxation of the first property would be relatively simple to handle. We have chosen to retain it because it simplifies the presentation of our algorithms. The second property, on the other hand, allows the secondary sites to operate independently of one another. Without it, enforcement of strong session 1SR would require coordination among the secondary sites.

Each secondary site has a *session manager*. The session managers, together with the propagation, refresh, and local concurrency control mechanisms described in Section 3.1, are responsible for enforcing strong session 1SR. Each session manager is responsible for the sessions that are associated with its secondary site.

3.2.1 The BLOCK Algorithm

We propose two algorithms that can be used by the session managers to enforce strong session 1SR. The first algorithm, BLOCK, is summarized in Figure 3. Each session manager maintains, for each of the sessions s for which it is responsible, a sequence number $seq(s)$. A session's sequence number essentially indicates database state "seen" by the most recently committed transaction in that session.

For each read-only transaction T in session s:
  WAIT UNTIL $seq(DB) \geq seq(s)$ AT SECONDARY SITE
  START T AT SECONDARY SITE
  PERFORM T'S READS AT SECONDARY SITE
  COMMIT OR ABORT T AT SECONDARY SITE
  IF T COMMITTED THEN
    $seq(s) \leftarrow seq(DB)$
  ENDIF

For each update transaction T in session s:
  START T AT PRIMARY SITE
  PERFORM T'S READS/WRITES AT PRIMARY SITE
  COMMIT OR ABORT T AT PRIMARY SITE
  OBTAIN $seq(T)$ FROM PRIMARY SITE
  IF T COMMITTED THEN
    $seq(s) \leftarrow seq(T)$
  ENDIF

Figure 3. The BLOCK Algorithm

[3] Strong serializability is guaranteed by well-known concurrency controls such as two-phase locking.

[4] There is an implicit assumption here that all copies of the database start in the same state.

The BLOCK algorithm works by delaying read-only transactions, if necessary, until the database state at the secondary site is at least as great as $seq(s)$.[5] This, plus the local concurrency control at the secondary site, is sufficient to ensure that each read-only transaction will be serialized after its predecessors in its session.

Theorem 2 *If each site guarantees strong serializability locally, then the* BLOCK *algorithm, in conjunction with the propagation and refresh mechanisms of the basic system, guarantees global strong session 1SR.*

Proof: Suppose that the claim is false, which means that there exists a pair of transactions T_1 and T_2 in the same session s for which T_1 is executed before T_2 but T_1 cannot be serialized before T_2. There are four cases to consider:
Case 1: Suppose T_1 and T_2 are update transactions. T_1 and T_2 both execute at the primary site. Since the primary site ensures strong serializability and since T_2 starts after T_1 finishes, T_1 is serialized before T_2, a contradiction.
Case 2: Suppose T_1 is a read-only transaction and T_2 is an update transaction. Since T_1 precedes T_2 and T_2 precedes its refresh transaction (T_2^R), T_1 precedes T_2^R at the secondary site. Since the secondary site ensures strong serializability, T_1 is locally serialized before T_2^R at the secondary, and thus it is globally serialized before T_2, a contradiction.
Case 3: Suppose T_1 is an update transaction and T_2 is a read-only transaction. When T_1 commits, $seq(s)$ is set to $seq(T_1)$. The blocking condition ensures that no subsequent read-only transaction in s can run until $seq(DB)$ is at least as large as $seq(T_1)$. Thus, T_2 does not execute until sometime after T_1's refresh transaction has finished. Since the secondary site ensures strong serializability, T_2 is serialized after T_1's refresh transaction at the secondary site, and therefore after T_1 in the global serialization order, a contradiction.
Case 4: Suppose both T_1 and T_2 are read-only transactions. Both transactions run at the secondary site. Since strong serializability is guaranteed locally there, T_2 sees the database in the same state as T_1, or in a later state. Thus, it can be serialized after T_1, a contradiction.□

3.2.2 The FORWARD Algorithm

The second algorithm, called FORWARD, handles update transactions the same way that the BLOCK algorithm does, but it handles read-only transactions differently. The handling of read-only transactions under the FORWARD algorithm is summarized in Figure 4.

As was the case under the BLOCK algorithm, each session manager maintains, for each of its sessions, a sequence number $seq(s)$. The FORWARD algorithm works by forwarding some read-only transactions to the primary site for execution. Specifically, if the database at the secondary site is too stale to execute a particular read-only transaction without violating session ordering constraints, then the read-only transaction is forwarded to the primary site. Since the primary site is always in the "latest" state, this ensures that the read-only transaction will "see" its session predecessors. However, it increases the load at the primary site, which is also responsible for all update transactions.

```
For each read-only transaction T in session s:
  IF seq(DB) ≥ seq(s) AT SECONDARY SITE THEN
    START T AT SECONDARY SITE
    PERFORM T'S READS AT SECONDARY SITE
    COMMIT OR ABORT T AT SECONDARY SITE
    IF T COMMITTED THEN
      seq(s) ← seq(DB)
    ENDIF
  ELSE
    START T AT PRIMARY SITE
    PERFORM T'S READS AT PRIMARY SITE
    COMMIT OR ABORT T AT PRIMARY SITE
    OBTAIN seq(T) FROM PRIMARY SITE
    IF T COMMITTED THEN
      seq(s) ← seq(T)
    ENDIF
  ENDIF
```

Figure 4. Read-Only Transactions Under the FORWARD Algorithm

In Section 3.1.1, we assumed that the primary site is able to associate a sequence number with each update transaction that commits there. When the FORWARD algorithm is used, it is also necessary for the primary site to associate a sequence number with each read-only transaction that is forwarded to it. The primary site should not associate new sequence numbers with read-only transactions. Rather, each read-only transaction should be assigned the same sequence number as the most recent (in the primary's local serialization order) committed update transaction.

.

Theorem 3 *If each site guarantees strong serializability locally, then the* FORWARD *algorithm, in conjunction with the propagation and refresh mechanisms of the basic system, guarantees global strong session 1SR.*

Proof: Suppose that the claim is false, which means that there exists a pair of transactions T_1 and T_2 in the same session for which T_1 is executed before T_2 but T_1 cannot be serialized before T_2. There are four cases to consider:

[5] Recall that $seq(DB)$, which indicates the current state of the database at the secondary site, is maintained by the refresh transactions.

Case 1: Suppose T_1 and T_2 both execute at the primary. This is the same as Case 1 in Theorem 2.
Case 2: Suppose T_1 executes at the secondary and T_2 executes at the primary. If T_2 is an update transaction, this is the same as Case 2 in Theorem 2. If T_2 is a read-only transaction, let T_u be the latest update transaction whose effects were seen by T_1. T_u preceded T_1, and thus T_2, at the primary site. Since the primary ensures strong serializability, T_2 must be serializable after T_u, and thus after T_1.
Case 3: Suppose T_1 executes at the primary and T_2 executes at the secondary. When T_1 commits, $seq(s)$ is set to $seq(T_1)$. The forwarding condition ensures that no subsequent transaction in s can run at the secondary until $seq(DB)$ is at least as large as $seq(T_1)$. If T_1 is an update transaction, then the condition ensures that T_2 does not run until sometime after T_1's refresh transaction has finished. Since the secondary site ensures strong serializability, T_2 is serialized after T_1's refresh transaction at the secondary site, and therefore after T_1 in the global serialization order, a contradiction. If T_1 is a read-only transaction, a similar argument shows that T_2 sees at least the same updates as T_1, and thus can be serialized after T_1.
Case 4: Suppose T_1 and T_2 both execute at the secondary. This is the same as Case 4 in Theorem 2.□

3.2.3 Discussion

Both the BLOCK algorithm and the FORWARD algorithm require the session manager to read the value of $seq(DB)$. As a result, the session manager conflicts with refresh transactions running at its secondary site, and $seq(DB)$ is a potential database hotspot. However, the session manager need not read $seq(DB)$ within the scope of the requested transaction (T in Figures 3 and 4). Instead, it can minimize contention for $seq(DB)$ by using a separate, short transaction to read it.

A second issue with both algorithms is maintaining the strong session 1SR guarantee in the case of a failure of a session manager. Under either algorithm, the session manager updates $seq(s)$ outside the scope of the requested transaction T. This creates a small failure window between the commit of T and the change to $seq(s)$. Of course, this window could be closed completely by simply updating $seq(s)$ within the scope of T in Figures 3 and 4. However, this may hurt performance by increasing the size of T, and by turning read-only transactions into update transactions. Furthermore, updating $seq(s)$ within T may imply reading $seq(DB)$ within T, since the new value of $seq(s)$ may come from $seq(DB)$. This further increases the size of T, and may introduce conflict between T and refresh transactions.

To avoid these potential performance problems, we allow the failure window to exist, and we address the problem by properly reinitializing $seq(s)$ when the session manager recovers from a failure. Specifically, the session manager should initialize $seq(s)$ (for each of its sessions s) to the current sequence number of the database at the *primary* site. The session manager can determine the primary's sequence number by executing a dummy transaction T_{init} at the primary, obtaining T_{init}'s sequence number from the primary, and then initializing $seq(s)$ to $seq(T_{init})$.

Parameter	Description	Default
num_clients	number of clients	varies
num_sec	number of secondary sites	5
think_time	mean client think time	7s
session_time	mean session duration	15 min.
update_tran_prob	probability of an update transaction	20%
conflict_prob	transaction conflict probability	20%
tran_size	mean number of operations per transaction	10
op_service_time	service time per operation	0.02s
update_op_prob	probability of an update operation	30%
propagation_delay	propagator think time	10s

Table 2. Simulation Model Parameters

4 Simulation Model

We have developed a simulation model of the lazy master replicated database system described in Section 3.1. We have used the simulator to compare the Block and Forward global concurrency control protocols described in Section 3.2 and to determine the impact of the correctness criterion (1SR, strong session 1SR, or strong 1SR) on system performance. The model is implemented in C++ using the CSIM simulation package [12].

The model consists of one CSIM resource (server) for each site in the system. Client processes simulate the execution of transactions requested by system clients. Each client process is associated with a particular secondary site, and it submits all of its transactions to that site. The client processes are uniformly distributed over the available secondary sites. In addition, there are processes that simulate update propagation and refresh transactions at the secondary sites. The model's parameters are summarized in Table 2.

There are *num_clients* client processes. Each client process generates a series of sessions. Session lengths are exponentially distributed with a mean length of *session_time*.

Within each session, a client iteratively generates transactions, with exponentially distributed think times of mean length *think_time* between them. When a session is finished, the client immediately initiates a new session so that the total number of concurrent sessions is *num_clients* at all times.

A transaction is an update transaction with probability *update_tran_prob*, otherwise it is a read-only transaction. Our default transaction mix is 80% read-only, 20% update, although we also ran some experiments with a 95%/5% mix. Note that the "shopping" mix specified in the TPC-W benchmark is an 80/20 mix of read-only and update web interactions, and the "browsing mix" is a 95/5 mix.[6] The default *think_time* and *session_time* values are also taken from the TPC-W benchmark specification [19].

In the transaction execution model, each transaction proceeds first to the session manager at the secondary site to which it is submitted. The session manager directs each transaction to the primary site or to the secondary site for execution. If the session manager is using the BLOCK algorithm, it may also cause the transaction to block.

After leaving the session manager, each transaction proceeds to the local concurrency control at the site to which it is assigned. We use a simple model of the local concurrency control, which works as follows. Each newly arriving transaction has a probability *conflict_prob* of conflicting with each transaction that is already in the local system (either running or waiting to run). If a newly arriving transaction does not conflict with any existing transactions, it begins running immediately. Otherwise, it waits for all conflicting transactions to complete before it begins running.

Once a transaction has finished with the local concurrency control, it can run. Each transaction consists of a number of operations. Each operation takes time *op_service_time* at the server. The number of operations in each transaction is randomly chosen in the range *tran_size* plus or minus 50%. For update transactions, each operation has probability *update_op_prob* of being an update, otherwise it is a read. All transactions running at a given site share that site's single server, which uses a round-robin queuing discipline with a time slice of 0.001 seconds.

Propagation processes running at each site are used to simulate the propagation mechanism. There is one propagation process per site. At the primary site, the propagation process goes through a series of propagation cycles, with a delay of *propagation_delay* time units between the end of one cycle and the beginning of the next. During each propagation cycle, the process propagates all transactions that have committed at the primary since the last propagation cycle. To do this, the process generates a single message describing the updates of all of these transactions and broadcasts the message to the secondary sites. The primary's propagation process consumes *op_service_time* during each propagation cycle. It does not use the local concurrency control at the primary site, since we assume that the propagator is implemented as a log sniffer.

Propagation messages are assumed to be delivered at the secondary sites after a delay of length *propagation_delay*. The simulation does not include an explicit resource to represent the network. We assume that the network has sufficient capacity so that network contention is not a significant contributor to the propagation delay.

At each secondary site, the local propagation process receives the broadcast propagation messages and installs the transaction update information in an update queue in the local database. The propagation process consumes *op_service_time* for each propagation message it handles. It does not use the local concurrency control when it inserts update information into the update queue. The queue operations conflict only with the refresh processes, which reads from the other end of the queue. Thus, we assume that any contention would be insignificant.

At each secondary site there is a set of refresh processes. Each refresh process iteratively waits to obtain one transaction's update records from the local update queue, and then runs a single refresh transaction to apply those updates to the secondary database. The refresh transaction must pass through the local concurrency control at the secondary site. It has probability *conflict_prob* of conflicting with each read-only transaction at the site. In addition, we force a conflict between every pair of refresh transactions at a site. This, together with our local concurrency control mechanism, ensures that refresh transactions are not reordered by the local concurrency control. The refresh process consumes *op_service_time* at the secondary server to retrieve the transaction record from the update queue. It consumes an additional *op_service_time* per update operation during the execution of the refresh transaction itself.

5 Performance Analysis

We ran a series of experiments using the simulation model, with two goals in mind. The first was to determine the cost, in terms of transaction throughput and response time, of providing strong session 1SR rather than the weaker 1SR. The second was to compare the two algorithms, BLOCK and FORWARD, that guarantee strong session 1SR. To facilitate these comparisons we implemented two more algorithms for the session manager, in addition to BLOCK and FORWARD:

ALG-1SR: ALG-1SR provides only global serializability (1SR), not strong session 1SR. ALG-1SR simply routes all update transactions to the primary site, and

[6] The TPC-W benchmark specifies the read/write mix in terms of web interactions, not database transactions. There is not necessarily a one-to-one correspondence between the two.

all read-only transactions to the secondary site. Transactions are never blocked by Alg-1SR itself, although they may, of course, be blocked by the local concurrency control at their assigned site. Under this algorithm, the system operates like the basic system described in Section 3.1.

ALG-STRONG-SITE-1SR: This is the same as the BLOCK algorithm of Figure 3 except that there is only a single session per secondary site, rather than one session per client. Thus, ALG-STRONG-SITE-1SR maintains only one session sequence number ($seq(s)$) at each secondary site. It enforces many more transaction ordering constraints than those required by strong session 1SR, but fewer than would be required by a true implementation of strong 1SR. In our experiments we have used it as a surrogate for a strong 1SR algorithm. ALG-STRONG-SITE-1SR should result in performance no worse than (and probably significantly better than) that of any algorithm that provides true strong 1SR.

5.1 Methodology

For each run, the simulation parameters were set to the default values shown in Table 2, except as indicated in the descriptions of the individual experiments. Each run lasted for 35 simulated minutes. We ignored the first five minutes of each run to allow the system to warm up, and measured transaction throughput, response times and other statistics over the remainder of the run. Each reported measurement is an average over five independent runs. We computed 95% confidence intervals around these means. These are shown as error bars in the graphs.

5.2 Performance of the Default Configuration

We ran a series of experiments in which our default configuration, with five secondary sites, was subjected to load from an increasing number of clients. The results of these experiments are shown as throughput, read-only transaction response time and update transaction response time plots in Figures 5, 6 and 7, respectively. Each curve describes the behavior of one of the four global concurrency control algorithms (ALG-1SR, BLOCK, FORWARD and ALG-STRONG-SITE-1SR) that we considered.

At low loads, the FORWARD algorithm provides better read response times than BLOCK, and approximately the same overall throughput and response times as ALG-1SR. However, as the load increases, FORWARD breaks down because it saturates the primary site. The BLOCK algorithm is more consistent. Its performance is similar to that of ALG-1SR over the load range that we tested,

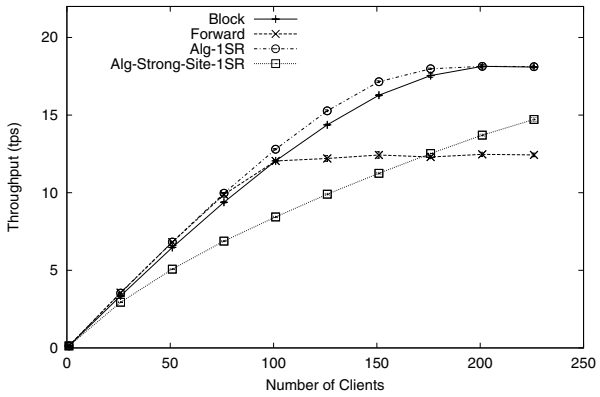

Figure 5. Transaction Throughput vs. Number of Clients

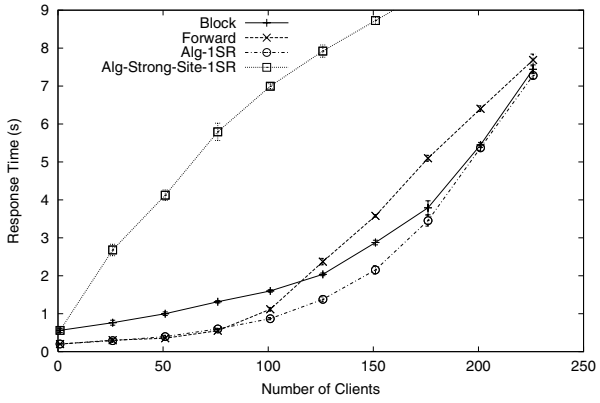

Figure 6. Read-Only Transaction Response Time vs. Number of Clients

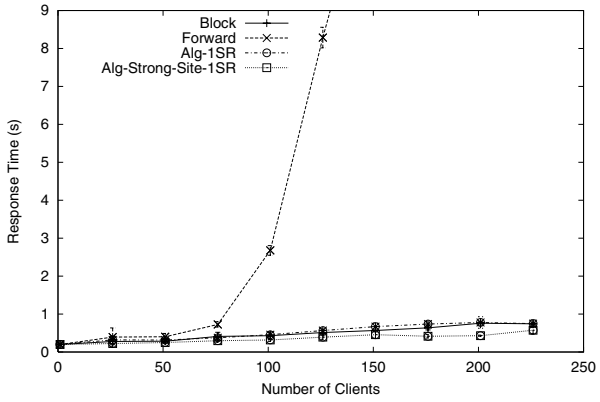

Figure 7. Update Transaction Response Time vs. Number of Clients

and it was much better than that of ALG-STRONG-SITE-1SR. This indicates that strong session 1SR, which is guaranteed by the BLOCK algorithm, can be achieved at about the same cost as plain 1SR. Furthermore, a comparison of the performance of ALG-STRONG-SITE-1SR with that of BLOCK shows that it is important to avoid enforcing unnecessary transaction ordering constraints. The BLOCK algorithm, which only enforces transaction ordering constraints within session, performs better than ALG-STRONG-SITE-1SR, which enforces additional constraints.

These results are sensitive to the update propagation delay, which is a key parameter in our performance model. Ideally, updates would be propagated to the secondary sites very quickly, and the secondary databases would be almost as fresh as the primary. In our model, this can be simulated by setting *propagation_delay* close to zero. We ran experiments under that condition (not shown), and found that all four of the algorithms have almost identical performance. That is, if all of the replicas are very current, strong session 1SR is easy to achieve, and it does not matter which of the algorithms is used to achieve it. In practice, however, scheduling at the primary site, network latencies, and batching may introduce propagation latencies, and will make very low latencies difficult to achive. A nice property of strong session 1SR is that it allows some propagation latency to be tolerated without impacting transaction throughput and response time. In particular, inter-transaction think times within a session should effectively hide propagation latency. In our default configuration, the mean transaction think time (7 sec) is comparable to the propagation latency (10 sec). This is why the BLOCK algorithm performs well in the results shown in Figures 5, 6 and 7.

5.3 Scalability

A desirable feature of our system is that the number of secondary sites can be scaled with the client load. To examine the effects of doing so, we ran an experiment in which both the number of secondary sites and the number of clients were gradually increased, with the number of clients per secondary site held constant at 20. Figures 8, 9, and 10 show the results of this experiment.

In these experiments, the BLOCK algorithm again performed about as well as ALG-1SR, and significantly better than ALG-STRONG-SITE-1SR. The FORWARD algorithm performed poorly, again because of the extra load it places on the primary site. Throughput for both BLOCK and ALG-1SR eventually peaks (with about 10 secondary sites) when the primary site saturates. Since the primary site is the bottleneck that eventually limits throughput, a key scalability parameter is the mix of read-only and update transactions in the workload. Figure 11 shows the result of the scalability experiment using a 95/5 read/write transaction mix in

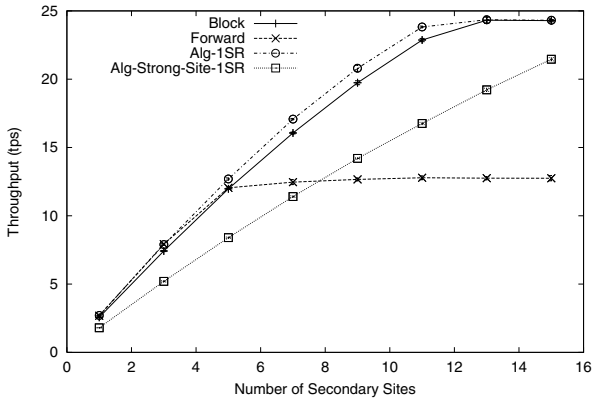

Figure 8. Transaction Throughput, 20 Clients per Secondary

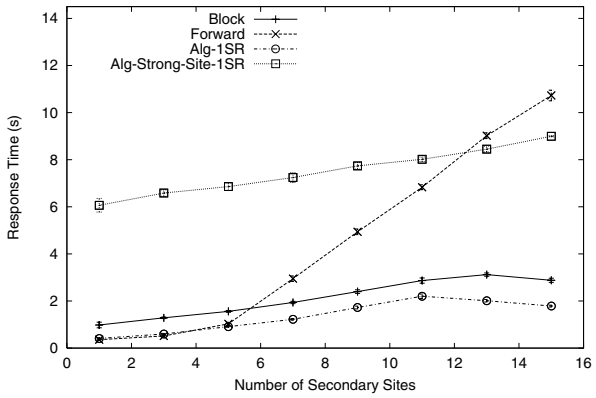

Figure 9. Read-Only Transaction Response Time, 20 Clients per Secondary

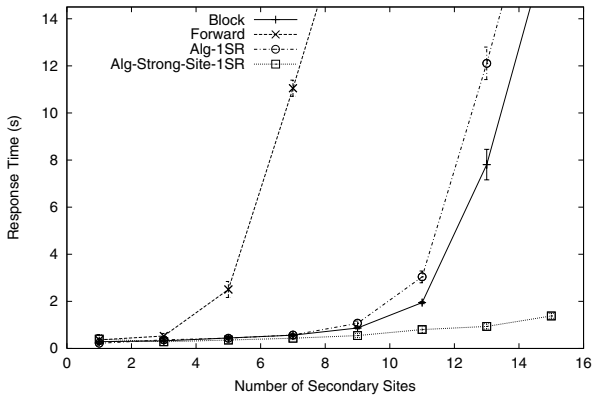

Figure 10. Update Transaction Response Time, 20 Clients per Secondary

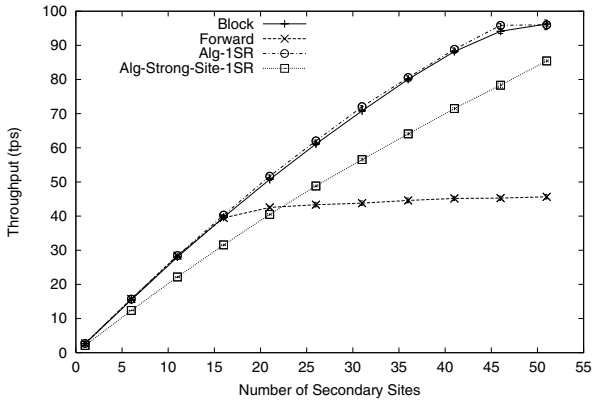

Figure 11. Transaction Throughput, 20 Clients per Secondary, 95/5 read/write mix

place of the default 80/20 mix. As might be expected, much greater scalability is achieved in this case (compare Figures 8 and 11).

6 Related Work

The techniques described in this paper are global concurrency controls implemented in a federated, replicated database system. Breitbart, Garcia-Molina and Silberschatz have provided a thorough overview of concurrency control issues for federated databases [3]. That overview is not concerned with issues of replication and data freshness. However, many of the general concerns of transaction ordering that are addressed by our basic system (Section 3.1) arise in all federated systems, whether they manage replicated data or not. Our definition of strong serializability is based on the one in that paper.

Bayou [18, 17] is a system that provides data freshness to user sessions through causal ordering constraints on read and write operations. However, Bayou guarantees only eventual consistency, which means that servers converge to the same database state only in the absence of updates. For replicated data, this is the same weak notion of consistency as convergence [21]. Bayou does not necessarily guarantee that constraints within a session are preserved. It is possible for the system to report that a requested session guarantee, e.g. read-your-writes, cannot be provided.

Ladin and colleagues [9] proposed a scheme that provides data freshness guarantees to read and write operations. In their work, if all writes are *forced* operations, they are directed to a single "primary" site, which orders them. This site uses a 2-phase commit protocol to propagate, in order, each forced write's update to a majority of replicated sites. Subsequent operations are guaranteed to see the effects of the forced writes. In this work there is no notion of sessions and thus no provision of session guarantees.

Some recent work has considered the specific problem of concurrency control for lazy master replicated database systems. Breitbart and colleagues have developed several protocols for guaranteeing 1SR in lazy master systems [4]. These protocols, called DAG(WT), DAG(T), and Backedge, operate in a more general environment than the one considered here. Different database objects may have their primary copies located at different sites, and an acyclic (except in the case of the Backedge protocol) site graph is used to guide the propagation of updates among the sites. Like our protocols, these rely on in-order propagation and application of updates. Other related work on concurrency control protocols includes the virtual sites protocol of Breitbart and Korth, the quorum consensus protocol of Satyanarayanan and Agrawal, which uses a gossip mechanism to lazily propagate updates to sites that have missed them, and the epidemic update propagation protocol of Agrawal and colleagues [5, 16, 1]. Pacitti, Minet, and Simon [13] proposed a lazy master update propagation protocol that uses a worse-case estimation of update propagation time, allowing a global ordering of refresh transactions. None of these techniques address the transaction ordering problem that we have considered, and none have a notion of transaction sessions. All guarantee 1SR, but not strong session 1SR or strong 1SR.

Pacitti and Simon [14] have studied the effects of different update propagation techniques on the freshness of replicated data. Our update propagation technique, in which propagation occurs after commit, would be classified as *deferred-immediate* in their work. The goal of their work is to keep the replicas as fresh as possible as efficiently as possible. They did not consider the relationship between freshness and transaction ordering and execution that we have attempted to exploit in our work.

King and colleagues considered techniques for maintaining a replicated database to be used as backup in case the primary copy fails [8]. Theirs is a lazy master approach, like ours. Their work did not consider execution of application transactions at the backup (secondary) site, so they were not concerned with global concurrency controls. However, they faced an issue that arises in all lazy replication techniques: how to ensure that the updates are applied in the same order at the secondary site as they were at the primary. They proposed the use of transaction conflict analysis to allow non-conflicting refresh transactions to run in parallel at the secondary site. A similar approach could be used in our system. However, it does require an analysis of transactions' read sets, as well as their updates.

7 Conclusion

In this paper, we proposed a new correctness criterion, strong session 1SR, for transaction scheduling. Strong session 1SR is a generalization of one copy serializability (1SR) and strong serializability (strong 1SR), and guarantees data freshness by allowing important transaction ordering constraints to be captured and unimportant ones to be ignored.

Starting with a basic system that provides only 1SR over lazily synchronized replicated data, we showed how to modify the system so that it ensures strong session 1SR. We proposed two global concurrency control algorithms, called Block and Forward, to achieve this. One works by delaying transactions that need to see fresher data, the other works by redirecting such transactions to the fresh database copy at the primary site. We studied the performance of our algorithms for the lazily synchronized replicated system. We found that when propagation latencies are low, that is, when the secondary database copies can be kept very fresh, ensuring strong session 1SR, and even strong 1SR, costs very little in terms of transaction throughput and response time. However, for longer, more realistic propagation latencies, strong 1SR becomes very difficult to achieve while strong session 1SR can be maintained almost as efficiently as 1SR. Of the two algorithms, Block has consistent performance at all load levels. Strong session 1SR appears to be a useful and practical basis for enforcing transaction ordering constraints in scalable replicated database systems.

Acknowledgement: We are grateful to M. Tamer Özsu for his comments on an earlier draft of this paper.

References

[1] D. Agrawal, A. E. Abbadi, and R. Steinke. Epidemic algorithms in replicated databases. In *Symposium on Principles of Database Systems*, pages 161–172, 1997.

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

[3] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–293, 1992.

[4] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 97–108, June 1999.

[5] Y. Breitbart and H. F. Korth. Replication and consistency: Being lazy helps sometimes. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 173–184, May 1997.

[6] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 314–323, 1991.

[7] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 173–182, 1996.

[8] R. P. King, N. Halim, H. Garcia-Molina, and C. A. Polyzois. Management of a remote backup copy for disaster recovery. *ACM Transactions on Database Systems*, 16(2):338–368, 1991.

[9] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[10] C. Liu, B. G. Lindsay, S. Bourbonnais, E. Hamel, T. C. Truong, and J. Stankiewitz. Capturing global transactions from multiple recovery log files in a partitioned database system. In *Proceedings of 29th International Conference on Very Large Data Bases*, Sept. 2003.

[11] D. A. Menasce, V. Almeida, R. H. Riedi, F. Ribeiro, R. C. Fonseca, and W. M. Jr. In search of invariants for e-business workloads. In *ACM Conference on Electronic Commerce*, pages 56–65, 2000.

[12] Mesquite Software Inc. *CSIM18 Simulation Engine (C++ version) User's Guide*, Jan. 2002.

[13] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3):237–267, 2001.

[14] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3-4):305–318, 2000.

[15] Y. Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *Proceedings of 18th International Conference on Very Large Data Bases*, pages 292–312, Aug. 1992.

[16] O. T. Satyanarayanan and D. Agrawal. Efficient execution of read-only transactions in replicated multiversion databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):859–871, 1993.

[17] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, and M. Theimer. Flexible update propagation for weakly consistent replication. In *Symposium on Operating Systems Principles*, pages 288–301, 1997.

[18] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *Conference on Parallel and Distributed Information Systems*, pages 140–149, 1994.

[19] Transaction Processing Performance Council. *TPC Benchmark W (Web Commerce)*, Feb. 2001. http://www.tpc.org/tpcw/default.asp.

[20] Transaction Processing Performance Council. *TPC Benchmark H (Decision Support) Revision 2.0.0*, 2002. http://www.tpc.org/tpch/default.asp.

[21] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998.