Lazy Database Replication with Freshness Guarantees

Khuzaima Daudjee
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
kdaudjee@uwaterloo.ca

Kenneth Salem
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
kmsalem@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 do not provide freshness guarantees. 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. It has many of the advantages of strong serializability, but 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. Gray et al [8] classified synchronization schemes into two general categories: eager and lazy. 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 transactions are distributed and relatively long-lived, the approach does not scale well. Lazy schemes, on the other hand, update replicas using separate transactions.

Several algorithms have been proposed for guaranteeing globally serializable executions when updates are lazy. [15, 5, 4] Although serializability is a desirable property, it may not be enough. Consider a very simple example involving an on-line ticketing service that uses two types of database transactions: *Reserve* and *Status*. The *Reserve* transaction books seats and records the booking in the customer's virtual shopping bag . The *Status* transaction reports the current contents of a customer's shopping bag.

Technical Report CS-2002-31, School of Computer Science, University of Waterloo, November, 2002..

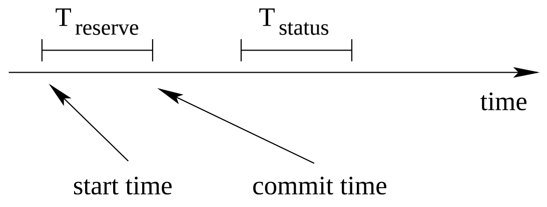

Figure 1: Transaction Execution History Example

Figure 1 illustrates a possible transaction execution history that includes *Reserve* and *Status* transactions performed on behalf of a hypothetical customer. If the database system guarantees serializable transaction executions, it is natural to assume that $T_{reserve}$ will be serialized before T_{status}, since $T_{reserve}$ finishes before T_{status} has even started. However, this is *not* the case. Serializability ensures that transactions will appear to execute serially in *some* order, not necessarily in the order in which they are submitted. Since the customer presumably expects the booking to appear in the shopping bag, serializing $T_{reserve}$ after T_{status} is a problem.

In many database systems this kind of transaction reordering cannot occur. The concurrency controls in such systems guarantee a property that has been called *strong serializability*.[3] Informally, a strongly serializable transaction history is a serializable history in which a transaction that starts after a previous transaction has finished is serialized after its predecessor. For the history illustrated in Figure 1, this means that T_{status} would be guaranteed serialization after $T_{reserve}$.

At first glance, it might appear to be unrealistic to worry about concurrency controls that guarantee serializability but not strong serializability. After all, many well-known and widely used concurrency control algorithms, including strict two-phase locking, do guarantee strong serializability. However, when data are replicated and the copies are synchronized lazily, transaction reordering can easily occur. Reordering can occur if transactions are allowed to see stale replicas. For example, if T_{status} in Figure 1 runs against a stale replica, it may "see" a database state that does not include the effects of $T_{reserve}$. Thus, in the context of replicated databases, guarantees on transaction ordering can also be thought of as guarantees on the freshness of data. Saying that T_{status} must follow $T_{reserve}$ is the same as saying

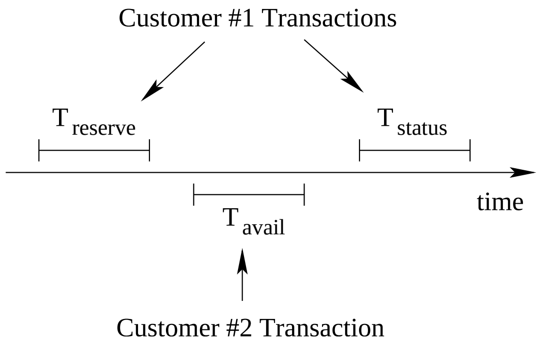

Figure 2: Execution Example with Two Customers

that T_{status} must see a database state that is at least fresh enough to include the effects of $T_{reserve}$.

Concurrency control algorithms that have been proposed for lazy replication environments do not address this problem. They guarantee serializability, but not strong serializability.[15, 5, 4] To address this concern, one could design a concurrency control that will guarantee strong serializability. The problem with this approach is that such algorithms are likely to obviate any benefits (performance, scalability) that might have been derived from replication in the first place. Once some copy of the data has been updated, replicas would effectively become useless until they too had been updated. In effect, this is turning lazy propagation into eager propagation.

In this paper, we adopt a more flexible approach in an attempt to avoid this problem. The basic idea can be explained using the example in Figure 2. This example is like the one in Figure 1, only now there are two different customers. The first customer is booking tickets, as in Figure 1. The second customer is executing T_{avail}, which simply checks ticket availability. In this example, we argue that it is important that T_{status} be serialized after $T_{reserve}$. However, the serialization order of T_{avail} and $T_{reserve}$ is *not* important, in the following sense. The first customer can observe the sequential nature of his own requests, and expects it to be preserved. However, he does not observe the arrival times of other customers' requests, and thus he has no such expectations for their ordering relative to his own. Indeed, he does not observe others' requests at all, except indirectly through their effects on the database. Similarly, the second customer is not aware of the arrival time of $T_{reserve}$ relative to his own T_{status} transaction.

Arguably, it is desirable to show every transaction the freshest possible database state. Our point is that if either T_{status} or T_{avail} has to see a stale state that does not include $T_{reserve}$, it should be T_{avail} that does so. Notice that this has nothing to do with data conflicts between the transactions - like T_{status}, T_{avail} may very well conflict with $T_{reserve}$. Nor does it have to do with transaction start and end points - both T_{avail} and T_{status} start after $T_{reserve}$ has finished.

In the example from Figure 2, neither serializability nor strong serializability is appropriate. Serializability alone is too weak, as it does not guarantee that T_{status} will appear to follow $T_{reserve}$. Strong serializability, on the other hand, is too strong. It permits one and only one serialization order: $T_{reserve}$ followed by T_{avail} followed by T_{status}. This imposes unnecessary restrictions that can hurt performance. It should be possible to place T_{avail} anywhere in the serialization order.

This paper makes the following contributions. First, we introduce a new correctness criterion, called *session level strong serializability*, or *strong session 1SR*, for concurrent transaction executions in replicated systems. Strong session 1SR generalizes both serializability and strong serializability by capturing the intuition that some transaction ordering constraints are important, and others are not. Second, we describe several techniques for implementing strong session 1SR in a simple lazy master (under Gray's classification scheme) replication architecture. Finally, we present the results of simulation studies which quantify the costs and benefits of these techniques. In particular, we identify conditions under which strong session 1SR can be implemented much more efficiently than strong serializability, and, in fact, almost as efficiently as plain serializability, which provides no ordering guarantees.

The rest of the paper is organized as follows. In Section 2, we start by presenting the lazy master architecture, and describe how regular serializability (with no ordering guarantees) can be implemented within it. In Section 3 we define strong session 1SR and relate it to existing transaction correctness criteria. Section 4 describes how the lazy master system from Section 2 can be enhanced so that it guarantees strong session 1SR. Sections 5 and 6 present the simulation model and the results of our performance evaluation, and Section 7 describes related work.

2. SYSTEM ARCHITECTURE

In this section we describe the distributed system architecture that we will consider in this paper. We describe how transactions are processed and how updates are propagated among the sites, and we show that these mechanisms guarantee one-copy serializability. We will call this the *base system*, since it represents our starting point. The base system uses well-known techniques adapted to work within the architecture we are considering. The base system guarantees global serializability. Later, in Section 4, we will show how to extend the base system so that it guarantees strong session 1SR.

Figure 3 illustrates the architecture of the base 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 site(s) 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 et al, this is a lazy master architecture.[8] Clearly, it

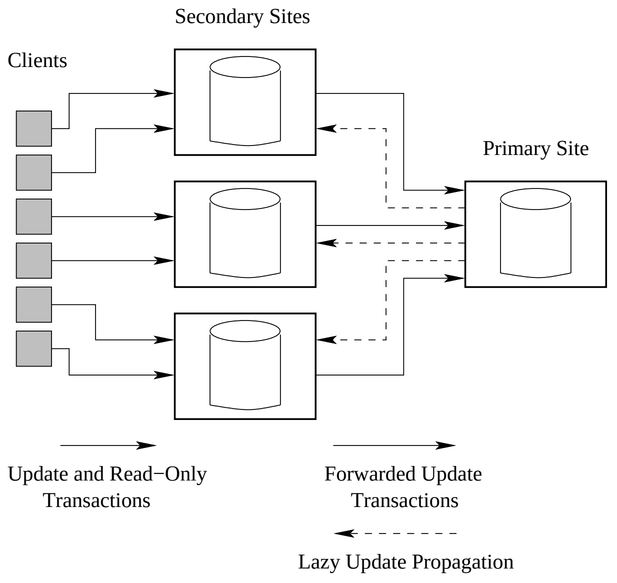
Secondary Sites
Clients
Primary Site
Update and Read−Only Transactions
Forwarded Update Transactions
Lazy Update Propagation

Figure 3: Lazy Master System Architecture

is appropriate only for read-mostly workloads since the primary site is a potential update bottleneck. However, many common workloads in application areas such as information retrieval [1], data dissemination [6] and web commerce [16], are read-mostly. The architecture has the advantage that an arbitrary number of secondary sites can be added to scale the system with increasing read-only transaction workloads.

The base 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 transactionally installed there in the same order in which they are 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.

2.1 Update Propagation

Updates made by committed transactions are propagated lazily from the primary site to the secondary sites. This means that propagation occurs some time after the update has been committed at the primary. A variety of mechanisms, including log sniffers and triggers, can be used to generate a change log for propagation. One such mechanism, more general than the one required here, is described in [14]. Propagated updates arriving at each secondary are placed in a FIFO update queue there. We will not be concerned in this paper with the exact format of the change log or the update propagation messages.

We assume that the local concurrency control at the primary site guarantees that transactions can be serialized in the order in which they commit. Many well-known concurrency control protocols, such as strict two-phase locking, have this property, which has been called *commitment ordering* (and *strongly recoverable*).[3, 13] We also assume that the propagation mechanism propagates updates in commit order, and that they are not lost or reordered by the network. Thus, the updates appear in the secondary update queues in commit order, which is the same as their serialization order.

2.2 Refresh

At each secondary site, a *refresh* process removes propagated updates from the local update queue and applies them to the local database copy. For each transaction's queued updates, the refresher uses a separate local refresh transaction to install those updates in the local secondary database copy. Thus, for every update transaction at the primary there is eventually one corresponding refresh transaction at each secondary site.

It is important that the local concurrency control at each secondary site should serialize the refresh transactions in the same order in which the corresponding update transactions were serialized at the primary site. We use a simple mechanism to achieve this: the refresh process performs the refresh transactions sequentially in the order in which updates are retrieved from the local update queue, and the local concurrency controller at the secondary is assumed to ensure that sequentially executed transactions can be serialized in the order in which they are executed. This property, which is slightly weaker than commitment ordering, has been called *strong serializability*.[3]

A drawback of this mechanism is that refresh transaction throughput is limited by sequential execution. This restriction can be relaxed, allowing refresh transactions to run in parallel. However, if refresh transactions are executed in parallel, there must be some mechanism to ensure that they will be serialized in the proper order. Several such mechanisms, including ticketing and transaction conflict analysis, have been proposed in the literature.[7, 9] Any of these techniques could be applied to parallelize the refresh transactions. However, since our global concurrency control techniques depend 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.

Finally, we note that the refresh process must contend with concurrent read-only transactions for access to the database at the secondary, and with the propagator for access to the local update queue. The refresh process uses one transaction, called a retrieval transaction, to remove updates from the queue, and a second transaction (the refresh transaction) to apply those updates to the database tables. The local concurrency control at the secondary is used to serialize both types of transactions. However, the retrieval transactions are a special case since propagation is append-only and the retrieval transactions read the updates from the queue in append order. It is sufficient for the retrieval transactions to run at a reduced SQL isolation level, namely, *read committed*.[10] The refresh transactions, however, must run at the serializable level.

2.3 Discussion

It should be clear that the database at each secondary site evolves through exactly the same sequence of states as the primary database, although it may lag behind because propagation is lazy. That is, each time a refresh transaction commits, the resulting secondary database state is the same as the primary database state when the corresponding update transaction committed. Zhuge et al called this property *completeness.*[17] Since we have completeness, it should also be easy to see that this system guarantees global, one-copy serializability (1SR).

LEMMA 1. *If the primary and all secondary database copies begin in the same state, the base system guarantees 1SR.*

PROOF. Let T_i represent the ith transaction to commit at the primary and let S_i represent the database state at the primary that results from that commit. Since the primary's local concurrency control guarantees commitment ordering, primary transactions are serialized in the order $T_1, T_2, T_3, \ldots$ and the primary database moves through the sequence of states $S_0, S_1, S_2, \ldots$ as the primary transactions commit. Now consider a secondary site. The only updates that occur at this site are made by refresh transactions. Since the propagation and refresh mechanism provides the completeness property, the database at the secondary site moves through the states $S_0, S_1, S_2, \ldots$ as refresh transactions commit. Since the local concurrency control at the secondary site serializes read-only transactions and local refresh transactions, each read-only transaction at the secondary site will see one of these states. Suppose that read-only transaction T_r sees state S_i at the secondary. This is equivalent to running T_r at the primary after T_i and before T_{i+1}. Since every read-only transaction can be fit into the update transaction serialization order in this way, 1SR is ensured. $\square$

As we have already noted, a desirable feature of the base system is that an arbitrary number of secondary sites can be added to scale the system's capacity for processing read-only transactions. Ultimately, the system's bottleneck will be the ability of the primary site to handle the updates. Note, however, that the only constraint on the primary site is that its local concurrency control must provide the commitment ordering property for update transactions. Parallelization and distribution can be used to improve the capacity of the primary site, as long as this property is maintained.

The primary disadvantage of the base system, of course, is that it does not provide the ordering guarantees that were described in Section 1. Globally, the base system provides serializability but not strong serializability. Consider again the example illustrated in Figure 1. In the base system, $T_{reserve}$, an update transaction, would be executed at the primary site, while T_{status}, a read-only transaction, would be executed at a secondary site. Since the secondary databases are stale, T_{status} may "see" a database state that does not include $T_{reserve}$, even though T_{status} is executed after $T_{reserve}$. In the remainder of this paper, we will consider how to extend the base system to correct this problem.

3. CORRECTNESS

Breitbart et al used the term *strong serializability* to mean serializability in which the ordering of non-concurrent transactions is preserved in the serialization order.[3] 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 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 in a given application, e.g., the constraint that T_{avail} must follow $T_{reserve}$ in the example of Figure 2. Furthermore, it may be very costly to implement Strong-1SR. Consider what would be involved in enforcing Strong-1SR in the base system described in the previous section, for the example from Figure 2. Since $T_{reserve}$ is an update transaction, it would be executed at the primary site. T_{avail}, which is a read-only transaction, would execute at a secondary site. However, T_{avail} may not be able to execute immediately at the secondary since Strong-1SR requires that T_{avail} see the effects of $T_{reserve}$. T_{avail} would have to wait until $T_{reserve}$'s reservation was propagated to the secondary and applied to the database there. Since the ordering of T_{avail} and $T_{reserve}$ is not important in this application, the wait is unnecessary. This difficulty is not simply an artifact of the base system. In any replicated system, Strong-1SR will have the effect of limiting the usefulness of replicas while updates are being propagated.

We propose a weaker notion of correctness that reflects the idea that ordering constraints may be necessary between some pairs of transactions, but not 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. **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 3. **Strong session 1SR:** A transaction execution history H is session-level strong one-copy serializable (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.

Note that 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, which provides no freshness guarantees.

To use Definition 3, we must specify a session labeling for the transactions. The appropriate choice depends on the

	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

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 2, 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 session 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, we can imagine that the upper tiers create session labels and pass them to the database system to inform it of the ordering constraints.

Suppose that, using such a mechanism, 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.

4. ENFORCING STRONG SESSION 1SR

In Section 2, we described a lazy master base system and showed that it guaranteed 1SR. In the base system, the global serialization order of transactions that execute at the primary site is determined by their commit order at the primary site. Each primary transaction has a corresponding refresh transaction at each secondary site, and the refresh transactions are serialized locally at the secondaries in the same order as the corresponding primary transactions. A secondary read-only transaction that is serialized locally between the refresh transactions corresponding to T_i and T_{i+1} is globally serialized between T_i and T_{i+1}.[1]

In this section, we show how to augment the base system with a global concurrency control that will guarantee strong session 1SR. We present two global concurrency control algorithms. Both are based on the same local concurrency control assumptions and the same update propagation mechanism used by the base system. Both guarantee global 1SR for the same reason the base system does (see Lemma 1). Here, we will show that the augmented system guarantees the more restrictive strong session 1SR property. To simplify the presentation, we will assume initially that individual sessions are not distributed across secondary sites. That is, all of the transactions of any given session are directed to the same secondary site.[2] At the end of this section, we will consider how to relax this assumption.

Global concurrency control is implemented by a set of global concurrency control modules, one at each secondary site. The individual modules are independent of one another, because of our assumption that each session is directed to a single site. Each module is responsible for ensuring transaction ordering within the sessions that are directed to its site.

We assume that the primary site is capable of reporting a commit sequence number for each local transaction when it commits. We will use $seq(T)$ to denote the sequence number for transaction T. These sequence numbers must have the property that $seq(T_1) < seq(T_2)$ if and only if T_1 commits before T_2 at the primary site. Since the local concurrency control at the primary site is assumed to guarantee the commitment ordering property, and since the local serialization order of primary transactions is the same as their global serialization order, this means that the sequence numbers capture the global serialization order of primary transactions. A natural choice for $seq(T)$ would be the log sequence number of T's commit log record at the primary site. However, as long as the sequence numbers are monotonically increasing with serialization order, the exact choice is not important. When a transaction T is propagated from the primary site to the secondaries, we assume that its propagation record is tagged with $seq(T)$. Again, if commit sequence numbers are log sequence numbers and if the propagator uses the log to extract updates, this is relatively simple to achieve.

The global concurrency control module at each secondary site maintains a sequence number for the database at its site, which we will refer to as $seq(DB)$. When a refresh transaction applies T at the secondary site, it sets $seq(DB)$ to $seq(T)$. Since transactions are propagated in sequence number order, and applied to the secondary databases in the order in which they arrive, $seq(DB)$ is monotonically increasing at each secondary site. (Since the secondary sites are not synchronized, each will, in general, have a different value for $seq(DB)$.)

We propose two algorithms for guaranteeing strong session 1SR within this framework. The first, called *Block*, is summarized in Figure 4. The second, called *Forward*, is summarized in Figure 5. Both algorithms maintain one sequence number for each session. Both algorithms immediately forward update transactions to the primary site for execution. However, they differ in the way they handle read-only transactions. Immediate execution of a read-only transaction at the secondary site may violate strong session 1SR if updates from a previous transaction in the session are not yet reflected in the secondary database. The Block algorithm handles this problem by delaying the read-only transaction until the secondary database reflects the previous transaction's updates. The Forward algorithm, in contrast, handles this problem by forwarding the read-only transaction to the primary site, where it can see the update immediately.

Figure 4 describes the actions that are taken by the global concurrency control under the Block algorithm. Each action describes how a global concurrency control module at a secondary site reacts when transactions from its sessions either

[1]If there are multiple read-only transactions between T_i and T_{i+1}, they can be globally serialized in any order.

[2]The secondary site may then forward some transactions to the primary.

```
At start of update transaction $T$ in session $s$:
    START $T$ AT PRIMARY SITE

At commit of update transaction $T$ in session $s$:
    COMMIT $T$ AT PRIMARY SITE, OBTAIN $seq(T)$
    $seq(s) \leftarrow seq(T)$

At start of read-only transaction $T$ in session $s$:
    $minseq(T) \leftarrow seq(s)$
    BEGIN TRANSACTION
        $d \leftarrow seq(DB)$
    END TRANSACTION
    WHILE $d < minseq(T)$ DO
        DELAY
        BEGIN TRANSACTION
            $d \leftarrow seq(DB)$
        END TRANSACTION
    END WHILE
    START $T$ AT SECONDARY SITE

At commit of read-only transaction $T$ in session $s$:
    COMMIT $T$ AT SECONDARY SITE
```

Figure 4: The Block Algorithm

```
At start of update transaction $T$ in session $s$:
    START $T$ AT PRIMARY SITE

At commit of update transaction $T$ in session $s$:
    COMMIT $T$ AT PRIMARY SITE, OBTAIN $seq(T)$
    $seq(s) \leftarrow seq(T)$

At start of read-only transaction $T$ in session $s$:
    BEGIN TRANSACTION
        $d \leftarrow seq(DB)$
    END TRANSACTION
    IF $d < seq(s)$ THEN
        $site(T) \leftarrow$ "PRIMARY"
        START $T$ AT PRIMARY SITE
    ELSE
        $site(T) \leftarrow$ "SECONDARY"
        START $T$ AT SECONDARY SITE

At commit of read-only transaction $T$ in session $s$:
    IF $site(T) =$ "PRIMARY" THEN
        COMMIT $T$ AT PRIMARY SITE, OBTAIN $seq(T)$
        $seq(s) \leftarrow seq(T)$
    ELSE
        COMMIT $T$ AT SECONDARY SITE
```

Figure 5: The Forward Algorithm

start or commit.[3] In particular, the initiation of a read-only transaction is blocked until the database sequence number is at least as great as that transaction's session's sequence number, which represents the serialization order of the most recently committed update transaction in the session. This ensures that the read-only transaction will "see" the update transactions that precede it in its session. In Figure 4, this check is implemented by polling the database sequence number ($seq(DB)$). Other, more efficient implementations are also possible. In particular, blocked transactions could be signaled once the refresh transactions have advanced the database sequence number far enough.

In both algorithms, each concurrency control action must be executed atomically with respect to other events from the same session. An exception is that while a read-only transaction is waiting for $seq(DB)$ to advance, other events may occur. Another issue is the contention between the concurrency control actions, which read $seq(DB)$, and refresh transactions, which update it. When the global concurrency control module needs to read $seq(DB)$ it uses a short auxiliary transaction to do so. (These transactions are delimited in Figures 4 and 5 by BEGIN TRANSACTION and END TRANSACTION.) The auxiliary transactions are serialized with other transactions at the secondary site by the local concurrency control.

Figure 5 describe the concurrency control actions taken under the Forward algorithm. New read-only transactions are executed locally if the local database state ($seq(DB)$) is recent enough. Otherwise, they are forwarded to the primary. When a read-only transaction is forwarded to the primary site, it obtains a sequence number in the same manner as an update transaction would.

Both the Block algorithm and the Forward algorithm guarantee Strong Session 1SR.

THEOREM 1. *If each transaction session is directed to a single secondary site, the local concurrency controls ensure commitment ordering (primary site) or strong serializability (secondary sites), and all database copies start in the same state, then the Block algorithm 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:

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 commitment ordering 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

[3]No actions are required when a transaction aborts.

set to $seq(T_1)$. $seq(s)$ is monotonically increasing over time, since it is set by the commit of update transactions at the primary. Therefore, when T_2 attempts to start, $seq(s) \geq seq(T_1)$. T_2 will be blocked until an auxiliary transaction finds $seq(DB) \geq seq(s)$, which implies $seq(DB) \geq seq(T_1)$. Since T_2 runs after the auxiliary transaction(s) and the local concurrency control ensures strong serializability, T_2 also sees the database in at least state $seq(T_1)$. Thus, it is globally serialized after T_1, 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.

□

THEOREM 2. *If each transaction session is directed to a single secondary site, the local concurrency controls ensure commitment ordering (primary site) or strong serializability (secondary sites), and all database copies start in the same state, then the Forward algorithm 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:

Case 1: Suppose T_1 and T_2 both execute at the primary. This is the same as Case 1 in Theorem 1.

Case 2: Suppose T_1 executes at the secondary and T_2 executes at the primary. This is the same as Case 2 in Theorem 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 $seq(T_1)$. $seq(s)$ is monotonically increasing over time, since it is set by the commit of update transactions at the primary. Therefore, when T_2 attempts to start, $seq(s) \geq seq(T_1)$. At this point, an auxiliary transaction reads $seq(DB)$. If $seq(DB) < seq(s)$, T_2 is run at the primary. Since the primary ensures commitment ordering and T_2 follows T_1, T_1 is serialized first, a contradiction. If, instead $seq(DB) \geq seq(s)$, then we have $seq(DB) \geq seq(s) \geq seq(T_1)$. Since T_2 runs after the auxiliary transaction, and the local concurrency control ensures strong serializability, T_2 sees the secondary database in at least state $seq(T_1)$. Thus, it is serialized after T_1, a contradiction.

Case 4: Suppose T_1 and T_2 both execute at the secondary. This is the same as Case 4 in Theorem 1.

□

The Block and Forward algorithms were presented under the assumption that each session is the responsibility of a single secondary site. Relaxing this assumption introduces two complications which are not addressed in Figures 4 and 5. First, the session sequence number $seq(s)$ must be replicated across all secondary sites to which session s may direct transactions. The replicated sequence numbers must be relatively tightly synchronized, so that the global concurrency control actions for each new transaction on s are taken using the latest value of $seq(s)$.

Parameter	Description	Default
num_clients	number of clients	varies
num_sec	number of secondary sites	5
think_time	mean client think time	7 sec.
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	0.001s

Table 2: Simulation Model Parameters

The second problem is more subtle. Suppose that T_1 and T_2 are read-only transactions in the same session, and T_2 follows T_1. Strong session 1SR requires that T_2 be serializable after T_1. If T_2 and T_1 are directed to *different* secondary sites, the global concurrency control must ensure that T_2 sees at least as current a database state at its site as T_1 sees at its site. Thus, a limited form of secondary site synchronization is required. This can be accomplished by updating $seq(s)$ after each read-only transaction. The value assigned to $seq(s)$ must be at least as large as the value of $seq(DB)$ seen by the read-only transaction. This can be accomplished using an auxiliary post-transaction (run after the read-only transaction) that reads $seq(DB)$ and updates $seq(s)$.

5. SIMULATION MODEL

We have developed a simulation model of the lazy master replicated database system described in Section 2. We have used the simulator to compare the Block and Forward global concurrency control protocols described in Section 4 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.[11]

The model consists of one resource for each site in the system, and a set of processes that use those resources. For each client there is a transaction execution process that simulates the execution of transactions requested by that client. Each client is associated with a particular secondary site, and it submits all of its transactions to that site. The clients 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.

The transaction execution model used by each client is shown in Figure 6. Each client iteratively generates transactions, with exponentially distributed think times averaging *think_time* between them. A transaction is an update transaction with probability *update_tran_prob*, otherwise it

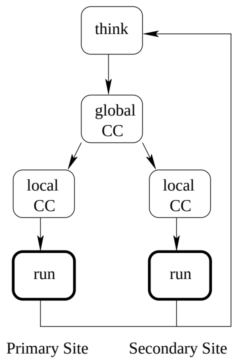

Figure 6: Logical Transaction Execution Model

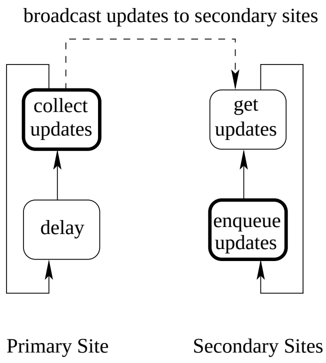

Figure 7: Update Propagation Model

is a read-only transaction. Our default transaction mix is 80% read-only, 20% update. We also ran some experiments with a 95%/5% mix. (For comparison, 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 95/5. However, there is not necessarily a one-to-one correspondence between web interactions and database transactions.)

The sequence of transactions generated by a client is considered to form a session for the purposes of enforcing Strong Session 1SR. Periodically, each client ends it current session and immediately begins a new one. Session lengths are exponentially distributed with a mean length of *session_time*. The default *think_time* and *session_time* values are taken from the TPC-W benchmark specification.[16]

Figure 6 illustrates the transaction execution model. In Figure 6 (and also in Figures 7 and 8), steps shown in bold consume time at either the primary resource or at a secondary resource. The remaining steps may introduce delays, but they do not consume resources. As shown in Figure 6, each transaction proceeds first to the global concurrency control. The global concurrency control directs each transaction to either the primary site or the secondary site for execution. If the global concurrency control is the Block algorithm, it may also cause the transaction to block.

After global concurrency control, 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. The total service time for a transaction is the number of operations it has times *op_service_time*. 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 of *update_op_prob* of being an update, otherwise it is a read. All transactions running at a given site share a single server, which uses a round-robin queuing discipline with a time slice of 0.001 seconds.

The update propagation model is illustrated in Figure 7. At the primary site, a propagation process cycles between propagating updates and pausing. The duration of each pause is *propagation_delay*. During each propagation cycle, the propagator propagates all transactions that have committed at the primary since the last propagation cycle. For each committed transaction, the propagator enqueues a record in an update queue at each secondary site. The record indicates the number of update operations that were performed by the committed update transaction. This batch of records is then broadcast to propagation processes at the secondary sites. The primary's propagation process consumes *op_service_time* during each propagation cycle. As can be seen from Figure 7, the propagator does not use the local concurrency control at the primary site. This is because we assume that the propagator is implemented as a log sniffer. Finally, note that the simulation does not include a resource to represent the network. We assume that the network has sufficient capacity so that network contention is not significant. As for latency, we use the propagator's *propagation_delay* parameter to model propagation latencies from all sources, including the network.

Each secondary site also includes a propagation process, which reads batches of propagation records broadcast by the primary and installs them in an update queue in the local database. These processes consume *op_service_time* at their secondary site for each batch of update operations that they install. The secondary propagators do not use the local concurrency control when they install their updates. These operations conflict only with the refresh processes and, as discussed in Section 2.2, the conflict is at a reduced isolation level. Thus, we assume that any contention would be insignificant.

At each secondary site there is a set of refresh processes. The execution model for refresh processes is illustrated in Figure 8. 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 ensures that refresh transactions are not reordered by the local concurrency control. As discussed in Section 2, this is needed to ensure complete-

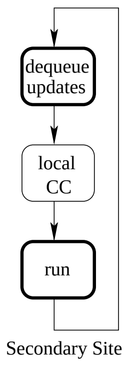

Figure 8: Refresh Model

ness at the secondary sites. 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* times the number of updates to run the refresh transaction itself.

6. 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 global concurrency controls in addition to Block and Forward.

Alg-1SR: Alg-1SR is the global concurrency control used in the base system that was described in Section 2. It provides only global serializability (1SR), not strong session 1SR. Alg-1SR simply routes all update transactions to the primary site, and 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.

Alg-Strong-1SR: Alg-Strong-1SR is the same as the Block algorithm of Figure 4 except that there is only a single session per secondary site, rather than one session per client. Thus, Alg-Strong-1SR maintains only one session sequence number ($seq(s)$) at each secondary site. Alg-Strong-1SR does *not* guarantee strong 1SR, since that would require a single session for the entire system, rather than one per secondary site. It enforces many more transaction ordering constraints than must be enforced by Block and Forward, 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-1SR should result in performance no worse than (and probably significantly better than) that of any algorithm that provides true strong 1SR.

6.1 Methodology

For each run, the simulation parameters are set to the default values found in Table 2, except as indicated in the descriptions of the individual experiments. Each run lasted

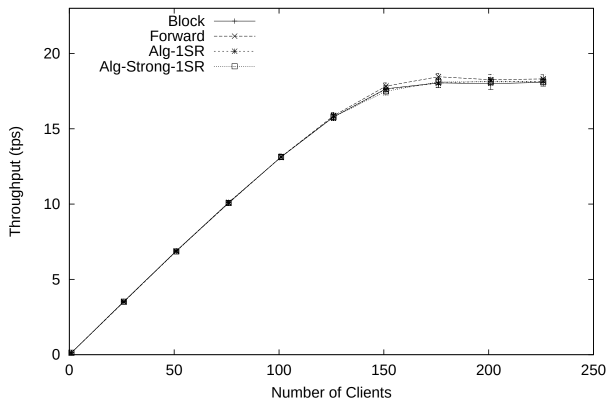

Figure 9: Transaction Throughput vs. Number of Clients, Default Parameters

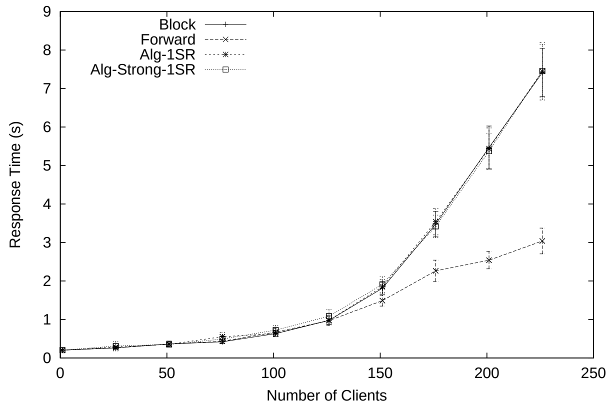

Figure 10: Read-Only Transaction Response Time vs. Number of Clients, Default Parameters

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 5 independent runs. We computed 95% confidence intervals around these means. These are shown as error bars in the graphs.

6.2 Default Con£guration

We ran an initial series of experiments in which our default configuration, with five secondary sites, was subjected to load from an increasing number of clients. Figures 9 and 10 show the results of this experiment. Each curve describes the behavior of one of the four global concurrency control algorithms (Alg-1SR, Block, Forward and Alg-Strong-1SR) that we considered.

The most important property of our default configuration is that the *propagation_delay* is very small. This makes it possible to keep the secondary databases very fresh. Under these conditions, the Block and Forward algorithms have nearly identical throughput, and both algorithms are indistinguishable from Alg-1SR, which does not guarantee strong session 1SR. Thus, strong session 1SR comes almost for free. However, Alg-Strong-1SR, which enforces unneces-

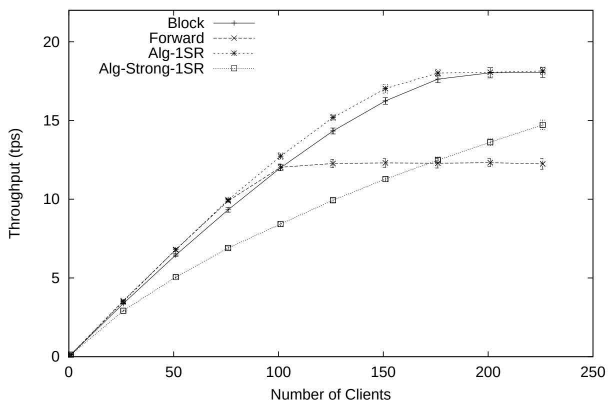

Figure 11: Transaction Throughput vs. Number of Clients, *propagation_delay* $= 10$

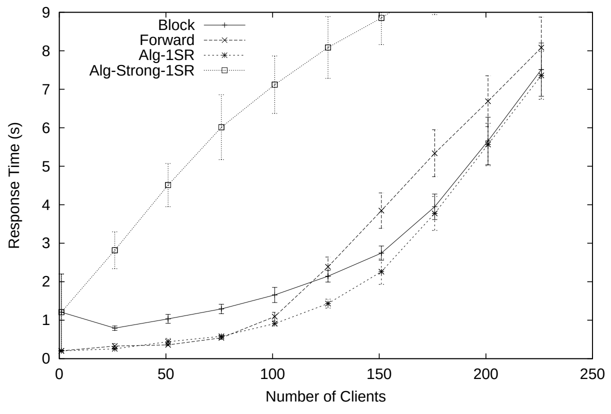

Figure 12: Read-Only Transaction Response Time vs. Number of Clients, *propagation_delay* $= 10$

sarily strong ordering constraints, also performs as well as strong session 1SR (Block and Forward) and Alg-1SR. All of the algorithms perform equally well because read-only transactions are rarely delayed or forwarded to the primary site. This, in turn, is because the secondary databases do not lag far behind the primary.

Under very high load conditions ($num\_clients > 150$), Figure 10 shows that the Forward algorithm results in lower response times for read-only transactions than does the Block algorithm, and even Alg-1SR. With 150 clients, utilization at the secondary sites is approximately 95% under Alg-1SR and Block, causing read-only response times to grow. However, the utilization of the primary is slightly lower. Since the Forward algorithm deflects load (read-only transactions) from the secondary sites to the primary, it reduces resource contention at the secondary sites, resulting in improved read response times. This load balancing effect is short-lived, however, since the primary site quickly becomes over-utilized. Furthermore, as might be expected, response times of update transactions (not shown) are significantly higher under Forward than under the other algorithms because of the extra load it places on the primary site.

6.3 Propagation Latency

In the default configuration, updates are propagated from the primary site almost as soon as possible after they occur. In practice, however, this may be difficult to achieve. Scheduling at the primary site, network latencies, and batching may introduce propagation latencies. We model this using the *propagation_delay* parameter.

A nice feature of strong session 1SR is that it should allow a certain amount of 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. To explore this effect, we ran experiments with *propagation_delay* set to 10 seconds. Because the propagator sends a batch of updates after each propagation delay, this has the effect of introducing a delay of between zero and ten seconds for each individual transaction's propagation, or about 5 seconds on average. The mean client inter-transaction think time remained fixed at its default value of 7 seconds. The results are shown in Figures 11 and 12.

With propagation delay, the results look significantly different than they did under the default configuration. The Block algorithm suffers a only small throughput penalty as compared to Alg-1SR, right up into high load conditions. It also suffers a response time penalty (compared to Alg-1SR) which is most significant under low loads. However, overall the performance degradation is small considering that Block enforces strong session 1SR and Alg-1SR does not.

At low loads, the Forward algorithm provides better read response times than Block, and approximately the same overall throughput as Alg-1SR. However, as the load increases, Forward breaks down because it saturates the primary site. The difference between Forward's performance with low propagation delay (Figure 9) and its performance with high propagation delay (Figure 11) is that it deflects more read-only transactions to the primary site when propagation delays are higher. This quickly turns the primary site into a bottleneck.

The performance of Alg-Strong-1SR is very poor under these conditions because many read-only transactions are delayed at the secondary sites. A comparison of the performance of Alg-Strong-1SR with that of Block and Alg-1SR shows that an overly restrictive correctness criterion can significantly impact performance.

In summary, the Block algorithm is effective at using the flexibility provided by strong session 1SR to mask much of the effect of the propagation latency.

6.4 Scalability

A desirable feature of the base system on which the global concurrency controls are implemented is that the number of secondary sites can be scaled with the client load. To examine this, 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. *propagation_latency* remained at 10 seconds. Figures 13 and 14 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-1SR. Throughput for both Block and Alg-1SR eventually peaks (with about 12 secondary sites) when the primary site saturates. Since the primary site is the bottleneck that eventually limits throughput, the key scalability pa-

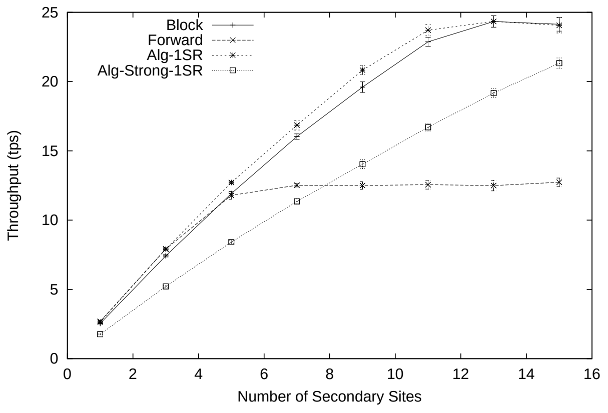

Figure 13: Transaction Throughput, 20 Clients per Secondary, *propagation_delay* $= 10$

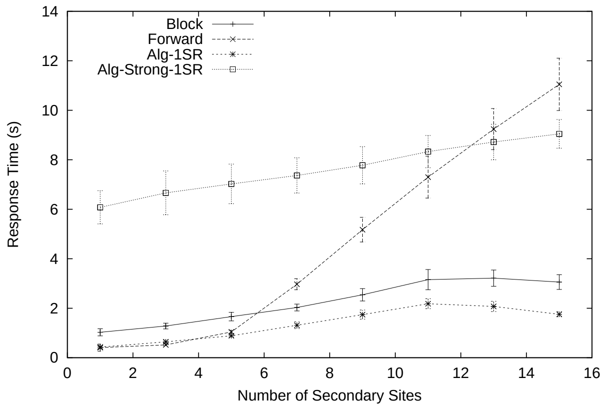

Figure 14: Read-Only Transaction Response Time, 20 Clients per Secondary, *propagation_delay* $= 10$

rameter is the mix of read-only and update transactions in the workload. As might be expected, similar experiments with a 95/5 transaction mix (not shown here) showed much greater scalability than the 80/20 mix shown here.

The Forward algorithm does not scale as well as Block, again because of the extra load it places on the primary site.

7. RELATED WORK

The techniques described in this paper are global, multiversion concurrency controls implemented in a federated, replicated database system. Gray et al provide a relatively recent discussion of synchronization in replicated database systems and the challenges that that poses.[8] According to Gray's classification, the system described in this paper has a lazy master architecture. 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 base system (Section 2) 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.

A general overview of multiversion serializability theory, including a definition of 1SR, and multiversion concurrency control algorithms is provided by Bernstein, Hadzilacos and Goodman.[2] The multiversion mixed method they describe distinguishes between read-only and update transactions. It serializes update transactions using locking, and allows read-only transactions to see stale versions of the data. Of the concurrency controls they describe, it is closest to the technique presented here. However, those algorithms ignore issues of distribution and do not explicitly consider the impact of data freshness. Read-only transactions are shown the latest committed state.

Some recent work has considered the specific problem of concurrency control for lazy master replicated database systems.[5, 4] Unless care is taken in how updates are propagated and applied, lazy master architectures do not automatically guarantee 1SR.[8] Breitbart et al proposed 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 work on concurrency control protocols in lazy master systems includes the virtual sites protocol of Breitbart and Korth, and the quorum consensus protocol of Satyanarayanan and Agrawal, which uses a gossip mechanism to lazily propagate updates to sites that have missed them.[5, 15] None of these protocols consider data freshness, and none have a notion of transaction sessions. All guarantee 1SR, but not strong session 1SR or strong 1SR.

Pacitti and Simon have studied the effects of different update propagation techniques on the freshness of replicated data.[12] They considered whether the primary site should begin propagating updates before the updating transaction commits, or whether propagation should instead wait for commit. If propagation begins early, they also considered whether the application of propagated updates (refresh) should or should not wait for the original update transaction to commit at the primary. The approach we have assumed here, in which propagation occurs after commit, is called *deferred-immediate* by Pacitti and Simon. These propagation issues are essentially orthogonal to the techniques we have considered here. It should be possible to combine our global concurrency controls with any one of these propagation options. The goal of Pacitti and Simon's 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 et al considered techniques for maintaining a replicated database to be used as backup in case the primary copy fails.[9] According to Gray's classification, their approach is lazy master. King et al call their techniques *1-safe*, which emphasizes the fact that between the time an update commits at the primary and the time the corresponding update is applied at the secondary, a failure may cause the update to be lost. In their work, King et al 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. King et al propose to use 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. King et al propagate the read sets to the secondary site for analysis there. Alternatively, the analysis could be performed at the primary site.

8. CONCLUSION

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

Starting with a base 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 simple 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 developed a simulation model of the lazily synchronized system, and used it to study the impact of the new correctness criterion and the performance of the Block and Forward algorithms. We found that when propagation latencies are low, that is, when the secondary database copies can be kept very fresh, ensuring strong session 1SR costs very little in terms of transaction throughput and response time. However, even strong 1SR costs very little, so strong session 1SR does not provide any significant benefits. However, as the propagation latencies increase, strong 1SR becomes very difficult to achieve, while strong session 1SR can be maintained with only a small penalty relative to 1SR. Of the two algorithms, Block has consistent performance at all load levels. The Forward algorithm performs somewhat better at low load, but deteriorates rapidly as the load increases. We conclude that strong session 1SR appears to be a useful and practical way of capturing data freshness and transaction ordering constraints in scalable replicated database systems.

9. REFERENCES

[1] R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, 1990.

[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] P. Chrysanthis and E. Pitoura. Scalable processing of read-only transactions in broadcast push. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 432–439, 1999.

[7] 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.

[8] 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.

[9] 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.

[10] J. Melton and A. Simon. *SQL:1999 Understanding Relational Language Components.* Morgan Kaufmann, 2002.

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

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

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

[14] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How to roll a join: Asynchronous incremental view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 129–140, 2000.

[15] 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.

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

[17] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 316–327, May 1995.