Technical Report TRCS00-01
Database Replication Using Epidemic Update [*]

JoAnne Holliday Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106
USA
{joanne46,agrawal,amr}@cs.ucsb.edu

Abstract

Due to severe performance penalties associated with synchronous replication, there is an increasing interest in asynchronous replica management protocols in which database transactions are executed locally, and the effects of these transactions are incorporated asynchronously on remote database copies. However, the asynchronous protocols currently in use either do not guarantee consistency and serializability as needed by transactional semantics or they impose restrictions on placement of data and on which data objects can be updated. In this paper we investigate an epidemic update protocol that guarantees consistency and serializability in spite of a write-anywhere capability. We conducted experiments on a detailed simulation of a distributed, replicated database to evaluate this protocol. Our results establish that this epidemic approach is indeed a viable alternative to traditional eager update protocols for a distributed database environment where consistency and full serializability are needed.

1 Introduction

Data replication in distributed databases is an important problem that has been investigated extensively for more than a decade. In spite of numerous proposals, the solution to efficient access of replicated data remains elusive. Data replication has long been touted as a technique for improved performance and high reliability in distributed databases. Unfortunately, data replication has not delivered on its promise due to the complexity of maintaining integrity and consistency of replicated data. Traditional approaches for replica management incur significant performance penalties. In a recent paper entitled "The Dangers of Replication and a Solution" [7], Gray, Helland, O'Neil, and Shasha have argued that the conventional approach to replica management with write-anywhere capability and full serializability is impractical if not impossible.

Many replica management techniques have been proposed. The traditional approach requires the synchronous execution of the individual read and write operations to be executed on some set of the copies *before* transaction commit. For example, a read-one write-all locking protocol would execute each read operation by acquiring a read lock, while write operations are executed by obtaining write locks on all copies. After all the operations are executed, on all sites, the transaction is committed using two phase commit. Similarly, the quorum locking approach [6, 19] requires reads to obtain locks on a read quorum of copies and writes to obtain locks on a write quorum of copies.

An alternative approach that was first promoted by the networks and distributed operating systems community is to execute operations locally without synchronization with other sites, and after termination, the

[*] This work was partially supported by NSF grant CCR97-12108.

updates are propagated to other copy sites [20, 14, 17]. In this approach, changes are propagated throughout the network using an epidemic approach [17], where updates are piggy-backed on messages, thus ensuring that eventually all updates are propagated throughout the system.

The epidemic approach or its variant, referred to as asynchronous logging, has been adopted by several database companies like Oracle, Sybase and Informix. However, when used for multi-operation transactions, these techniques are not safe and do not ensure serializability. To overcome this problem, Anderson et al. [3] and Breitbart et al. [5] impose a graph structure on the sites and classify copies into primary and secondary copies, thus restricting how and when transactions can update copies of data objects. Rabinovich et al. [18] have proposed an algorithm for managing replicated data asynchronously. The algorithm provides for eventual consistency and can allow non-serializable executions when used with transactional semantics. However, they propose an interesting method of reducing overhead due to the necessary version comparisons. Recently, we have developed a hybrid approach where a transaction executes its operations locally, and then before committing uses epidemic communication to propagate all its updates to all replicas. Once a site is sure that the updates have been incorporated at all copies, the transaction is committed. This approach ensures serializability without imposing access restrictions on transactions, while exploiting some of the advantages of epidemic propagation, namely the asynchronous propagation and execution of update operations throughout the system.

In this paper we explore the potential benefits of epidemic communication for replica management systems that ensure serializability. We use a detailed database simulation to evaluate the performance of the proposed protocol. The results indicate that, although a transaction cannot commit until it is certain that its updates have been propagated to all other sites, this approach is a viable solution to the replication problem in many distributed environments. With the increasing reliance on Web-based computing and E-commerce, data replication and replicated database servers will become important for designing highly fault-tolerant and scalable servers. It is therefore very important to develop and explore new strategies for supporting replicated data efficiently. Our work in this paper represents efforts in that direction.

The paper is organized as follows. Section 2 defines our system model and Section 3 discusses the specific replicated data update protocols used. Section 4 describes our simulation program. In Section 5 we present the results of our experiments. The paper concludes with a discussion in Section 6.

2 System and Communication Model

In this paper we consider a distributed system consisting of a number of database server sites each maintaining a copy of all the items in the database. We refer to these sites as *server copies*. The copies are all equal copies, that is, there is no *primary* copy which must serialize the writes of a particular data item while the other copies are only *secondary*. Thus, an update can be done at any site. The sites are connected by an efficient point-to-point network. The sites are fail-stop and the communication network is not required to be reliable. Messages can arrive in any order, take an unbounded amount of time to arrive, or may be lost entirely. However, messages will not arrive corrupted. In order to accomplish the epidemic update, a site contacts another site and sends it information that it knows about the system; thus the information is propagated in a transitive manner.

A transaction is a sequence of read and write operations on one or more data items. A transaction can originate at any site, and that site becomes the initiating or home site. Since strict two-phase locking [9] is widely used, we assume that concurrency control is locally enforced by the strict two phase locking protocol at all server copy sites.

An event model [13] is used to describe the system execution, $\langle E, \rightarrow \rangle$, where E is a set of operations and $\rightarrow$ is the *happened-before* relation which is a partial order on all operations in E. The happened-before relation is the transitive closure of the following two conditions:

- Local Ordering Condition: Events occurring at the same site are totally ordered.
- Global Ordering Condition: Let e be a send event and f be the corresponding receive event then $e \rightarrow f$.

This model preserves the potential causality among operations captured by the happened-before relation. Minimally, if two operations are causally ordered their effects should be applied in that order at all sites. Lamport uses the happened-before relation to define a clock with the following property [13]:

$$\forall e, f \in E \text{ if } e \rightarrow f \ \text{ then } \ Time(e) < Time(f)$$

We will refer to this clock as Lamport's clock. Vector clocks [15] are an extension of Lamport clocks and ensure the following property:

$$\forall e, f \in E \ e \rightarrow f \ \text{ iff } \ Time(e) < Time(f).$$

Vector clocks can detect all three possible relationships between two events e and f. Either $Time(e) < Time(f)$ in which case $e \rightarrow f$, or $Time(f) < Time(e)$ in which case $f \rightarrow e$, or $Time(e)$ and $Time(f)$ are incomparable, denoted $Time(e) <> Time(f)$, in which case e and f are concurrent.

In addition to vector clocks, each site maintains a *communication* or *event* log of transaction operations. This log is not the same as the *database recovery* log as it is used solely for epidemic communication purposes [9]. Sites exchange their respective communication logs to keep each other informed about the operations that have occurred in the system. This information exchange ensures that eventually all sites incorporate all the operations that have occurred in the system. Due to the unreliable nature of the communication medium, a record must be included in every message until the sender knows that the recipient of the message has received that record.

Wuu and Bernstein [20] combine communication logs and vector clocks to solve the distributed dictionary problem efficiently. Each site S_i keeps a two-dimensional time-table T_i, which corresponds to S_i's most recent knowledge of the vector clocks at all sites. Each time-table ensures the following *time-table property*: if $T_i[k, j] = v$ then S_i knows that S_k has received the records of all events at S_j up to time v (which is the value of S_j's local clock). To reduce communication it is desirable for a site to know which sites have received the record of a particular event. To this end a predicate $HasRecvd$, is defined as follows [20]:

$$HasRecvd(T_i, t, S_k) \equiv T_i[k, Site(t)] \geq Time(t)$$

where t is an event, $Site(t)$ is the site at which t occurred, and $Time(t)$ which is $T_{Site(t)}[Site(t), Site(t)]$, is the local time at $Site(t)$ when t occurred. When $HasRecvd(T_i, t, S_k)$ is true S_i knows that S_k has received a record of event t. When a site S_i performs an update operation it places an event record in the log recording that operation. When S_i sends a message to S_k it includes all records t such that $HasRecvd(T_i, t, S_k)$ is false, and it also includes its time-table T_i. When S_i receives a message from S_k it applies the updates of all received log records and updates its time-table in an atomic step to reflect the new information received from S_k. When a site receives a log record it knows that the log records of all causally preceding events either were received in previous messages, or are included in the same message. This is referred to as the *log property* which is stated as follows with respect to a local copy of the log L_i at site S_i:

$$\forall e, f \in E \text{ if } \ (e \rightarrow f) \wedge (f \in L_i) \ \text{ then } \ e \in L_i.$$

The protocol ensures that whenever a site is aware of an event (insert or delete in the dictionary), it is aware of all causally proceeding events.

3 Epidemic Update Protocols

Epidemic protocols were originally developed for single update operations [20, 14, 17]. We now describe an epidemic update protocol [1] for supporting multi-operation transactional semantics [9].

When a transaction, t, successfully completes its operations at the home site, S_i, using a local concurrency control method such as two phase locking, it *pre-commits*. If the transaction is read-only, it can be committed at that time and no further processing is needed. Otherwise, a pre-commit record containing the readset ($RS(t)$), writeset ($WS(t)$), the values written, and a pre-commit timestamp ($TS(t)$) from the home site's vector clock is written to the local communication log and the read-locks held by the transaction are released. The pre-commit timestamp is the i^{th} row of the S_i's time-table, i.e., $T_i[i, *]$, with the i^{th} component incremented by one. This timestamp assignment ensures that t dominates all those transactions that have already pre-committed on S_i regardless of where they were initiated. Note that at this point there is still the possibility that the transaction will be aborted due to conflicts with other pre-committed transactions. To enforce serializability, if two transactions conflict, they are both aborted.

When a site S_i contacts site S_k to initiate an epidemic transfer, S_i determines which of its log records have not been received by S_k. All transaction records t such that $HasRecvd(T_i, t, S_k)$ is false (those records which S_i believes are unknown to site S_k) are sent in a message along with S_i's time table T_i. When S_k receives the message, it reads the transaction records in order and determines if there is any conflict with transactions already in S_k's log that have not yet committed and updates its time table with the information from S_i.

Two operations conflict if they are concurrent, they operate on the same data item, they originate from different transactions and at least one of them is a write operation. The vector timestamps given to pre-committed transactions can be used to determine concurrency and the read and write sets in the log records determine conflicts. Thus, two transactions t and t' conflict if the following conditions hold:

- $TS(t')$ is incomparable to $TS(t)$ (denoted $TS(t') <> TS(t)$). which is to say that t' executed concurrently with t.

- The intersections of $RS(t)$ and $WS(t')$, $WS(t)$ and $WS(t')$, or $WS(t)$ and $RS(t')$ is nonempty. This indicates that t and t' execute conflicting operations.

If there are any conflicts, both transactions involved in the conflict are aborted by releasing any locks they hold and marking the pre-commit record in the log as aborted or inserting such a pre-commit record if there is not one already. An aborted transaction is retained in the log and sent to other sites until it is known (via the time table) that all sites have knowledge of that transaction's termination.

If the transaction t whose record was sent from S_i to S_k is not aborted, it is executed at S_k by obtaining write locks and incorporating the updates to the local copy of the database. If there are local transactions that have not yet pre-committed that hold conflicting locks, they are aborted and t is granted the locks.

A transaction is committed and the remainder of its locks released when it is not aborted and it is known that all sites have knowledge of that transaction. That is,

$$Commit(t, S_i) \equiv \left[\begin{array}{c} \forall k \; T_i[k, Site(t)] \geq TS(t) \\ \bigwedge \\ \neg Aborted(t, S_i) \end{array} \right]$$

This protocol ensures serializability as was shown in [1]. The protocol also tolerates site crashes and network partitioning, since the information is stored in the log and remains there until the failures are repaired, and the information transmitted to all relevant sites. The viability of the asynchronous propagation approach for replicated information has been demonstrated in maintaining global control information in distributed systems. The main issue remaining is whether the asynchronous approach will work for replicated databases that require strong consistency and multiple operation atomicity. That is, can it be used as an alternative to the traditional approach that requires a transaction to acquire locks at all replica sites, receive acknowledgments for each operation, and end with an atomic commit protocol such as Two Phase Commit? In the rest of this paper, we analyze the performance of this epidemic protocol using a computer simulation and we demonstrate that for a variety of environments, it in fact is a feasible approach for maintaining data asynchronously while preserving one-copy serializability.

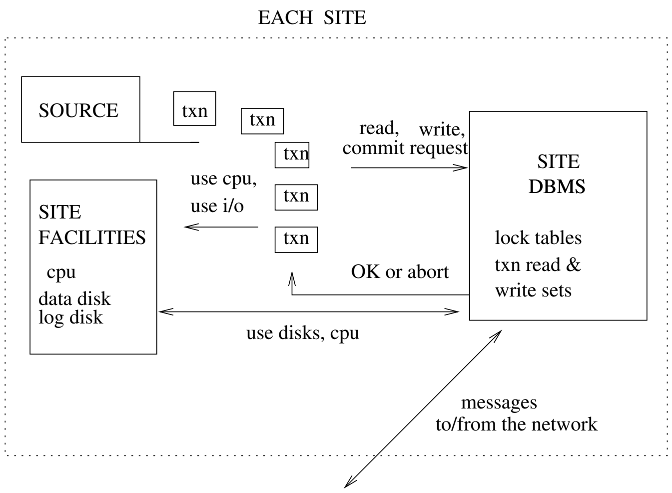


Figure 1: Model Structure of a Single Site

4 The Simulation Model

Our simulation is based on standard database simulation models [12, 10] where each site is modeled as shown in Figure 1. Each site has a *Source* thread that generates transactions. The transactions make read and write requests to the *Site DBMS* thread which maintains the database itself, enforces two-phase locking, maintains the time-table and the communication log and initiates epidemic messages to other sites. Resources that are used for a given time by the transactions and the Site DBMS are: the CPU, the database disk, the recovery log disk, and the network. Transactions are modeled as sequences of read and write operations. All the reads are done before all the writes and the writeset is a subset of the readset. The time between successive operation requests within a transaction is a parameter, IntTime, set at 10 ms.

Each site has a copy of the 1000 page database. By making the number of data pages small, we can study data contention issues more easily. These 1000 pages could represent the "hot spots" of a larger database. For this study we consider complete replication. This assumption could easily be relaxed to reflect copies on only a subset of the sites. A page is assumed to be 2Kbytes and is the locking granularity as well as the data disk granularity. The DBMS uses strict two-phase locking to ensure local serializability. A *wait-for-graph* is maintained and is checked for cycles each time an edge is added. The deadlocked transaction with the highest transaction ID is chosen as the victim. This handles all single site deadlocks and the epidemic protocol we are using guarantees there will be no global deadlocks [1].

The system parameters of the model are given in Table 1 along with their values. The parameters governing the generation and behavior of transactions are given in Table 2. The generation of new transactions is governed by the ThinkTime. This is the per site transaction interarrival time and can be made arbitrarily

Parameter Name	*Meaning*	Value
ThinkTime	Transaction interarrival time	varies
NumSites	Number of sites	varies
ER	Epidemic rate	varies
NumDisks	Number of data disks per site	1
MinDiskTime	Smallest data disk access time	4 ms
MaxDiskTime	Maximum data disk access time	14 ms
CPUInitDisk	CPU time needed to access disk	0.4 ms
LogDiskTime	Time needed for forced write of log	10 ms
LogPageSize	Number of log records per page	100
HitRate	Probability of cache hit	0.8
LockTime	CPU time to handle data item locks	0.006 ms

Table 1: System Parameters

Parameter Name	*Meaning*	Value
ReadSetMin	Smallest transaction read set size	5
ReadSetMax	Maximum read set size	11
ROPercent	Percentage of read only transactions	75% (50%,90%)
WriteSetMin	Smallest write set for update transactions	1
WriteSetMax	Maximum write set size	4
CPUPageTime	CPU time spent on a data page	1.0 ms
IntTime	Time between successive operation requests	10 ms

Table 2: Transaction Parameters

long or short. This corresponds to the "open" queuing model and is appropriate for a system accepting requests from an unbounded number of users. For most of the experiments, we set the percentage of read-only transactions to 75%. This is a reasonable assumption since in most database applications, most transactions are queries. The readset size is between 7 and 11 read operations. Update transactions have a readset size between 5 and 8 read operations and a writeset size of 1 to 4 write operations. The writeset is a subset of the readset so there are no *blind writes*.

The simulation model assumes that the network is fully connected (unless otherwise specified) so that any site can exchange messages directly with any other site. We assume that the network is fast (100 Mbits/sec) and efficient. On a 100 M bit/sec network, the amount of CPU time needed to send or receive a message was set to .09 ms. When a site is ready to initiate an epidemic session with another site, it chooses a receiver site at random from the remaining sites. The epidemic rate determines how often a site may initiate an epidemic update.

5 Experimental Results

In this section, we present the results of a variety of experiments that were conducted. The overall goals of the experiments were:

1. To evaluate the performance of the epidemic protocol as the system load is varied.

2. To evaluate the performance of the epidemic update protocol as the rate of message exchange (the epidemic rate) is varied.

3. To determine the impact of the degree of replication on the system performance.

4. To ascertain the advantages of replication as a means for improving overall system performance.

All measurements in these experiments were made by running the simulation until a 95% confidence interval was achieved for each data point. This was typically after 15,000 transactions. The following are some of the key measurements and abbreviations used in the analysis. All measurements of time are given in milliseconds unless stated otherwise.

Pre-commit time If an update transaction pre-commits, it records the elapsed simulation time since it made its first read request of the system. The mean value for all such update transactions is the pre-commit time. All commit times (response times) are measured in milliseconds.

Final-commit time If a transaction commits (final commit), it records the elapsed simulation time since it made its first read request of the system and the mean for all committing transactions is the commit time. Only update transactions were included in this measure.

Read-only commit time Read-only transactions do not do a pre-commit, they simply commit. The commit time of read-only transactions is recorded separately from update transactions.

ThinkTime (TT) This is the mean of the exponential distribution for the transaction interarrival time per site. As the ThinkTime is made shorter, the load on the system increases and there is no multiprogramming level to limit the number of active transactions vying for resources and data item locks.

Epidemic rate (ER) The epidemic rate is given in milliseconds and tells how often a site is allowed to initiate an epidemic communication. If the site is busy with local processing, the actual rate may be less than the stated value.

5.1 Response Time Analysis

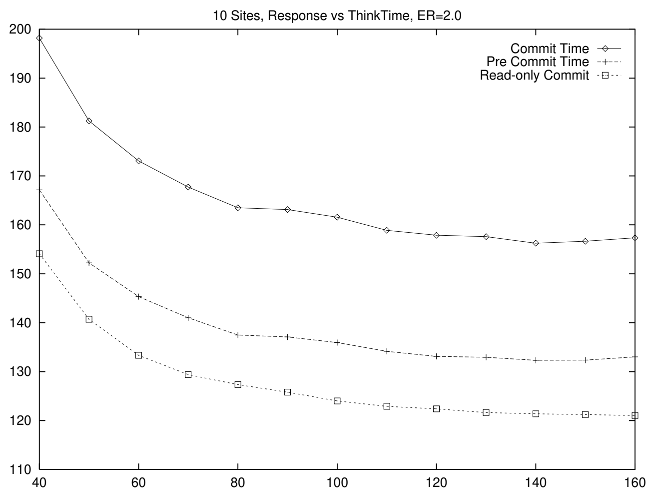


Figure 2: Response time for 10 sites

In our first set of experiments, we analyze the response time for both read-only and update transactions as a function of increasing ThinkTime. We consider a system with 10 sites (i.e., 10 copies) and an epidemic rate of 2.0 ms. The results in Figure 2 show the average commit time for read-only transactions as well as both the average pre-commit and commit time for update transactions. Both the x- and y-axis are in milliseconds. As expected, a greater load on the system, represented by a smaller ThinkTime, causes an increase in the time required for a transaction pre-commit as well as the transaction commit. Since read-only transactions execute locally, they are not adversely affected by the change in work load except at very low ThinkTime (i.e., high load) when the rate of concurrently executing transactions at each site is so high that conflicts are frequent and there is competition for resources. In fact, it is quite easy to account for the response time of read-only transactions. Since each transaction has an average of 9 operations requested 10 milliseconds apart, the issuing of the operations takes over 90 ms on the average. In addition, 1.0 ms of CPU time is consumed for processing each operation, adding 9 ms to the transaction time. Disk I/O for reads takes up 16.9 ms (9 operations at an 80% hit rate with an average 9.4 ms disk access time) for a total of 115.9 ms. At a low load, e.g., ThinkTime = 160 ms, a read-only transaction commits in about 121.0 ms. Hence a total of 5.1 ms (121.0 - 115.9) is spent on various database management functions such as log writes, lock table management and deadlock detection, that take place during the lifetime of the transaction as well as competition with other transactions for resources.

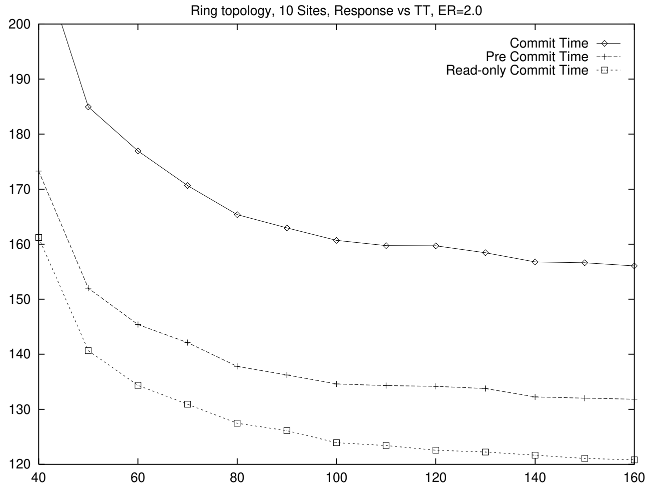
Update transactions take longer than read-only transactions to pre-commit since they must force write update data to the recovery log disk, requiring approximately 10 ms, and the average cost of a write is slightly higher than the average cost of a read operation. However, as with read-only transactions, an update transaction pre-commits based on local execution and requires no communication. Therefore the response time for the pre-commit of update transactions closely follows the response time of read-only transactions.

Committing update transactions, however, requires communication with all the other sites in the system. For a transaction to commit at the home site, the site must know that all sites have pre-committed that transaction. On average this delay is approximately 24 ms (at ThinkTime = 160 ms, transactions commit in 157.3 ms and pre-commit in 133.0 ms for an additional delay of 24.3 ms).

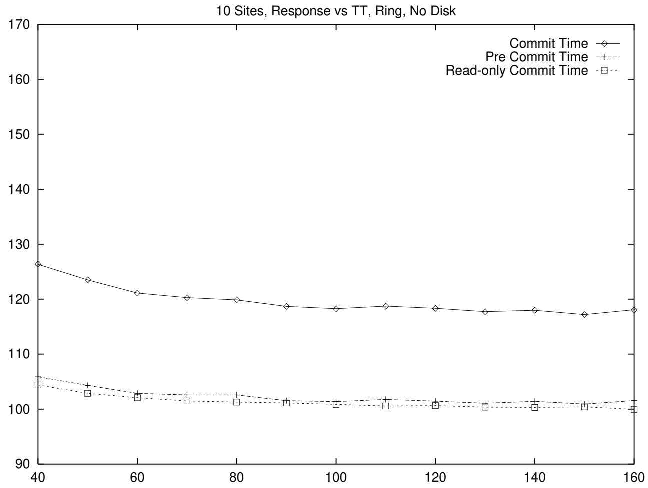
The additional time to commit is spent on disk I/O (a site which receives a pre-commit record must do the writes before putting it in its log and sending it on) and on communication costs.

To clearly account for the time between disk I/O and communication latencies, we need to avoid the random nature of the communication structure. We therefore considered a system of 10 sites ordered in a logical ring. In this setup, sites always send messages to the next one in order. This deterministic setting allowed us to compute how the time between pre-commit and commit was spent. A pre-commit record must visit every site in the system before committing. Also, the information that all sites have been visited will be reported back to the originating site after one round of messages through the ring. It turned out that in a 10 site ring (Figure 3(a)) the pre-commit and commit times did not change significantly from the random epidemic messages (see Figure 2). At high system loads, (ThinkTime less than 60 ms) the ring topology actually increased the commit time. Thus at TT=50, with random messaging the pre-commit time was 152.2 ms and the commit time was 181.2. With ordered messaging the pre-commit time was 152 ms but the commit time was 184. The situation worsened at TT=40. The large increase in response times in the ring configuration was due to queuing effects for the disk (all update records come in a single message at one time rather than spread out over time).

Since a lot of the time was consumed with disk I/O, we ran additional experiments with a cache hit rate of 1.0, thus all read and write accesses are to the memory. The results are shown in Figure 3(b) (note the change in the Y-axis). The results are interesting in several ways. First, all response times are faster. Second, the read-only commit and pre-commit times become the same. Lastly, the gap in time from pre-commit to commit is smaller and can be accounted for by known messaging costs. Hence, if replication is used in conjunction



(a) 10 sites on a Ring ER=2 ms



(b) 10 sites, Ring, No disk

Figure 3: 10 sites on a ring with and without disk I/O costs

with main-memory databases, update transactions with 10 database copies incur very low overhead in our evaluation (less than 20%). The results may actually be of value in a main memory system where replication (10 copies in the current experiment) is used for fault-tolerance instead of disks. If a site fails after pre-committing a transaction but before spreading it's record to other sites, the transaction information would be lost. The source of the transaction would never receive a final commit acknowledgment and could therefore detect the error. If a site fails after pre-committing a transaction and after spreading it to another site, the transaction information will be shared with other sites and can be recovered from them.

5.2 Varying Degree of Replication

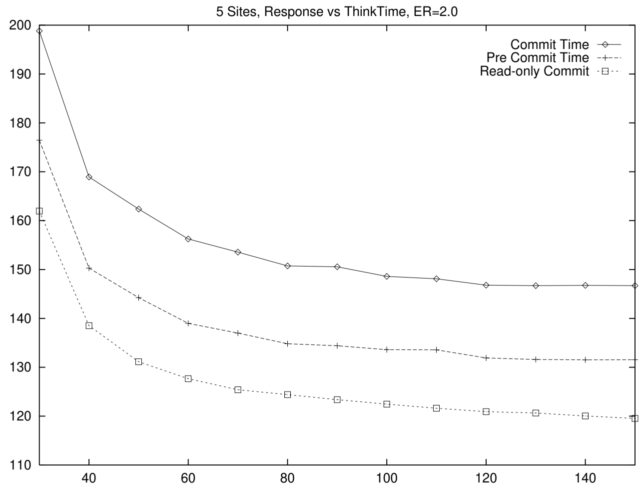
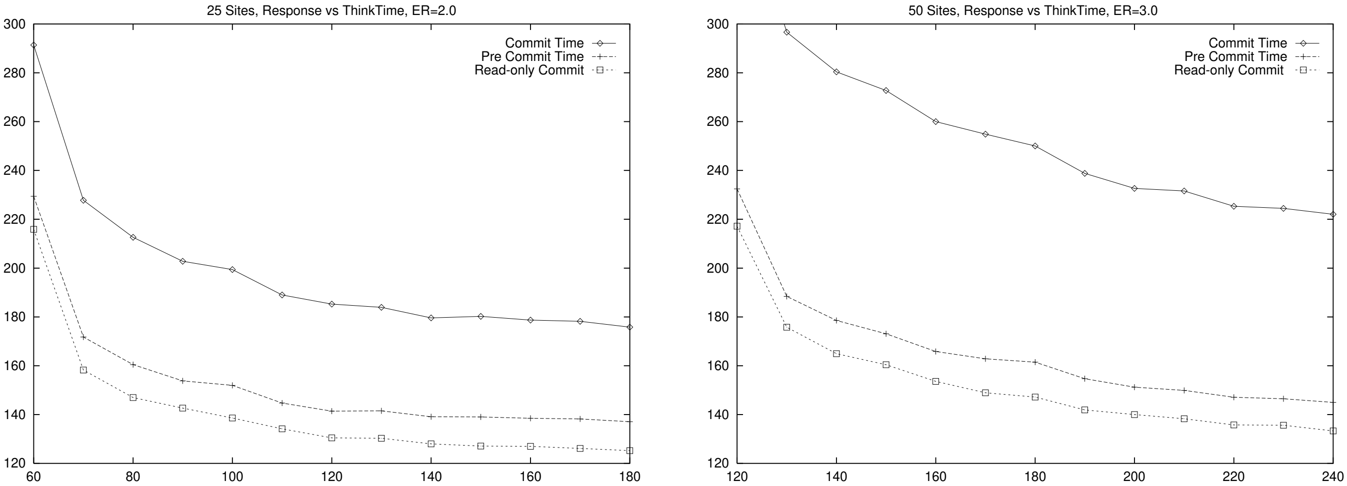


Figure 4: Response time for 5 sites



(a) 25 sites, ER = 2.0

(b) 50 sites, ER = 3.0

Figure 5: Response time for 25 and 50 sites

In this experiment we compared systems with 5 (Figure 4) 10 (Figure 2), 25 (Figure 5(a)) and 50 (Figure 5(b)) copies on different sites. We were particularly interested in the communications overhead introduced by the additional sites as opposed to the advantage of being able to handle more read-only transactions. Recall that 75% of the transactions generated are read-only and can thus be executed and committed at the home site. For the 50 site system an epidemic rate of 3.0 ms was used since the system was inundated with a large number of messages when all 50 sites sent messages every 2 milliseconds. These graphs show the effect of increasing the number of sites. Recall that a ThinkTime (transaction interarrival time) of 100 ms means a transaction is started every 100 ms at each site. A ThinkTime of 100 for 50 sites means 50 transactions are started every 100 ms. Thus, to evaluate a system load of 200 newly generated transactions per second, we need to consider a ThinkTime of 50 ms for a 10 site system, a ThinkTime of 120 ms for a 25 site system and a ThinkTime of 240 for a 50 site system.

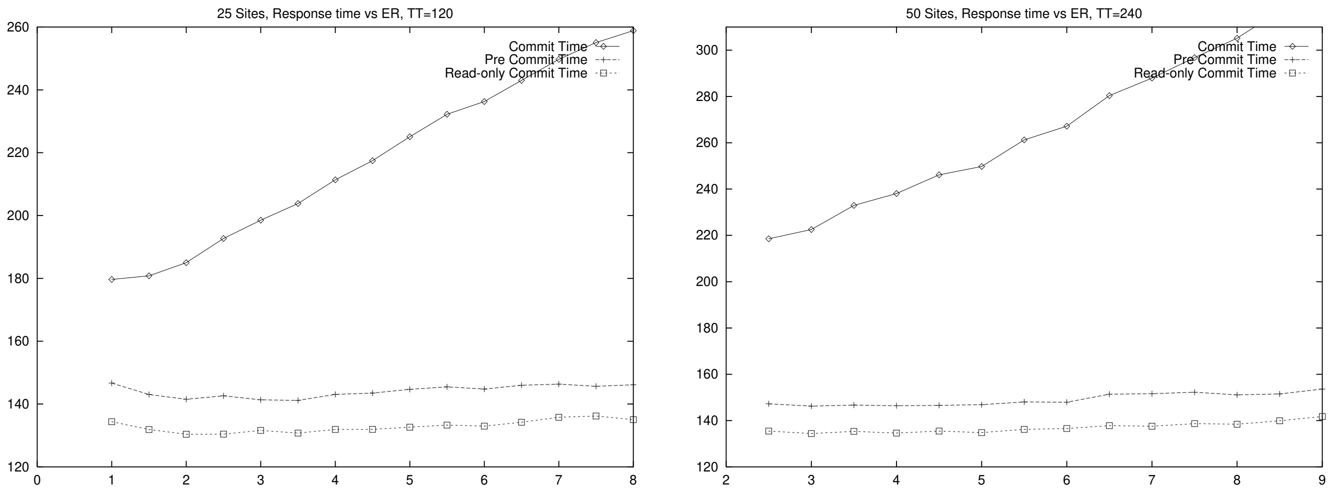
Notice that as the number of sites increases, the relative distance between pre-commit and commit time increases, indicating additional overhead introduced by the network. In a 10 site system with 200 new transactions per second (TT=50), the pre-commit time is 152.2 and the commit time is 181.2 This means the overhead introduced by the network to enable the transactions to commit and ensure serializability is (181 - 152) / 152 = 19.1% In a 25 site system with 200 transactions per second (TT=120), the pre-commit time is 141.4 and the commit time is 185.2 This means the overhead introduced by the network is 31.2% Finally, in a 50 site system 200 transactions per second (TT=240), the pre-commit time is 145 and the commit time is 222.0 with communications overhead of 71%

If we look at pre-commit times of less than 145 ms, that is, a system that is not excessively loaded, a 5 site system can handle 100 transactions per second (TT=50) a 10 site system can handle 166 transactions per second (TT=60) while a 25 site system can handle 227 transactions per second (TT=110) and a 50 site system can successfully handle 200 (TT=240). After a certain point, the possibility of being able to handle more transactions by adding sites to the system is outweighed by the additional overhead introduced by those sites. The optimal sized system, given the parameters of our model, for maximizing transaction capacity at a reasonable pre-commit time, is around 25 sites.

5.3 Varying the Epidemic Rate and Transaction Mix

In these experiments, we first varied the epidemic rate (ER) while holding other parameters constant. We varied epidemic rates from 1 ms to 8 ms on a 25 site system with a ThinkTime of 120 ms. The results (Figure 6(a)) show that as expected, the pre-commit time does not depend on the ER; however, the final commit time is highly sensitive to the ER. As the frequency of communication among the sites increases, transactions commit faster.

There is an interesting effect at an epidemic rate of 1 ms where the response times are slightly higher. The read-only commit time is 134.3 ms at ER=1 ms and 130.4 ms at ER=2 ms. Similarly, the update pre-commit time is 146.6 ms at ER=1 ms and 141.5 ms at ER=2 ms. When the epidemic rate is 1 ms, the sites are constantly sending and receiving messages with no new information (if site S_i's TimeTable indicates that the chosen receiver, S_k, knows about all the transaction records in S_i's log, S_i sends its Time Table T_i only). This overloads the CPU resulting in the delays in processing both read-only and update transactions. We observed that many of the messages sent at ER=1 ms had no new information and frequently a site had several incoming messages waiting to be processed. Thus the advantage of the shorter epidemic rate is offset by the disadvantage of a large amount of message processing. This effect was more pronounced in the 10 stie system (not shown).



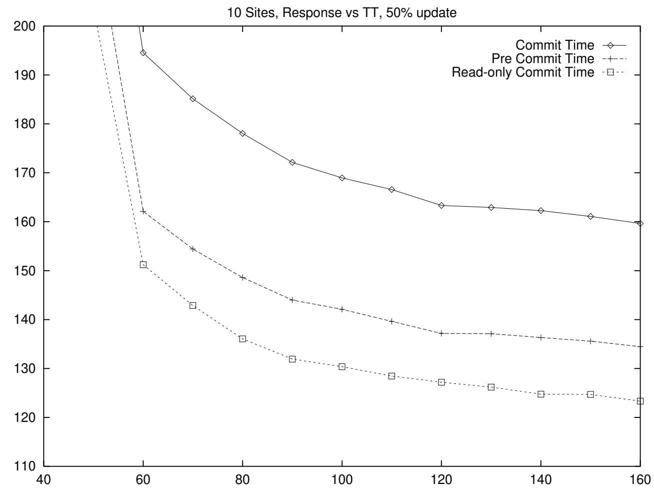
(a) 25 sites, TT=120

(b) 50 sites, TT=240

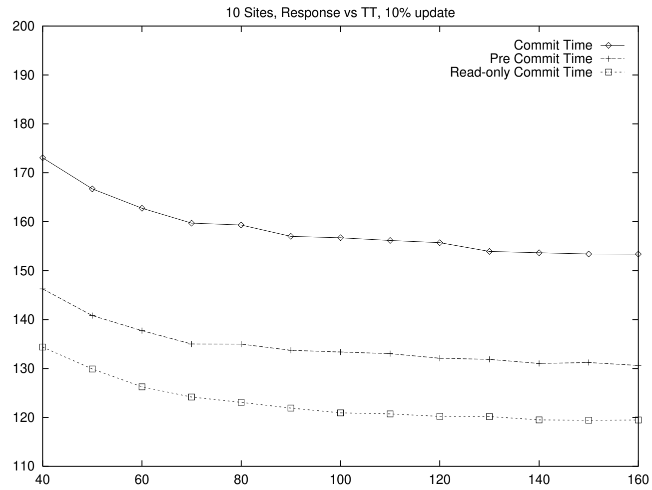
Figure 6: Response time as a function of epidemic rate

These experiments were repeated for 10 and 50 sites under different conditions. Figure 6(b) shows the 50 site experiment at a ThinkTime of 240 (thus it has the same number of new transactions generated per second as Figure 6(a)). This graph is typical of the other experiments in that pre-commit and read-only commit times are not affected by the epidemic rate, while final commit time increases with epidemic rate. From these experiments we learned that more frequent communication had a beneficial effect on the commit response time. Only when the epidemic rate was so fast that sites frequently sent messages with no transaction records did the performance of the system suffer.

In order to study the effects of updates on system performance, we varied the percentage of update transactions generated. Our earlier results that indicate that the response time of read-only transactions is unaffected by replication breaks down when the proportion of update transactions is increased to 50% from 25%. As



(a) 10 sites, 50% Read-only



(b) 10 sites, 90% Read-only

Figure 7: 10 sites with different proportion of read-only transactions

shown in Figure 7(a), the read-only commit time is greater due to the increased data contention (more update transactions means more write locks are held for a long time forcing readers to wait). At high load (low ThinkTime) the system is overwhelmed by data contention to a much greater degree than a 75% read-only environment. We also conducted experiments with higher percentage of read-only transactions. The results of an experiment with 90% read-only transactions are depicted in Figure 7(b). We observe that increasing the percentage of read-only transactions does not significantly influence the latency incurred in committing update transactions globally. However, due to decreased contention on resources there is an overall performance gain for all transactions.

5.4 Pre-commits as an Indicator of Commit

All read-only transactions commit locally and release all their locks without the need for any communication. Update transactions, on the other hand, first pre-commit, and then later commit at the home site when it is known that the transaction has pre-committed at all sites. When a transaction pre-commits, it releases all its read locks, however, the write locks are held until commit time. Recently there has been increasing interest in optimistic protocols which allow update transactions to release all locks at pre-commit time (and maybe even commit) by optimistically assuming that the transaction will actually succeed in committing. This is realistic in a system with low conflict rates and where the application can recover from any incorrect ramification of the optimistic decision. Note that if transactions release all locks at pre-commit time and later a conflict is detected, serializability can still be maintained, but the aborting of transactions may result in cascading aborts [4, 1]. The more radical approach of actually committing the transaction at pre-commit time (and thus the complete execution of the transaction is local) is actually more in line with current commercial systems that use replication. Oracle [16], for example, has developed application dependent reconciliation rules for conflict resolution. Jagadish, Mendelzon, and Mumick [11] have proposed a meta-rule language to specify which rules apply and in what order and to determine if a particular set of rules is unambiguous.

Since both cascading aborts and the application of reconciliation rules to resolve the effect of committing conflicting transactions should be minimized as much as possible, it is important to determine in an opti-

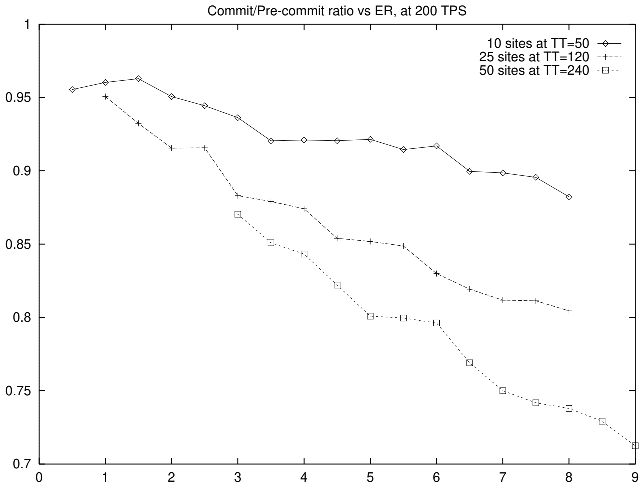


Figure 8: Commit/pre-commit ratio as a function of ER

mistic environment how frequently will pre-committed transactions not succeed in committing. We therefore measured the ratio of committed to pre-committed update transactions.

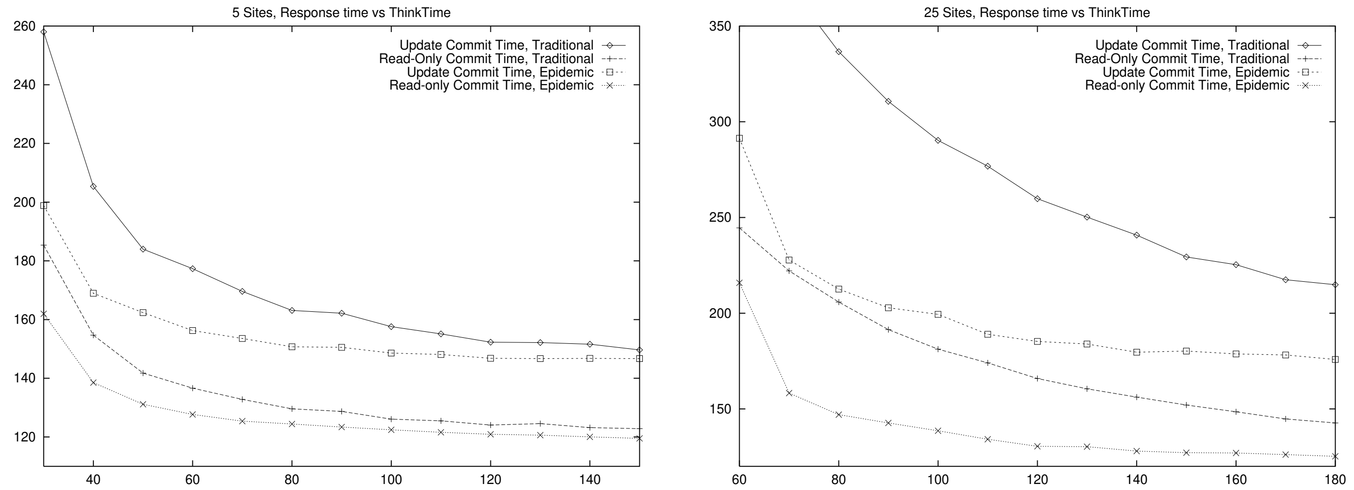
Presumably, when the epidemic rate was longer, the transactions would remain pre-committed but not yet committed for a longer time, and thus be more susceptible to conflicts and the subsequent abortion. We investigated 10, 25, and 50 site systems at a constant load of 200 new transactions per second. The results (Figure 8) show that if communication is frequent (every two milliseconds or less) in a 10 site system, 95% of the update transactions that pre-commit, go on to commit. This is quite encouraging as this is a moderately heavy load. The commit/pre-commit ratio for a 50 site system was less encouraging, but given the longer transaction lifetime caused by increased communication costs, not entirely unexpected.

We therefore conclude that in a moderately sized system with frequent communication between sites, an optimistic approach that uses the pre-commit as a *probabilistic* indicator of commitment is actually a viable option and deserves further investigation.

5.5 Comparison with Traditional Methods

In this section we explore the advantages of epidemic based updates versus a more traditional synchronous approach. A simple traditional update protocol allows for local execution of read-only transactions just like the epidemic protocol. When an update transaction does a write, the home site DBMS must acquire write locks for that data page at each replica site. The home site DBMS sends a message to each other site requesting a write lock. When the remote site is able to grant the lock, it responds with an acknowledgment. When the home site receives acknowledgments from all other sites, it lets the transaction perform the data write and proceed with its next operation. When the transaction has completed all its operations, the home site DBMS starts an atomic commit protocol such as a two phase commit [4].

In order to assess the performance of the epidemic protocol we modeled the traditional update protocol with our simulator. Since the traditional protocol can cause local and global deadlocks, we used a timeout mechanism to abort transactions which were waiting for locks past the timeout period. The optimal timeout period [2] for maximizing the transaction throughput was found to be approximately the response time of an update transaction. Accordingly, an adaptive timeout period was used that was based on the estimated transaction response time for a given ThinkTime (transaction interarrival rate).



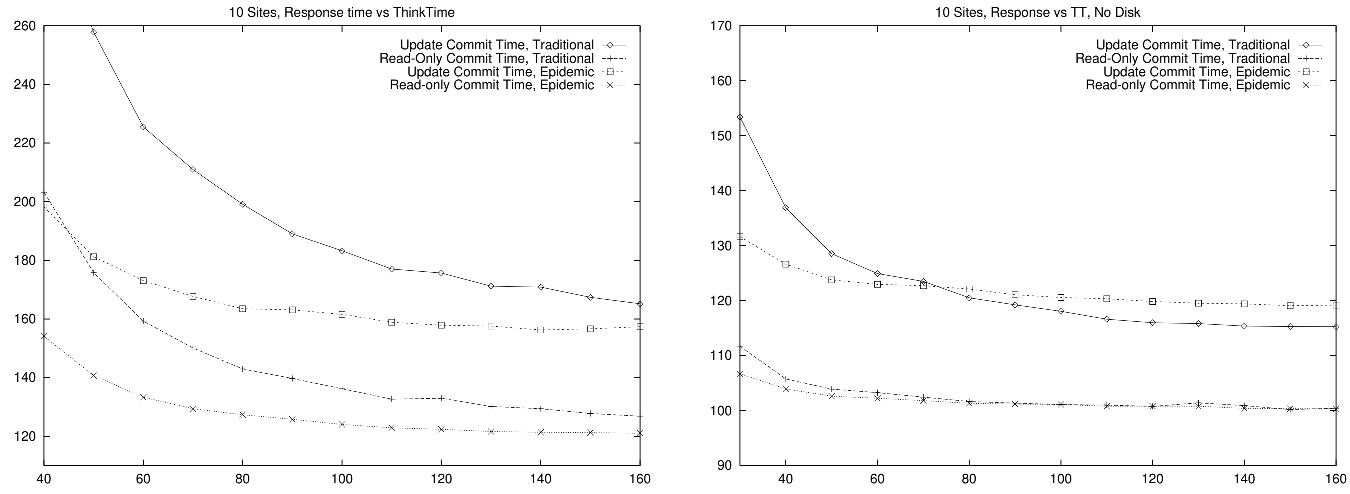
(a) 5 sites

(b) 25 sites

Figure 9: Response time for 5 and 25 sites

Experiments were performed using the traditional protocol with the same system and transaction parameters as the epidemic experiments. Response times for epidemic and traditional protocols are contrasted for 5 (Figure 9(a)) 10 (Figure 10(a)), and 25 sites (Figure 9(b)). In each case, the response times are greater for the traditional than for the epidemic based approach for both read-only and update transactions. It is especially worth noting that response time in a 25 site system improves dramatically. For example, with a think time of 100ms, the commit response time for update transactions goes from 290 ms for the traditional approach to 199 ms for the epidemic based protocol – an improvement of 31%.

The difference in read-only response time increases with increasing system load. We investigated the make-up of the read-only response times for traditional and epidemic protocols for the 10 site system (Figure 10(a)). Since the epidemic based protocol executes all write operations at remote sites together, the conflict potential and hence the blocking time between transactions is greatly reduced. We validated this hypothesis with the following measurements. Recall from Section 5.1 that 115.9 ms is spent on disk I/O and CPU processing and on inter-operation hold time. The remaining time in the read-only commit response time consists of waiting in a queue for the disk and the CPU and waiting for a lock on a data item, referred to as *blocking*. The average wait time for the disk and CPU can be calculated from the statistics reported by the simulation model, thus the remaining time is caused by blocking on a data item. At a ThinkTime of 120 ms, read-only transactions in the epidemic protocol spent an average of 4.2 ms waiting (for disk and CPU) and 3.9 ms blocked. The traditional protocol transaction spent 3.3 ms waiting and 12.7 ms blocked. At a higher system load (ThinkTime = 60 ms), epidemic read-only transactions spent 8.9 ms waiting and 10.0 ms blocked while the traditional read-only transactions spent 9.1 ms waiting and 34.6 ms blocked. It is clear that the traditional read-only transactions spend more time blocked and this increases with increasing system load. It is also worth noting that disk I/O and CPU wait times are greater for the epidemic transactions at ThinkTime = 120ms. (The difference in disk and CPU wait time at ThinkTime = 60 was not considered significant.) The apparent reason for this is that the write requests from a remote transaction are made all at once in the epidemic protocol, increasing the probability of having to wait for the data disk. In the traditional protocol, write requests from remote transactions are made one at a time with a 10 ms delay in between.



(a) Ten sites

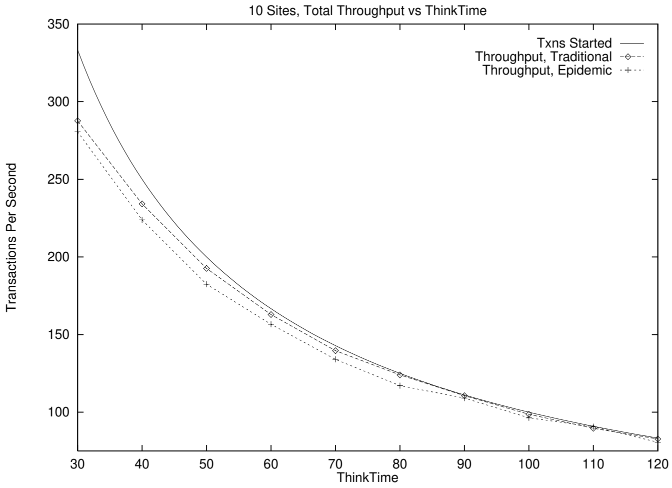
(b) Ten sites, No Disk

Figure 10: Response time for 10 sites with and without disk

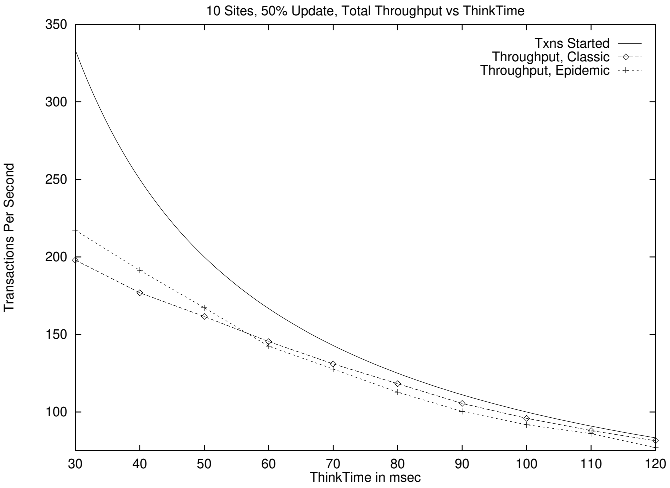
Update transactions have a longer commit time in the traditional protocol, even at low system load. This was surprising, since, at low system load, the effects of data and resource contention would be minimal and we expected that the efficiency of two phase commit would be an advantage over the somewhat random epidemic commit process (information propagation depends on the random communication patterns among sites). We ran an additional experiment with no disk (hit rate = 1 and log disk time = 0, as in section 5.1). The results (Figure 10(b)) show that removing the effects of disk I/O has a definite effect on commit time. The commit time for an update transaction in the traditional protocol reflects two forced writes of the recovery log disk: the home site force writes its log disk before initiating two phase commit and each remote site must force its log before responding in the affirmative. The commit time for update transactions in the epidemic protocol reflects only the forced write of the recovery log by the home site; the remote sites respond after an unforced write of the pre-commit record enabling the home site to commit the transaction. A remote site forces its log later when it commits the transaction. Removing the effects of disk I/O removes the advantage of the epidemic approach in terms of commit response time at low system loads and leaves it with an advantage only at high system loads, which, in the no disk case, means high data contention.

We also investigated the performance of the epidemic protocol in terms of total throughput. That is, how many transactions are actually committed per second. After all, good response time is meaningless if most of the submitted transactions are aborted. In figure 11(a) we show total throughput (number of transactions successfully committed per second) for a 10 site system as a function of ThinkTime for the epidemic and traditional protocols as well as a calculated line showing the number of transactions started (submitted to the system) for comparison. The proportion of read-only transactions is 75%. The throughput results for the epidemic and traditional protocols are very close, although for high system load (small ThinkTime) the traditional is slightly better. Figure 11(b) shows the same 10 site system but only 50% of the submitted transactions are read-only. With this heavier load of update transactions, the throughput results change to favor the epidemic protocol at high system load.

The epidemic protocol has several advantages over the traditional approach while maintaining the desirable characteristics of consistency and serializability. It relieves some of the limitations of the traditional



(a) Throughput for 10 Sites, 75% Read-only



(b) Throughput for 10 Sites, 50% Read-only

Figure 11: Throughtput Comparison, Epidemic and Traditional

approach by eliminating global deadlocks and reducing delays caused by blocking. The epidemic protocol maintains better performance as the system is expanded from 5 sites to 25 sites. In addition, the epidemic communication technique is more flexible than the reliable, synchronous communication required by the traditional approach. In order for an update transaction to commit in the traditional protocol, all sites must be simultaneously available and participating in the two-phase commit. In the epidemic protocol, all sites must eventually be available and participate in the epidemic commit, but because of the asynchronous communication, all sites need not be available at the same time. This is a great advantage in widely distributed systems that may experience transient failures and network congestion.

6 Discussion

Although lazy replication is bundled with many commercial database products, these products contain a disclaimer that they do not guarantee safety or consistency of replication. Currently proposed protocols include restricted execution models to ensure mutual consistency of database copies under lazy replication and protocols which allow inconsistency and non-serializable executions. We believe these limitations render the applicability of replication to very limited classes of applications. The epidemic protocol we study in this paper [1] ensures transactional serializability and replica consistency without requiring reliable communication and while tolerating transient network failures. The results of our performance evaluation indicate that for moderate levels of replication, epidemic replication is an acceptable solution to guarantee serializability. When the system is lightly to moderately loaded, read-only transactions are unaffected by replication and their response time is very predictable. Pre-commit and commit times for update transactions are also predictable. When compared to traditional synchronous replication, our approach results in improved response time for both read-only and update transactions. The improved performance is especially noted in situations with high data and resource contention. Also, as the number of sites increases, the advantages of epidemic updates increase (e.g., updates in a 25 site system at reasonable system loads have about 31% response time improvement). We intend to pursue our investigation of epidemic protocols in replicated database environments.

Many issues remain to be explored, including restricted network topologies, use of broadcast mechanisms to propagate messages, and the effect of network delay on performance.

References

[1] D. Agrawal, A. El Abbadi, and R. Steinke, "Epidemic algorithms in Replicated Databases," In *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson, Arizona, May, 1997.

[2] R. Agrawal, M. Carey, and L.W. McVoy. "The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems," In *IEEE Transactions on Software Engineering*, SE-13(12):1348-1363, December 1987.

[3] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. "Replication, consistency, and practicality: Are these mutually exclusive?" In *Proc. ACM SIGMOD Inter. Conf. Management of Data*, pages 484-495, Seattle, June 1998.

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

[5] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. "Update Propagation Protocols For Replicated Databases" In *Proc. ACM SIGMOD Inter. Conf. Management of Data*, pages 97–108, Philadelphia, PA, June 1999.

[6] D.K. Gifford, "Weighted Voting for Replicated Data", *Proceedings, 7th ACM Symposium on Operating Systems Principles*, pages 150-159, December 1979.

[7] J. Gray, P. Helland, P. O'Neil, and D. Shasha. "The Dangers of Replication and a Solution". In *Proc. ACM SIGMOD 1996 Int'l Conf. Management of Data*, Montreal, Quebec, pages 173-182, 1996.

[8] J.N. Gray. "Notes on database systems", In R. Bayer, R.M.Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, p. 393-481. Springer-Verlag, 1978.

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

[10] J. Holliday, D. Agrawal, A. El Abbadi, "The Performance of Database Replication with Group Multicast," *Proceedings, 29th IEEE International Symposium on Fault-Tolerant Computing*, pages 158-165, June 1999.

[11] H.V. Jagadish, A.O. Mendelzon, I.S. Mumick, "Managing Conflicts between Rules", *Proceedings, 1996 ACM Symposium on Principles of Database Systems*, pages 192-201, 1996.

[12] V. Kumar, editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, Prentice Hall, 1996.

[13] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, 21(7):558-565, July 1978.

[14] B. Liskov, R. Ladin, “Highly Available Services in Distributed Systems”, *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 29-39, August 1986.

[15] F. Mattern, “Time and global states of distributed systems”, *Proceedings, 1988 International Workshop on Parallel and Distributed Algorithms, Bonas, France*, North Holland, 1989.

[16] Oracle. *Oracle 7 Server Distributed Systems: Replicated Data*. http://www.oracle.com /products /oracle7 /server /whitepapers /replication /html /index.

[17] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A. J. Demers, “Flexible Update Propagation for Weakly Consistent Replication,” *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (SOSP-16), Saint Malo, France, October 5-8, 1997, pages 288-301.

[18] M. Rabinovich, N.H. Gehani, A. Kononov, “Scalable Update Propagation in Epidemic Replicated Databases,” *Proceedings, International Conference on Extending Database Technology*, p207-222, 1996.

[19] R.H. Thomas, “A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases”, *ACM Transactions on Database Systems*, 4(2):180-209, June 1979.

[20] G.T. Wuu and A.J Bernstein, “Efficient Solutions to the Replicated Log and Dictionary Problems,” *Proceedings, Third ACM Symposium on Principles of Distributed Computing*, p.233-343, August 1984.