Database Replication Using Epidemic Communication

JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi*

University of California at Santa Barbara, Santa Barbara, CA 93106, USA
{joanne46,agrawal,amr}@cs.ucsb.edu

Abstract. There is a growing interest in asynchronous replica management protocols in which database transactions are executed locally, and their effects are incorporated asynchronously on remote database copies. In this paper we investigate an epidemic update protocol that guarantees consistency and serializability in spite of a write-anywhere capability and conduct simulation experiments to evaluate this protocol. Our results indicate that this epidemic approach is indeed a viable alternative to eager update protocols for a distributed database environment where serializability is needed.

1 Introduction

Data replication in distributed databases is an important problem that has been investigated extensively. 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 consistency of replicated data. Traditional approaches for replica management incur significant performance penalties.

The traditional replica management approach requires the synchronous execution of the individual read and write operations to be executed on some set of the copies *before* transaction commit. An alternative approach is to execute operations locally without synchronization with other sites, and after termination, the updates are propagated to other copy sites [7, 8]. In this approach, changes are propagated throughout the network using an epidemic approach [8], where updates are piggy-backed on messages, thus ensuring that eventually all updates are propagated throughout the system. The epidemic approach (also called asynchronous logging) works well for single item updates or updates that commute. However, when used for multi-operation transactions, these techniques do not ensure serializability. To overcome this problem, Anderson et al. [2] and Breitbart et al. [3] 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. We have developed a hybrid approach where a

* This work was partially supported by NSF grants CCR97-12108, EIA 9818320, IIS 98 17432, and IIS 99 70700

transaction executes its operations locally, and before committing uses epidemic communication to propagate all its updates to all replicas [1]. Once a site is sure that the updates have been incorporated at all copies, the transaction is committed. This approach ensures serializability without imposing restrictions on which sites can process update transactions or which database items can be accessed. This approach also has the advantages of epidemic propagation, namely the asynchronous propagation of update operations throughout the system which is tolerant of network delays and temporary partitions. In this paper we explore the potential benefits of epidemic communication for replica management and use a detailed database simulation to evaluate its performance.

2 System Model and Epidemic Update Protocols

We consider a distributed system consisting of a number of database server sites each maintaining a copy of all the items in the database. The sites are connected by a point-to-point network that is not required to be reliable. A transaction can originate at any site, and that site becomes the initiating or home site. Vector clocks, an extension of Lamport clocks, are used to preserve potential causal relations among operations. Vector clocks can detect if an event causally precedes, follows, or is concurrent with another event. In addition to vector clocks, each site maintains an *event* log of transaction operations. This log is not the same as the database recovery log [4] as it is used solely for epidemic communication purposes. Sites exchange their respective event logs to keep each other informed about the operations that have occurred in the system. 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). 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 S_i does not know if S_k has received a record of t, 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.

In [1], this approach is extended to support multi-operational transactions in a database. Since strict two-phase locking [4] is widely used, we assume that concurrency control is locally enforced by the strict two phase locking protocol at all server copy sites. When a transaction, t, successfully completes its operations at the home site, S_i, it *pre-commits*. If the transaction is read-only, it can be committed at that time. 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 event 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. At this point there is still the possibility that the transaction will be aborted due to conflicts with other pre-committed transactions.

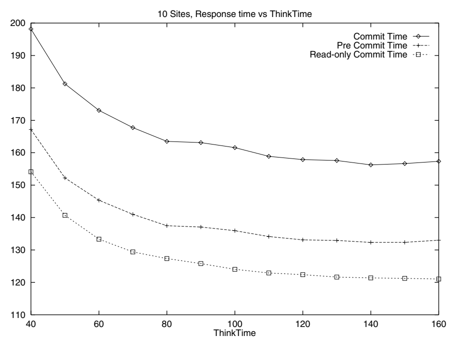
When a site S_i contacts site S_k to initiate an epidemic transfer, S_i determines which of its log records have not been received by S_k and sends those records along with S_i's time table T_i. When S_k receives the message, it reads the transaction records in order and determines if there is any conflict 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. If there are any conflicts, to enforce serializability 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. 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 (via the time table) that all sites have knowledge of that transaction. This protocol ensures serializability and is explained more completely in [1]. The protocol also tolerates temporary site failures, since the information is stored in the log, and remains there until it has been received by all sites.

3 Performance Results

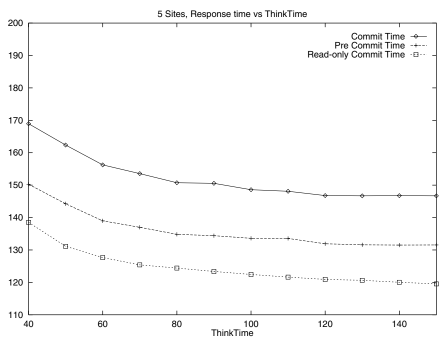
Our simulation [6] is based on standard database simulation models. The system and transaction parameters of the model are given in Table 1 along with their values. The generation of new transactions is governed by a parameter called

Parameter	Value	*Parameter*	Value
Number of data disks per site	1	ER – Epidemic rate	varies
CPU time needed to access disk	0.4 ms	Data disk access time	4–14 ms
Time for forced write of log	10 ms	Cache hit rate	0.8
Number of log records per page	100	Transaction read set size	5–11
Time between operation requests	10 ms	Transaction write set size	1–4

Table 1. System and Transaction Parameters



(a) Response Time for 10 Sites



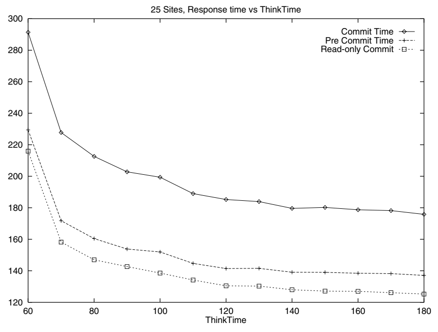
(b) Response Time for 5 Sites

Fig. 1. Response times as a Function of ThinkTime

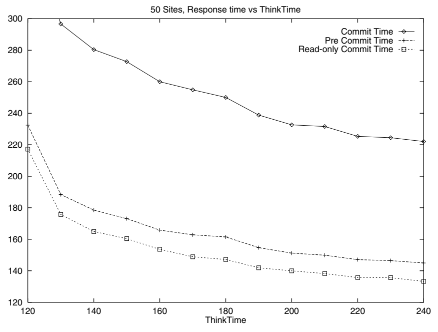
"ThinkTime". This is the per site transaction interarrival time and corresponds to the "open" queuing model. The percentage of read-only transactions is 75% unless otherwise stated. When a site is ready to initiate an epidemic session with another site, it chooses a receiver site at random. The epidemic rate determines how often a site may initiate an epidemic update. All measurements in these experiments were made by running the simulation until a 95% confidence interval of 1% was achieved for each data point. All measurements of time are given in milliseconds unless stated otherwise.

3.1 Response Time Analysis

In our first set of experiments, we analyze the response time for both read-only and update transactions as a function of ThinkTime. We consider a system with 10 sites (i.e., 10 copies) and an epidemic rate of 2.0 ms. The results in Figure 1(a) 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. Since read-only transactions execute locally, they are not adversely affected by the change in work load except at very low ThinkTime 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. Each transaction has an average of 9 operations requested 10 milliseconds apart, so the issuing of the operations takes over 90 ms on 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, 80% hit rate, 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 is spent on various database management functions such as log writes, lock table



(a) 25 sites, ER = 2.0



(b) 50 sites, ER = 3.0

Fig. 2. Response time for 25 and 50 sites

management and deadlock detection, which 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 requires communication with all the other sites in the system since the site must know that all sites have pre-committed that transaction. On average this delay which consists of 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 is approximately 24 ms.

3.2 Varying Degree of Replication

Next, we compared systems with 5 (Figure 1(b)) 10 (Figure 1(a)), 25 (Figure 2(a)) and 50 (Figure 2(b)) copies. We were 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. These graphs show the effect of increasing the number of sites. 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. In a 10 site system with 200 new transactions per second, the pre-commit time is 152.2 and the commit time is 181.2. Thus, the overhead introduced by the network to enable the transactions to commit and ensure serializability is 19.1%. In a 25 site system the overhead is 31.2% and in a 50 site system the overhead is 71%. If we look at pre-commit times of less than 145 ms, a reasonable response time, a 5 site system can handle 100 transactions per second, a 10 site system can handle 166 transactions per second, while a 25 site system can handle 227 transactions per second and a 50 site system can successfully handle 200. 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.

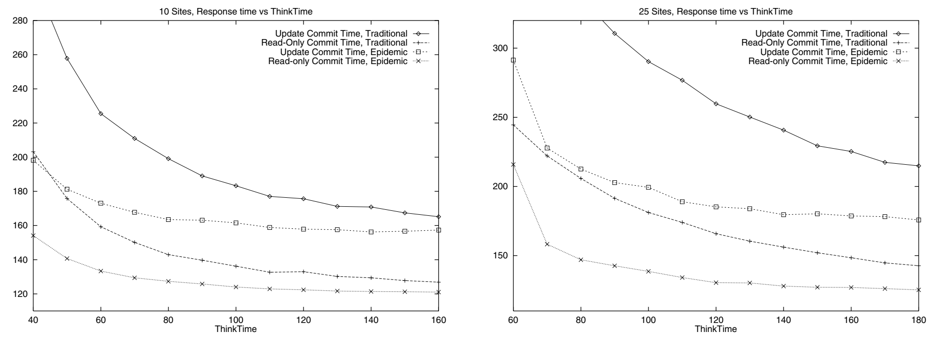
Since a lot of the time was consumed with disk I/O, we also performed experiments with a cache hit rate of 1.0, thus all read and write accesses are to the memory. Other experiments varied the transaction mix, network configuration and epidemic rate. These results are reported in [6].

3.3 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 a two phase commit protocol [4]. In order to assess the performance of the epidemic protocol we modeled the traditional update protocol with our simulator [6].

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 10 (Figure 3(a)), and 25 sites (Figure 3(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. For example, in a 25 site system with a think time of 100ms, the commit response time for update transactions for the epidemic based protocol is 31% less than for the traditional approach.

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 3(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 by measuring the wait time for the disk and CPU and the blocking time (the time a transaction is waiting for a lock on a data item). For example, 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, epidemic read-only transactions spent



(a) Ten sites (b) 25 sites

Fig. 3. Response time for 10 and 25 sites

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 was clear that the traditional read-only transactions spend more time blocked and this increases with increasing system load. This is further explained in [6].

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. The results (in [6]) 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.

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. When the proportion of read-only transactions is 75% on a 10 site system, the throughput results [6] for the epidemic and traditional protocols are very close, although for high system load the traditional is slightly better. When only 50% of the submitted transactions are read-only, the throughput results change to favor the epidemic protocol at high system load.

4 Discussion

The epidemic protocol [1] relieves some of the limitations of the traditional approach by eliminating global deadlocks and reducing delays caused by blocking. 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 but need not be available at the same time. This is a great advantage in distributed systems that may experience transient failures and network congestion. This protocol has also been extended to use quorums to resolve commit decisions [5].

Current 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 restrict replication to very limited classes of applications. The epidemic protocol we study in this paper [1] ensures transactional serializability and replica consistency without restricting updates or 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.

References

[1] Agrawal, D., El Abbadi, A., Steinke, R.: Epidemic Algorithms in Replicated Databases. Proceedings, ACM Symposium on Principles of Database Systems, May 1997, 161–172

[2] Anderson, T., Breitbart, Y., Korth, H.F., Wool, A.: Replication, consistency and practicality: Are these mutually exclusive? Proceedings, ACM SIGMOD, June 1998, 484–495

[3] Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S.: Update Propagation Protocols for Replicated Databases. Proceedings, ACM SIGMOD, June 1999.

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

[5] Holliday, J., Steinke, R., Agrawal, D., El Abbadi, A.: Epidemic Quorums for Managing Replicated Data. Proceedings, 19th IEEE IPCCC, Feb. 2000

[6] Holliday, J., Agrawal, D., El Abbadi, A.: Database Replication Using Epidemic Update. Technical Report TRCS00-01, Computer Science Dept. University of California at Santa Barbara, January 2000

[7] Liskov, B., Ladin, R.: Highly Available Services in Distributed Systems. Proceedings, 5th ACM Symposium, Principles of Distributed Computing, August 1986, 29–39

[8] Petersen, K., Spreitzer, M., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible Update Propagation for Weakly Consistent Replication. Proceedings, 16th ACM Symposium on Operating Systems Principles, 1997, 288–301