Epidemic Quorums for Managing Replicated Data *

JoAnne Holliday Robert Steinke Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, California 93106

Abstract

In the epidemic model an update is initiated on a single site and is propagated to other sites in a lazy manner. When combined with version vectors and event logs, this propagation mechanism delivers updates in causal order despite communication failures. We integrate quorums into the epidemic model to process transactions on replicated data while ensuring global serializability. We present a detailed simulation of a distributed replicated database and demonstrate the performance improvements.

1 Introduction

Asynchronous replication has been deployed successfully for maintaining control information in distributed systems and computer networks. For example, name servers, yellow pages, and server directories are maintained redundantly on multiple sites and updates are incorporated in a lazy manner through gossip messages, epidemic propagation, and anti-entropy [12]. In this paper we use the epidemic communication model as the basis for an algorithm that supports transaction processing in replicated databases.

In an epidemic system, sites perform update operations and then communicate in a lazy manner to propagate the effects of those operations. Sites communicate in a way that maintains the causal order of updates even in the face of lost messages. To learn of a particular update a site does not have to receive a message directly from the site that performed that update. Communication can pass through one or more intermediate sites. Therefore, the epidemic model provides an environment that is tolerant of communication failures and does not require that all sites be available at the same time as do traditional eager replication techniques. One can view an epidemic system as a set of computers that are normally disconnected except for short periods of time when one computer connects to another to send an epidemic message. In environments that suffer from constant disconnection and re-partitioning, and it may be that no partition ever contains a quorum of sites. However, by using the epidemic model, a quorum can be found and updates performed on a single site can eventually reach all other sites in the system.

Earlier protocols based on epidemic communication considered single operation applications such as replicated dictionaries. Bayou [12] is an example of a system that supports weakly consistent replicated servers. It accommodates a variety of update policies and operates in a variety of network topologies. Ladin et al. [8] demonstrate the usefulness of lazy propagation protocols to maintain highly available services in a distributed system. Other systems such as Coda [14] were developed for weakly consistent systems. However, these systems support either single operation transactions or multiple operations on a single file or data item. Some commercial database systems use lazy propagation, for example, Oracle 7 [11] uses a 2-tier replication scheme, however, inconsistencies may arise and a variety of reconciliation rules are provided to merge conflicting updates. Gray et al. [5] argue that synchronous approaches for managing replicated data do not scale well. They further state that lazy approaches should be explored for managing replicated data and propose a primary/secondary lazy replication scheme. More recently, lazy propagation protocols [3] have been introduced for database replication. Although these protocols guarantee one-copy serializability, they impose a structure on the sites restricting how and where transactions can perform updates.

We have recently adapted the epidemic approach for transaction based systems by developing a family of algorithms for managing replicated databases using the epidemic model [1]. In these algorithms, updates can occur at any site without any restriction to a designated primary site and without imposing a structure on the sites. Sites lazily exchange event log records recording the operations of each transaction and then

*This research was partially supported by LANL under grant number 6863V0016-3A, and by the NSF under grant numbers CDA94-21978, CCR95-05807, IIS98-17432 and IIS99-70700.

apply those transactions asynchronously. By comparing log records, sites can determine which transactions would create serializability conflicts and must be aborted thus preserving one-copy serializability.

Our approach in this paper is to combine the quorum approach [4] within the epidemic framework to achieve balanced treatment for both queries and updates while maintaining global serializability. The quorum approach is also more tolerant of site and communication failures. In the next section we present the epidemic model of replication. In Section 3 we begin with an overview of the epidemic read-one/write-all (ROWA) protocol and then develop the epidemic quorum algorithm. In Section 4 we present the results of a performance evaluation using a detailed simulation. Section 5 concludes the paper.

2 The Epidemic Model of Replication

We consider a distributed system consisting of n sites labeled $S_1, S_2, ..., S_n$. We assume a fail-stop model of site failures and an unreliable communications medium. 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. For this reason, timeouts were not used in the protocol to detect conflicts and deadlocks. An event model [9] 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 two conditions: events occurring at the same site are totally ordered, and if e is a send event and f is the corresponding receive event then $e \rightarrow f$. Epidemic algorithms generally are implemented using vector clocks [10] to ensure the property that if two operations are causally ordered, their effects should be applied in that order at all sites, thus preserving the happened-before relation. Vector clocks ensure the following property:

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

If $Time(e)$ and $Time(f)$ are incomparable, denoted $Time(e) <> Time(f)$, then events e and f are concurrent.

Application specific operations are executed locally and they are communicated to the other sites by using the epidemic *push* [12] model of communication which is that information is spread from site to site when two sites communicate and share information. In the log based approach each site maintains an event log of application specific operations. Sites exchange their respective event 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. Wuu and Bernstein [15] combine logs and vector clocks as follows. 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 Wuu and Bernstein define the $HasRecvd$ predicate as:

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

where t is an event, $Site(t)$ is the site at which t occurred, and $Time(t)$ 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 event 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.

3 Epidemic Transactional Replication

The Wuu and Bernstein approach supports single operation semantics. That is, all of the operations of a transaction are on a single data item and they all commute. Our approach [1] supports general database transactions in a fully replicated database where a single transaction can read and write several distinct data items with atomicity and isolation [2]. We will refer to this as the epidemic Read-One Write-All (ROWA) protocol. To enforce atomicity the algorithm treats each transaction as a single event with a single record in the event log rather than having one event per read or write operation. Each transaction runs under the local concurrency control mechanism on a single site referred to as its *initiating site*. When a read-only transaction has performed all of its reads, it can be committed locally. When an update transaction completes all of its operations at the initiating site, it requests a *pre-commit*. When a transaction t, pre-commits on its initiating site, S_i, the local clock is incremented and a pre-commit record containing the readset ($RS(t)$), writeset ($WS(t)$), the values written,

and a pre-commit timestamp ($TS(t)$) from the initiating site's vector clock is written to the local log and the read locks held by the transaction are released. Then sites communicate log records to detect global conflicts and propagate values written by 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. All transaction records t such that $HasRecvd(T_i, t, S_k)$ is false are sent in a message along with S_i's time table T_i.

When a site receives a log record it initiates a transaction to apply the results of the original transaction to that site. The original instance of a transaction running on its initiating site is referred to as a *local transaction*. The transaction instances that run on other sites to propagate the local transaction's updates are referred to as *remote transactions*. If the receiving site does not find a conflicting transaction already in the log, it executes a remote transaction 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.

To enforce one-copy serializability the algorithm aborts all concurrent transactions with conflicting operations. A site can detect if two transactions are conflicting because their log records contain their read sets, write sets and version vectors which determine concurrency. A transaction t cannot commit until it knows that there are no conflicting transactions in the system which would cause it to abort. Thus, 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. Site S_i will know about all possible concurrent and conflicting transactions when it knows that all sites have knowledge of t because the propagation of event log records preserves the causal order of transactions.

3.1 Epidemic Quorums

The read-one/write-all epidemic algorithm just described is inefficient in that it aborts all conflicting transactions. One-copy serializability only requires that for any set of conflicting transactions at most one commits. To increase system throughput it would be desirable to commit one transaction from each set of conflicting transactions. To take advantage of this optimization all sites must agree on which transaction to commit. This agreement problem can be solved in an efficient way through the use of *quorums* [4]. Quorums are sets of sites such that the intersection of any two quorums is non-empty. For example, a majority quorum [4] is any set that contains a majority of sites.

We propose to use voting and quorums to resolve commit decisions. This is similar to the way they are used by Keleher [7] except our protocol ensures serializability among multi-operation transactions, whereas Keleher treats only single-operation requests. Transactions are serialized in causal order, and the algorithm guarantees that for each pair of conflicting transactions at most one commits. This is accomplished by having each site vote *yes* or *no* on each transaction. A site never votes *yes* for two conflicting transactions. When a site votes, it places a vote record in its log indicating the transaction, the site voting, and whether the vote is *yes* or *no*. These vote records are piggy-backed on the usual epidemic messages so that all sites eventually receive a vote record from all sites for each transaction. When a site receives a quorum of *yes* votes for a transaction it commits the transaction. When a site knows that a transaction will never receive a quorum of *yes* votes it aborts the transaction.

How can a site know that a transaction will never receive a quorum of *yes* votes? Obviously, if one transaction commits then all conflicting transactions must abort, however there may be a situation where multiple conflicting transactions hold votes such that none will ever commit. For example, in the majority quorum system, three transactions could each hold one third of the votes. It is important that this situation not lead to indefinite blocking. To cope with this we introduce the idea of an *anti-quorum* such that having an anti-quorum of *no* votes implies that you can never get a quorum of *yes* votes. More formally, an anti-quorum is any set of sites that intersects with all quorums. Any quorum is an anti-quorum, but an anti-quorum is not necessarily a quorum because we do not require anti-quorums to intersect with each other.

Based on this definition a site S_i can abort transaction t as soon as it receives *no* votes from an anti-quorum of sites. At this point it is guaranteed that no site will ever commit t. Using these three conditions, commit on a quorum of *yes* votes, abort when a conflicting transaction commits, and abort on an anti-quorum of *no* votes, S_i will always be able to either commit or abort t by the time it receives votes on t from all sites.

When S_i has received t, but has not received enough information to commit or abort t, t is said to be *uncertain* at S_i and is similar to the state of a *tentative* transaction in the Bayou system [12]. This poses the problem of what to do with uncertain conflicting transactions. To preserve causality, when t arrives at S_i it acquires write locks and applies its writes before the next received transaction is processed. However, a conflicting transaction may have written to a common data item x, and if they are both uncertain neither can

be aborted. It seems that they must both hold write locks on x. This situation is not incorrect. The important point is that neither of these transactions will initiate any new operations on x, no other transaction can access x until its value is committed, and at most one of the transactions will commit. If t commits then all conflicting transactions must abort, and t can write the correct value of x before releasing the lock.

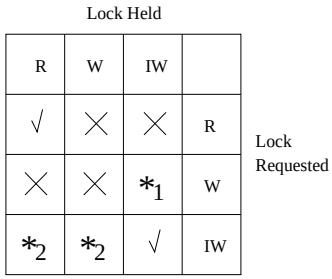


Figure 1: Lock conflict table for read (R), write (W), and intention-to-write (IW) locks.

To solve this problem we adapt an idea from multigranularity locking called *intention-to-write* locks [2]. The conflict table for intention-to-write locks is given in Figure 1. Intention-to-write locks do not conflict with other intention-to-write locks, but do conflict with read and write locks. When the log record of a transaction arrives at a site it initiates a remote transaction. This remote transaction acquires intention-to-write locks on all data items written by the original transaction and pre-commits. When a local transaction pre-commits it no longer needs access to local data items, and it might become involved in this kind of conflict. So, the local transaction releases all of its read locks, converts all of its write locks to intention-to-write locks, and behaves like a remote transaction. This prevents local transactions that have not pre-committed from accessing the data item, but allows conflicting, uncertain transactions to pre-commit and simultaneously reserve access to the data item. If two transactions hold intention-to-write locks on a data item and one of them aborts then the data item is still inaccessible until the fate of the second transaction is determined. When a transaction commits it converts its intention-to-write locks into write locks, applies its updates to the data items, and releases the locks.

In this algorithm intention-to-write (IW) locks create two special situations. In both of these situations, referred to as $*_1$ and $*_2$ in the lock conflict table (Figure 1) the lock request will cause the abort of the lock holder. In the first situation, $*_1$, two or more transactions hold intention-to-write locks on a data item, x, and one of the transactions, say t_1, wants to convert its IW lock to a write lock. This will only happen when t_1 has received enough *yes* votes and is ready to write its items and commit. Since all the other transactions holding IW locks on x are concurrent and conflicting, they must abort and allow t_1 to obtain the write lock.

The second situation is referred to as $*_2$ in the lock conflict table. In this situation, a local transaction, say t_2, which has not yet pre-committed (at pre-commit time it releases its read locks and converts its write locks to IW locks) holds a read or write locks on x and another transaction, t_1, wants a IW lock. Transaction t_1 is either a local transaction which has come to its pre-commit point or a remote transaction. In any case, t_1 and t_2 are concurrent conflicting transactions and one will have to abort. It makes sense to abort t_2 right away and give the intention-to-write lock to t_1 since t_2 has less work invested so far because it hasn't yet reached its pre-commit point.

Due to these two rules remote transactions never wait for a lock. This has the desirable effect of eliminating distributed deadlocks. Local transactions can wait for locks held by remote transactions, but these remote transactions will never wait for locks so there can be no deadlock cycle involving remote transactions. Local and remote transactions can wait for other sites to vote on associated remote transactions, but at this time the waiting transaction must be pre-committed and thus not be waiting for any locks. Deadlock cycles can only involve local transactions waiting for locks from other local transactions at the same site. These deadlocks can be detected without communication. The detailed algorithm and data structures needed for implementation and an example execution can be found in a UCSB technical report [6].

4 Performance Evaluation

In order to evaluate the performance of epidemic algorithms with and without quorums, we used a detailed simulation of a distributed replicated database. The simulation is based on accepted database modeling techniques [13]. All measurements in these experiments were made by running the simulation until a 95% confidence interval was achieved for each data point.

Transactions are modeled as sequences of read and write operations. The time between successive operation requests within a transaction is 3ms. 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. A page is assumed to be 2Kbytes and is the locking granularity as well as the data disk granularity. The each local database uses strict two-phase locking to ensure local serializability.

Parameter	*Meaning*	Value
ThinkTime	Transaction interarrival time	varies
NumSites	Number of sites	5, 10, 25
ER	Epidemic rate	3ms
MinDiskTime	Smallest data disk access time	4 ms
MaxDiskTime	Maximum data disk access time	14 ms
CPUInitDisk	CPU time needed to access disk	0.3 ms
LogDiskTime	Time used for forced write of log	8 ms
LogPageSize	Number of log records per page	100
HitRate	Probability of cache hit	0.9

Table 1: System Parameters

A *wait-for-graph* for handling local deadlocks is maintained and is checked for cycles each time an edge is added. The epidemic protocols we are using guarantee there will be no global deadlocks.

The system parameters of the model are given in Table 1 along with their values. The generation of new transactions is governed by the ThinkTime. This is the per site transaction interarrival time and can be made arbitrarily long or short. There is no multiprogramming level to limit the number of active transactions vying for resources and data item locks. The percentage of read-only transactions is 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 for read-only transactions. 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 so that any site can exchange messages directly with any other site. We assume that the network is fast (100 Mbits/sec). When a site is ready to initiate an epidemic session with another site, it chooses that site at random from the remaining sites. The epidemic rate is given in milliseconds and tells how often a site is allowed to initiate an epidemic communication. In all these experiments we fixed the epidemic rate to 3ms. If the site is busy with local processing, the actual time between transmissions may be more than the stated value.

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 in milliseconds since it made its first read request of the system. The mean value for all such update transactions is the pre-commit time.

Commit or Update commit time If a transaction commits (final commit), it records the elapsed simulation time in milliseconds since it made its first read request of the system and the mean for all committing transactions is the commit time. Only update transactions are 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 and is expressed in milliseconds. As the ThinkTime is made shorter, the load on the system increases.

Start rate The transaction start rate is measured in transactions per second and is the rate at which new transactions are generated. This determines the load on the system and is equal to NumSites*1000/TT.

Commit rate The commit rate, also called the throughput rate, is the number of transactions, both read-only and update, committed by the system per second.

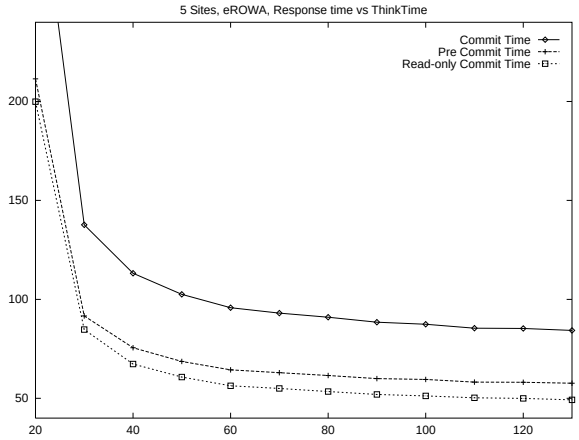
Read-only commit rate The read-only commit rate is the number of read-only transactions that are committed by the system per second.

Update commit rate The update commit rate is the number of update transactions that are committed by the system per second.

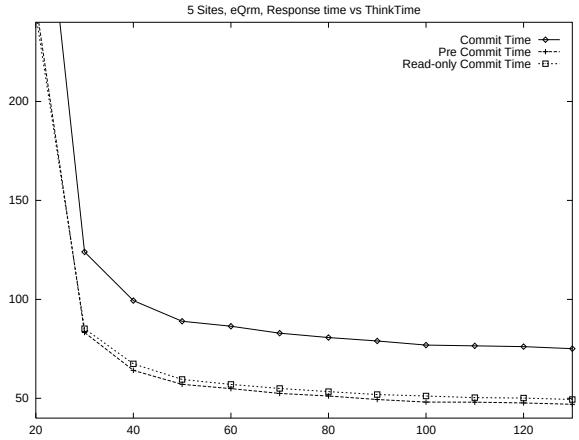
We refer to the epidemic read-one write-all algorithm as eROWA and use it to provide us with a baseline against which to measure the performance of the quorum algorithm, eQrm. A simple majority was used as a quorum. Thus, a transaction can commit when the home site knows that a majority of sites have received t and did not vote *no*. A transaction is aborted when the home site knows that a majority of sites have voted *no* for t. In the case of an even number of sites, a transaction which gets *no* votes from half of the sites is aborted. Other types of quorums, such as the grid quorum are possible and might even have better performance as the number of sites in a quorum would be less, however, majority is the simplest and most general.

4.1 Response Time Analysis

In the response time graphs, both the x- and y-axis are in milliseconds. In a system with five sites, as shown in Figure 2, the read-only commit time is the same for eROWA and eQrm except under heavy system loads indicated by a ThinkTime of less than 30ms. In eROWA, the pre-commit time is greater than the read-only commit time because the update transactions do all of their writes (which take slightly longer than reads) and a forced write of the log disk (8ms) before they can pre-commit. In eQrm, the update transactions acquire all needed locks but do not do the data writes or write to the log disk before pre-committing. Only when an eQrm transaction has enough votes to

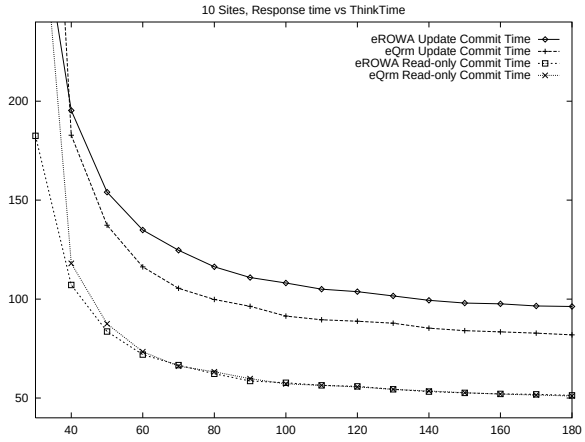


(a) Protocol eROWA

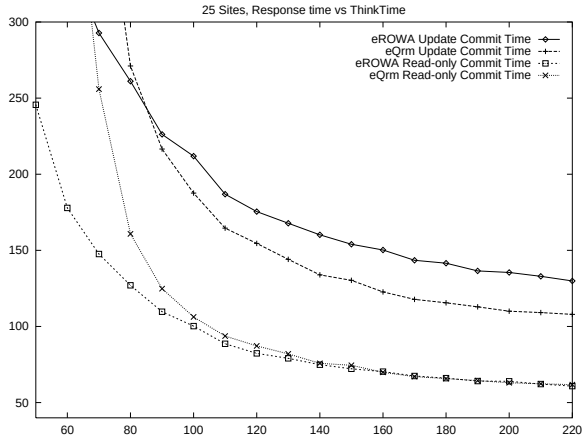


(b) Protocol eQrm

Figure 2: Response time for 5 sites

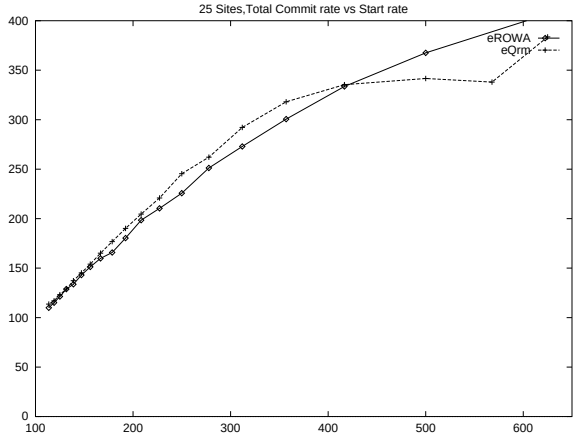


(a) 10 sites

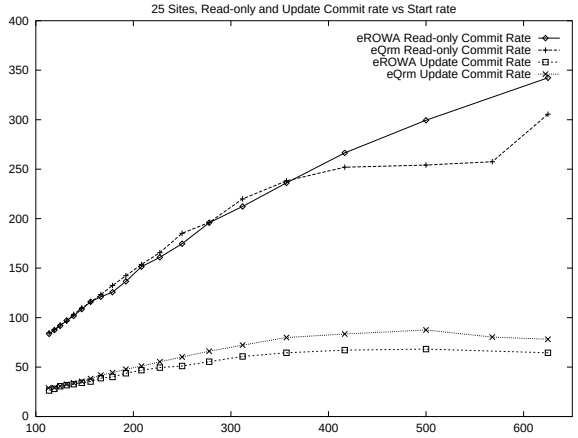


(b) 25 sites

Figure 3: Response times vs ThinkTime



(a) Total Commit Rate



(b) Read-only and Update Commit Rate

Figure 4: Commit rates for 25 sites

commit does it do the data writes and force write to the log disk. Thus, the pre-commit time is less than the read-only commit time for eQrm. An eQrm update transaction only needs to be approved by a majority of the sites rather than all of them for eROWA; this enables the transactions to commit earlier and, as expected, the smaller commit time for eQrm reflects this. Given that the pre-commit response time of update transactions closely tracks the read-only commit time, we drop it from our future graphs.

We then conducted experiments to study the performance of epidemic transaction protocols as the number of sites in the system increases. In Figure 3(a) we show the read-only and update commit times for a 10 site system for eROWA and eQrm. The read-only commit times for the two protocols are very similar, while for update transactions, eQrm does substantially better. The commit time is longer in a 10 site system than in a 5 site system, although the quorum protocol, eQrm, provides substantial benefit. Recall that a ThinkTime of 50ms for 5 sites means that 100 transactions are entering the system per second (5 sites * 1000 / 50ms). This same transaction generation rate would be represented by a ThinkTime of 100ms in a 10 site system.

In Figure 3(b) we report the response time for an epidemic system with 25 sites. In this larger system, two interesting trends start to manifest. First, the eROWA gives better response time to read-only transactions as the system load increases, especially with ThinkTime less than 80ms. In this range, eQrm cannot sustain reasonable response times for read-only transactions. The second trend is that for update transactions, eQrm gives much better response time up until about the same system load. In a heavily loaded system (ThinkTime less than 80ms), the response time for committing update transactions using eROWA out performs eQrm. By analyzing the type of transaction committing, we suspected that eROWA was changing the mix of committed transactions in favor of read-only transactions. We discuss this in detail next.

4.2 Throughput Analysis

To analyze the transaction mix and to assess overall performance, we now discuss the throughput, or commit rate, of the system under varying loads. In Figure 4, the x-axis (the transaction *start rate*) is the number of transactions generated and submitted to the system per second. The total commit rate for eROWA and eQrm is given in Figure 4(a), while in Figure 4(b), we present the update and read-only commit rates.

When the start rate is small, almost all transactions are committed. At a start rate of 150, both eROWA and eQrm commit close to 100% of the transactions (see Figure 4(a)). From a start rate of 250 to 400 transactions per second, the total commit rate for eQrm is better than for eROWA. At 500 transactions started per second, the total commit rate is 73.5% of the transactions started for eROWA and 68.3% for eQrm. At 625 transactions per second, the total commit rate is 65.1% for eROWA and 61.4% for eQrm. Under these heavy system loads, eROWA appears to perform better, however, as we shall see, this improvement in total commit rate comes at the expense of update transactions as eROWA is changing the transaction mix.

In Figure 4(b), we see the read-only and update commit rate for the two protocols. At a transaction start rate of 150 transactions per second, the read-only commit for eROWA is slightly over 75% of the committed transactions. The update commit rate is slightly less than 25%. The read-only commit rate for eQrm is 75% of the committed transactions and the update commit rate is about 25%. Both protocols are maintaining the original transaction mix of 75% read-only and 25% update transactions at this start rate. However, the picture changes as the start rate increases. At a start rate of 250, the update commit rate for eROWA begins to fall off, indicating that a greater proportion of update transactions are being aborted. Thus, already eROWA has begun to favor read-only transactions. Unlike eROWA, eQrm does not favor read-only transactions at the expense of update transactions and at a start rate of 250, eQrm maintains the original balance between update and read-only transactions. When the start rate increases to 500 transactions per second, the difference between the protocols becomes pronounced and eROWA's bias in favor of read-only transactions becomes quite obvious. The total commit rate for eROWA looks good; however, the read-only transaction commits now represent 81.4% of the committed transactions and the update commits are only 18.6%. At a start rate of 500 the read-only transaction commit rate for eQrm is 74.4% of the committed transactions and the update commits are 25.6%. At a start rate of 625, the update commit rate for eROWA actually begins to decrease over the update commit rate at a start rate of 500, whereas, the update commit rate of eQrm is continuing to increase. Thus eQrm does not favor one type of transaction over the other. In fact, at a start rate of 500 transactions per second, the system has reached its thrashing point and performance is starting to degrade due to data contention. At this point, eQrm maintains the percentage of committed update transactions while eROWA simply aborts update transactions. This explains the lower overall response time for eROWA observed in a heavily loaded system in Figure 3.

To further validate our observation that eQrm

maintains the percentage of committed update transactions even in heavily loaded systems, we conducted an experiment where the number of update transactions started was increased to 50% of all transactions. Increasing the proportion of update transactions increases the contention for data and resources at a given transaction start rate. It also gives us the opportunity to see if the eQrm protocol can maintain the transaction mix at 50% update. The results of this experiment are in [6]. As expected, eQrm maintains the mix of committed transactions.

5 Conclusion

Efficient, fault tolerant management of replicated data has been a difficult problem. In this paper we presented an algorithm which uses the epidemic model and quorums to provide a solution to this problem. Our solution is tolerant of communication failures due to the epidemic model, and tolerant of site failures due to quorum commitment. We conducted a detailed simulation study of the transactional epidemic protocols with and without quorums. The experimental results are quite positive and demonstrate the potential for the epidemic approach to support replication with transactional semantics where multiple operations must be treated atomically without designating sites as primary or secondary. Our experiments show that the epidemic quorum approach has several advantages, such as, even for high system load, the mix of submitted transactions matches the mix of committed transactions. There are, of course, tradeoffs that must be made, and in some cases, the simplicity of ROWA which does not need to handle vote records, will be preferred. With the inevitable popularity of E-commerce, reliability of commerce servers is likely to become an important issue. Unlike replication of Web servers, E-commerce will require multi-operational transactions and serializability of those transactions. We strongly believe that asynchronous replication techniques such as the one described in this paper will have a major impact in this area.

References

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

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

[3] Y. Breitbart, R. Komondoor, R. Rastogi, and S. Seshadri. Update Propagation Protocols for Replicated Databases. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, June 1999.

[4] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 150–159, December 1979.

[5] J. Gray, P. Helland, P. 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*, pages 173–182, June 1996.

[6] J. Holliday, R. Steinke, D. Agrawal, and A. El Abbadi. Epidemic Quorums for Managing Replicated Data. Technical Report TRCS 99-32, Department of Computer Science, University of California at Santa Barbara, 1999. http://www.cs.ucsb.edu/TRs/TRCS99-32.ps.

[7] P. Keleher. Decentralized Replicated-Object Protocols. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, April 1999.

[8] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions of Computer Systems*, 10(4):360–391, November 1992.

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

[10] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[11] Oracle. *Oracle7 Server Distributed Systems: Replicated Data.* http: //www. oracle. com /products /oracle7 /server /whitepapers /replication /html /index.

[12] K. Petersen, M. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 288–301, 1997.

[13] M. Livny R. Agrawal, M. Carey. Concurrency Control Performance Modeling: Alternatives and Implications. In V. Kumar, editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems.* Prentice Hall, 1996.

[14] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steer. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[15] G. T. Wuu and A. J. Bernstein. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 233–242, August 1984.