Technical Report TRCS99-32
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

October 15, 1999

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 processes transactions on replicated data. Causal order helps establish the global serialization order on transactions. Our approach enforces serializability by aborting transactions that may cause inconsistency. In the absence of conflict a transaction can commit as soon as it is known to a quorum of sites. In the presence of conflict, sites vote and a transaction can commit as soon as a quorum of sites vote for it. We present a detailed simulation study 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 [12, 33] through gossip messages [21, 17], epidemic propagation, and anti-entropy [11]. In this paper we use the epidemic communication model as the basis for a fault-tolerant 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. Furthermore, 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. When viewed in this manner the epidemic model seems like a natural paradigm to support users in disconnected and mobile environments. These environments suffer from constant re-partitioning, and it may be that no partition ever contains a quorum of sites. By using the epidemic model, updates performed on a single site can eventually reach all other sites in the system.

Earlier protocols based on epidemic communica-

*This research was partially supported by LANL under grant number 6863V0016-3A, by CALTRANS under grant number 65V250A, and by the NSF under grant numbers CDA94-21978 and CCR95-05807.

tion considered single operation applications such as replicated dictionaries. Bayou [26] 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. [19] demonstrate the usefulness of lazy propagation protocols to maintain highly available services in a distributed system. Adya and Liskov [1] successfully employ the notion of lazy propagation for optimistic execution of transactions based on client caches. Other systems such as Coda [29] and Ficus [16] were developed for weakly consistent systems. However, these systems do not support transactional semantics at the epidemic propagation level, rather, single operation semantics. Several database protocols [3, 2] employ epidemic propagation for maintaining replicated databases. However, in these systems the epidemic model is used only for communicating updates, and not for synchronizing updates. Some commercial database systems use lazy propagation, for example, Oracle 7 [25] 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. [15] 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 [6, 9] 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 [4]. 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 log records recording the operations of each transaction, and then 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. Sites rely on causal message delivery to determine when transactions will never be involved in a conflict and thus may commit.

Our approach in this paper is to combine the quorum approach [14] 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. The paper is organized as follows. 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 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. An event model [20] 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 [20] 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 $e \rightarrow f$.

Lamport uses the happened-before relation to define a clock with the following property:

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

We will refer to this clock as Lamport's clock.

Application specific operations are executed locally and they are communicated to the other sites by using the epidemic model of communication which is that information is spread from site to site when two sites communicate and share information. The communication model is such that it 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. Epidemic algorithms generally are implemented using vector clocks [23] to ensure this property. Vector clocks are an extension of Lamport clocks [20] and ensure the following property:

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

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

In the log based approach each site maintains a log of application specific operations. Sites exchange their respective 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 [33].

Wuu and Bernstein [33] combine 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 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 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:

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

The correctness of the algorithm can be established by using both the log and the time-table properties. The use of vector timestamps and time-tables can limit scalability. These issues have been addressed frequently by the distributed systems community. Rabinovich et al. [28] propose a more scalable mechinism than vector timestamps. A variant of the log and time-table algorithm called the two-phase gossip protocol [17] reduces the size of the two-dimensional time-table from n^2 to $2 \cdot n$.

3 Epidemic Transactional Replication

3.1 Epidemic ROWA

In [4] we extended the epidemic model to support the execution of transactions in a fully replicated database. To enforce atomicity the algorithm treats each transaction as a single event with a single log record 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*.

To enforce one-copy serializability the algorithm aborts all concurrent transactions with conflicting operations, hereafter called *conflicting transactions*. A site can detect if two transactions are conflicting because their log records contain their read sets, write sets and version vectors. That is, the condition for conflict is:

$$Conflicting(t,t') \equiv \left[\begin{array}{c} TS(t) <> TS(t') \\ \bigwedge \\ \left(\begin{array}{c} RS(t) \cap WS(t') \neq \emptyset \\ \bigvee \\ WS(t) \cap WS(t') \neq \emptyset \\ \bigvee \\ WS(t) \cap RS(t') \neq \emptyset \end{array} \right) \end{array} \right]$$

If two conflicting transactions attempt to pre-commit on the same site the second one to arrive will find the first, and they will both abort. Thus, a transaction t cannot commit until it knows that there are no conflicting transactions in the system which would cause it to abort.

When S_k receives the message, if the transaction t whose record was sent from S_i to S_k is not aborted due to encountering a conflicting transaction already in the log, 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,

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

3.2 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. However, 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* [14]. Quorums are sets of sites such that the intersection of any two quorums is non-empty. For example, a majority quorum [14] is any set that contains a majority of sites.

We propose to use voting and quorums to resolve commit decisions. This was first proposed by Steinke [31] and is similar to the way they are used by Keleher [18] except our protocol ensures serializability among multi-operation transactions, whereas Keleher treats only single-operation requests. In the epidemic-quorum algorithm we assume complete replication. 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* as a set of sites without which a transaction cannot acquire a quorum. 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. If the set of yes votes constitutes a quorum then t is committed. Otherwise, by definition, the set of no votes constitutes an anti-quorum and t is aborted.

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. 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, and 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.

Lock Held: R	W	IW	Lock Requested
√	×	×	R
×	×	$*_1$	W
$*_2$	$*_2$	√	IW

Figure 1: Lock conflict table for intention to write locks.

To solve this problem we adapt an idea from multi-granularity locking called *intention to write* locks [8]. 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.

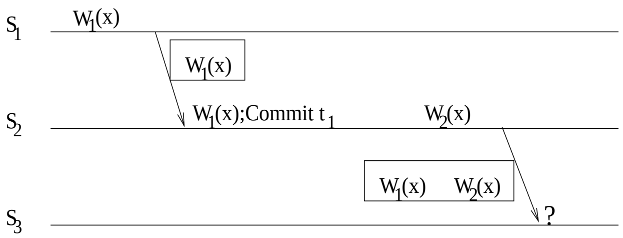


Figure 2: Situation $*_1$.

In this algorithm intention to write locks create two special situations. The first is referred to as $*_1$ in the lock conflict table, and is demonstrated in Figure 2. Transaction t_1 writes data item x on site S_1 and pre commits ($W_1(x)$.) Then the log record for t_1 is transmitted to site S_2 where a remote transaction representing t_1 acquires an intention to write lock on x and pre-commits. Now t_1 is known to two sites (a majority quorum) so it converts its intention to write lock to a write lock, writes x, and commits. Later, t_2 writes x on site S_2 and pre-commits ($W_2(x)$.) Now, when the records of t_1 and t_2 are transmitted to site S_3 they will both acquire intention to write locks on x and pre-commit. When they attempt to commit they will have to convert their intention to write locks to write locks to write their values to x, but this would be prevented by the other transaction's intention to write lock.

The solution to this is to force transactions to commit in causal order even if they are received in the same message. This requirement needs to be enforced anyway to make serialization order based on causal order, and it can be enforced if the order of transactions in a message preserves the order of transactions in the log. This is trivial because messages are generated from the log. In this case if a remote transaction t tries to convert its intention to write lock on a data item x to a write lock, and another remote transaction t' holds a conflicting intention to write lock on x then $t \rightarrow t'$. If $t' \rightarrow t$ then t' cannot be uncertain because its remote transaction on t's initiating site must have committed or aborted before t's local transaction could have acquired a write lock on x. Therefore, the remote transaction for t' must be committed or aborted on this site and cannot be holding an intention to write lock. If t and t' are concurrent, then they must be conflicting because they wrote the same data item. Since t is committing, t' must abort. This only leaves the case where $t \rightarrow t'$. In this case we want t to write before t' so we should allow t to acquire a write lock and commit regardless of conflicting intention to write locks.

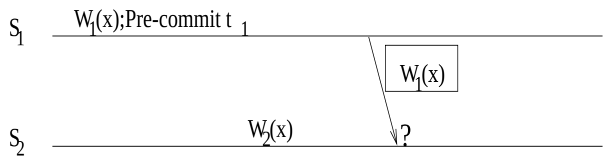


Figure 3: Situation $*_2$.

The second situation is referred to as $*_2$ in the lock conflict table, and is demonstrated in Figure 3. Transaction t_1 writes x on S_1 and pre-commits. Concurrently, Transaction t_2 writes x on S_2, but a message containing the log record of t_1 arrives before t_2 pre-commits. Now, t_1 needs to acquire an intention to write lock on x at S_2, but t_2 holds a conflicting write lock. If t_1 were to wait for the lock then t_2 would pre-commit and convert its write lock to an intention to write lock, and t_1 and t_2 would be concurrent and thus conflicting. In this case it makes sense to abort t_2 right away and give the intention to write lock to t_1. If the transactions become conflicting one would have to be aborted anyway, and at this point t_2 has less work invested so far because it hasn't been transmitted to another site. This situation also applies to read locks held by local transactions.

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.

3.3 Implementing the Epidemic Quorum Algorithm

Figure 4 illustrates the data structure used to represent transaction records. The elements of the record are the transaction's read set (RS), write set (WS), values written ($values$), the transaction's initiating site ($site$), and the version vector of the initiating site at the time its local transaction pre-commits ($time$). $Transaction.time$ is a vector timestamp to distinguish concurrent transactions from causally related ones, but the scalar value of the initiating site's clock is used to uniquely identify the transaction for the $HasRecvd$ predicate and vote records. This value is stored in $transaction.time[transaction.site]$.

```
type Transaction=
  record
    RS     : setof DataObjectType;
    WS     : setof DataObjectType;
    values : setof DataType;
    site   : SiteId;
    time   : array [1..n] of TimeType;
  end
```

Figure 4: A Transaction Record

Figure 5 illustrates the data structure used to represent vote records. A vote record uniquely identifies the transaction being voted on by its initiating site ($Tsite$), and its scalar timestamp ($Ttime$). It also contains the voting site ($site$), the voting site's local time when the vote record was created ($time$), and indicates whether the vote is yes or no ($VoteYes$). If $VoteYes$ is true then the vote is a yes vote. If it is false then it is a no vote. Concurrency among vote records is unimportant so the $time$ field of a vote record is a scalar value used only in the $HasRecvd$ predicate and for log management to determine whether a site needs to send the vote record to another site. Vote records are communicated in the same epidemic messages as transaction records and sites use a single time table for this purpose, but preserving causality between vote records is not necessary.

```
type Vote =
  record
    Tsite   : SiteId;
    Ttime   : TimeType;
    site    : SiteId;
    time    : TimeType;
    VoteYes   Flag;
  end
```

Figure 5: A Vote Record

The algorithm for processing a transaction at its initiating site is given in Figure 6. Each site keeps a scalar local clock ($clock_i$), a two-dimensional timetable (T_i), a log of transaction records (L_i), and a log of vote records (V_i). Our algorithm has separate logs for transaction and vote records for ease of explanation, however they do not necessarily need to be kept in separate data structures. A transaction starts execution at its initiating site by acquiring locks, performing computation, and writing values to the database. Then, before the transaction pre-commits it puts a record in the transaction log describing the transaction, and a record in the vote log indicating that the initiating site has voted yes for this transaction. The site's local time is updated for each record, and the transaction acquires a mutex to prevent conflicting access to the clock, time-table, and logs. Finally, the transaction releases its read locks, and converts its write locks to intention to write locks. After this point the transaction is processed identically at all sites as a remote transaction for the purpose of detecting conflict and committing the transaction. This is necessary in case a conflicting transaction arrives, and both transactions are required to simultaneously hold intention to write locks.

Persistent data:
  $clock_i$ $TimeType$ INITIALLY 0;
  T_i : array $[1..n, 1..n]$ of $TimeType$ INITIALLY 0;
  L_i : setof $Transaction$ INITIALLY $\emptyset$;
  V_i : setof $Vote$ INITIALLY $\emptyset$;

Transaction(RS, WS, f(x)):
begin
  $GetReadLocks(RS)$;
  $values := f(\text{read } (RS))$;
  $GetWriteLocks(WS)$;
  $WriteValues(WS, values)$;
  begin mutex
    $T_i[i,i] := ++\, clock_i$;
    $t := \langle RS, WS, values, i, T_i[i,*]\rangle$;
    $L_i := L_i \cup \{t\}$;
    $T_i[i,i] := ++\, clock_i$;
    $V_i := V_i \cup \{\langle t.site, t.time[t.site], i, T_i[i,i], true\rangle\}$;
    $Pre\text{-}Commit$;
  end mutex
  $ReleaseReadLocks(RS)$;
  $ConvertToIntentionToWriteLocks(WS)$;
end;

Figure 6: The Epidemic Algorithm for Executing Transactions at S_i

The algorithm for sending log records, and processing received log records is given in Figures 7 and 8. The send procedure acquires a mutex to prevent a local transaction from accessing the time-table or logs, and then sends all transaction and vote records for which the $HasRecvd$ predicate is false. Transactions in the local log are placed in causal order, and the send procedure preserves this order so that the receiving site can process the message's records in that order and be guaranteed that it is processing transactions in causal order. Vote records can be sent in any order. Vote records are processed after all transactions in an epidemic message, and the ordering of two vote records is irrelevant.

Send(m) $to\ S_k$:
begin
  begin mutex
    $SP := \{t | t \in L_i \wedge \neg HasRecvd(T_i, t, S_k)\}$;
    $VP := \{v | v \in V_i \wedge \neg HasRecvd(T_i, v, S_k)\}$;
    send $\langle SP, VP, T_i\rangle\ to\ S_k$;
  end mutex
end;

Figure 7: Epidemic propagation of transaction records from S_i to S_k

The receive procedure has two steps. First, it processes transaction records one at a time, and then it processes all vote records and checks for transaction commit or abort. The receive procedure starts at the beginning of the set of transaction records and looks for the first record that it has not already received. When it finds such a record it acquires the mutex to prevent interference from local transactions. The reason for this is more complicated than just preventing simultaneous access to the log or time table. Consider the following situation: The receive procedure processing transaction t checks the log and finds no conflicting transactions so it decides to vote for t. Then a conflicting local transaction l acquires locks and pre-commits. When l pre-commits, all transactions in the log are causally preceding so there can be no conflicting transaction and the site votes for l. However, t is not yet in the log, and it is conflicting, but the receive procedure will not check the log again so it will vote for both t and l. To prevent this from happening the receive procedure prevents local transactions from pre-committing between the time that t checks the log and the time that it inserts its log record and updates the time table.

Once t has the mutex it checks the log for

```
Receive(m) from S_k :
begin
    let m = ⟨SP_k, VP_k, T_k⟩ ;
    foreach {t | t ∈ SP_k ∧ ¬HasRecvd(T_i, t, S_i)} do
        begin mutex
            if {∃t' ∈ L_i | (Conflicting(t, t') = true)∧
              (∃v' ∈ V_i | v' = VoteFor(t', i) ∧ v'.VoteYes = true)} then
                T_i[i, i] := ++clock_i;
                V_i := V_i ∪ {⟨t.site, t.time[t.site], i, T_i[i, i], false⟩};
            else
                T_i[i, i] := ++clock_i;
                V_i := V_i ∪ {⟨t.site, t.time[t.site], i, T_i[i, i], true⟩};
            endif
            GetIntentionToWriteLocks(t.WS);
            T_i[i, t.site] := t.time[t.site];
            L_i := L_i ∪ {t};
            Pre-Commit(t);
        end mutex
    endfor
    begin mutex
        ∀K ≠ i, J T_i[K, J] := max(T_i[K, J], T_k[K, J]);
        V_i := V_i ∪ VP_k;
        foreach t ∈ L_i do
            if Abortable(t, S_i) then
                ReleaseIntentionToWriteLocks(t.WS);
                Abort(t);
            elseif Committable(t, S_i) then
                ConvertToWriteLocks(t.WS);
                WriteValues(t.WS, t.values);
                ReleaseWriteLocks(t.WS);
                Commit(t);
            endif
        endfor
        L_i := {t | t ∈ L_i ∧ ∃j | ¬HasRecvd(T_i, t, S_j)};
        V_i := {v | v ∈ V_i ∧ ∃j | ¬HasRecvd(T_i, v, S_j)};
    end mutex
end;
```

Figure 8: Epidemic propagation of transaction records from other nodes to S_i

conflicting transactions. In Figure 8, $v = VoteFor(t, i)$ if site i's vote for transaction t is recorded in vote record v. If the site has already voted yes on a conflicting transaction then the site votes no on t. Otherwise, the site votes yes on t. Then the site acquires intention to write locks on all data items in its write set. Because of the special cases in the intention to write lock conflict table this operation will always succeed without waiting. Transaction t then updates the time-table and log and pre-commits. Then the receive procedure releases the mutex before it handles the next transaction. Any non-conflicting local transactions waiting for the mutex can now pre-commit. In terms of causality this means messages are not received all at once. Messages are received one transaction at a time, and local transactions can execute in between two transactions from the same message. This still preserves correctness because transactions in a message are received in causal order, and the log and time table are updated incrementally for each transaction.

When all transactions in an incoming epidemic message have been processed the receive procedure updates the other rows of its time-table to update its knowledge of other sites' version vectors. Updating this information after processing transactions is correct because it is merely a lower bound on other sites' version vectors. This is not true for its own row which is used to create version vectors for local transactions. Those values are updated incrementally as each transaction was processed. Then the receive procedure adds all vote records in the epidemic message to its vote log, and checks each uncertain transaction to see if it can be committed or aborted. Testing for commit or abort of transactions must be done in causal order which is possible because the log preserves the causal order on transactions.

Finally, the site deletes all transaction and vote records that it knows are known to all sites. This is allowed because if S_i knows that S_j has received transaction t then S_i must have received S_j's vote on t. So when S_i knows that all sites have received t then S_i must have all vote records for t and must have committed or aborted t so the record for t is not needed locally. Likewise, t causally precedes all votes on t so if S_i knows that all sites have received a vote on t it knows all sites have received t and S_i must have committed or aborted t, and the vote record is no longer needed locally. When S_i knows that a record, either transaction or vote, is known to all sites that record will never be included in any message and thus it will not be needed for any purpose and can be deleted.

3.4 Read-Only Transactions

There has been a lot of work on increasing concurrency and fault tolerance for read-only transactions by reducing the number of sites they need to access [13, 10, 32, 7, 30]. In our protocol, read-only transactions can also be executed locally without the need global synchronization. To see that this is possible consider all of the committed update transactions in the system and a single read-only transaction t. All update transactions must be serializable, and t can be serialized just after the last transaction in the serialization order from which it read. No transaction concurrent to t and writing a data item read by t can be serialized before this point.

If t read x from transaction u, and transaction v wrote x and is concurrent to t then $u \to v$. v cannot have happened before u, or it would also have happened before t, and v cannot be concurrent to u or they would be conflicting and one would have aborted. If a fourth transaction w wrote a data item y also read by t then either w happened before v or w and v are concurrent and non-conflicting. If v also accessed y then $w \to v$ by the above argument. If not then w and v are non-conflicting and their serialization order can be reversed. So an equivalent total order is $w \to t \to v$. Since this can be done for all transactions concurrent to t the history is equivalent to t being serialized after all transactions that causally precede it, and before all transactions concurrent to it.

When more than one read-only transaction is included there at first appears to be a problem. Consider a database with data item copies $x_1, x_2, ..., x_n$

and $y_1, y_2, ..., y_n$. S_1 performs $t_1 : W(x_1)$, and S_2 performs $t_2 : W(y_2)$. Now S_1 communicates with a quorum of sites not including S_2, and S_2 communicates with a quorum not including S_1. The transactions are not conflicting so they both receive a quorum of yes votes and commit. Now S_1 performs $t_3 : R(x_1), R(y_1)$, and S_2 performs $t_4 : R(x_2), R(y_2)$. One might think that t_3 reads from t_1 and not t_2 while t_4 reads from t_2 and not t_1 causing a serialization conflict, but this is not the case. When t_1 and t_2 were each received by a quorum of sites there must have been at least one site, S_i, that received both. Consider without loss of generality that S_i received t_2 second. In order for S_2 to commit t_2 and release its locks S_2 must have received S_i's vote, and S_i's vote for t_2 causally succeeded S_i's receipt of t_1. Therefore, t_1 must be received by S_2 before S_2 can commit t_2. Since the receive procedure processes all transactions of a message before processing any vote records from that message t_1 must have been precommitted on S_2 before t_2 committed on S_2. If t_4 held a read lock on y_2 when t_1 arrived then t_4 would be aborted and restarted when t_1 acquired an intention to write lock. If t_4 requests a read lock after t_1 arrives it will wait on t_1's intention to write lock. Either way, t_4 must wait for the read lock on x_2 until t_1 commits on S_2. Therefore, t_4 reads x from t_1 and the serialization order is $t_1 \rightarrow t_3 \rightarrow t_2 \rightarrow t_4$.

```
Read-Only(RS, f(x)):
begin
    GetReadLocks(RS);
    values := f(read (RS));
    ReleaseReadLocks(RS);
    Commit;
    Return(values);
end;
```

Figure 9: The Epidemic Algorithm for Executing Read-Only Transactions at S_i

This protocol relies on the fact that the database is fully replicated so the quorums of any two transactions will overlap even if the transactions do not conflict. Therefore, concurrent non-conflicting transactions can be serialized in the order in which they become committable. If a transaction $t1$ gets a quorum of yes votes before a transaction $t2$ then every site will have received $t1$ by the time it commits $t2$, and no read-only transaction can read $t2$ without being serialized after $t1$.

The algorithm for processing read-only transactions is given in Figure 9. The read-only transaction acquires read locks, reads values, and releases read locks. Then the transaction can commit and return the values read. If any conflicting transaction arrives while the read-only transaction is waiting for read locks then the read-only transaction is aborted. Otherwise the read-only transaction will not violate serializability as explained in the previous section and it can commit.

3.5 Optimizations

The first optimization avoids explicit yes votes. The quorum algorithm just presented creates a vote record for each update transaction at each site. It is possible to reduce the number of vote records that must be stored at each site and transmitted among sites. If a transaction t pre-commits on every site without encountering any conflicting transactions, then obviously every site votes yes for t. Therefore, a site does not have to create vote records for t until it learns of a conflicting transaction t'. Also, sites do not need to record and send out both yes and no votes. A site S_i can send only its no votes and the other sites can infer that S_i voted yes if it is known (using the time table) that S_i received the transaction and S_i did not send out a no vote. This will work if S_i records its no vote for t before it updates its time table indicating it has received t. Then when S_i sends its time table to S_k, S_k knows that S_i has received and voted on t and the vote is yes unless S_k has received an explicit no vote from S_i for t. Thus S_i can infer that S_k voted yes on t if:

$$HasRecvd(T_i, t, S_k) \wedge \neg\{\exists v \in V_i \mid v = no\}$$

Because we are interested in performance and it was thought that this optimization would scale better to a large number of sites, this optimization was used in the performance evaluation.

The second optimization is not forcing write locks by aborting local un-pre-committed transactions. In the quorum protocol, when a pre-commit transaction record is received at a site, all conflicting local transactions that have not yet pre-committed are aborted and the incoming remote transaction is given the locks. These local transactions may have been needlessly aborted if the remote transaction fails to get enough yes votes and is aborted. With this in mind, the quorum algorithm could be modified so that the incoming remote transaction waits for the locks and only aborts local transactions when and if it receives sufficient votes to commit. This modification preserves correctness since all it does is delay the decision to abort some conflicting transaction. We previously argued that the quorum protocol does not need global deadlock detection because remote transactions are never involved in a deadlock since they never have to wait for a lock. For this new modification, we still do not require distributed deadlock detection. If two conflicting remote transactions request locks at different sites in a different order, the conflict will be resolved when one of them is aborted due to voting. The one that is allowed to commit will not be stopped by the other that holds the lock and has not yet aborted, because it is only an intention-to-write lock. Any deadlock that might be detected locally must include a local non-pre-committed transaction which can be aborted. To see that you cannot have a deadlock consisting only of remote transactions, recall that remote transactions request all of their locks at once. Similarly, it is not possible to have a deadlock consisting only of remote transactions and local transactions that have already pre-committed since in order for a transaction to pre-commit, it must be holding all the locks it needs. Since there must be a local non-pre-committed transaction in any deadlock, we can select one of them as the victim and abort it. We did preliminary performance measurements without this optimization and discovered that it was helpful in maintaining a good overall throughput.

3.6 An Example

An example execution is illustrated in Figure 10. The arrows indicate messages sent from one site to another, and the boxes indicate the contents of those messages. In this example there are three sites, and any two sites constitute a majority quorum. Each site performs one transaction. Transaction t at S_1 reads and writes x. Transaction u at S_2 reads and writes x and y. Transaction v at S_3 reads and writes y. Transactions t and u conflict as well as transactions u and v, but t and v do not conflict. So either t and v should commit or just u should commit.

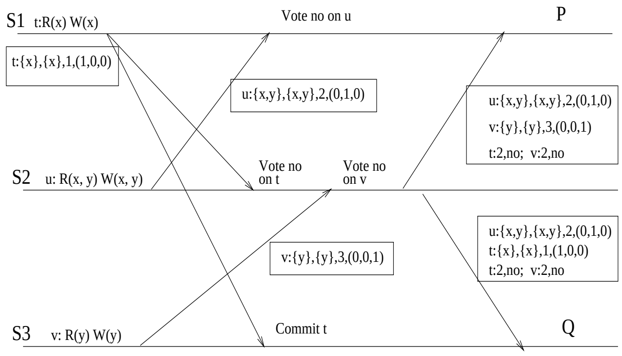


Figure 10: Example history.

First, S_1 sends epidemic messages to S_2 and S_3, S_2 sends an epidemic message to S_1, and S_3 sends an epidemic message to S_2. Each message contains a transaction record and the time table of the sender (which is not shown). The figure shows the read set, write set, initiating site, and version vector of the transaction records, and the voting site and vote value of the vote records. For example, the message from site S_1 contains the transaction record $t : x, x, 1, (1, 0, 0)$ which means that transaction t read x, wrote x, initiated at site S_1, and has version vector $(1, 0, 0)$. There is no explicit vote record as the receiver of the message will know that site S_1 voted yes on t because of the absence of a no vote record in the message. When these messages are received the sites compare log records. Site S_1 votes no on u because it implicitly voted yes on t. It now knows of two transactions with one yes vote each so neither can commit. Site S_2 votes no on t and v because it voted yes on u. It now knows of three transactions with one yes vote each so none can commit. However, when S_3 receives the message from S_1 sees that t and v do not conflict. Now S_3 knows that two sites, S_1 and S_3, know about t and did not vote no, so there are two yes votes. This constitutes a quorum so it commits t.

Now, S_2 sends messages to S_1 and S_3. S_2's log contains three transaction records, t, u, and v, and two vote records, $t : 2, no$ and $v : 2, no$. S_2 does not send the record for t to S_1 because it knows that S_1 already has it, but it sends its vote record for t. It also sends the transaction record for u that it sent in a previous message because this message might have been lost and the transaction and vote records for v. Likewise, it does not send the record for v to S_3, but it does send its vote for v. S_2 also sends the transaction and vote record for t and the transaction record for u. We now describe what happens at points P and Q.

At point P, S_1 has already received the record for u so it ignores it. When it processes the record of v it votes yes on v because it voted no on u and t is non-conflicting with respect to v. Now it processes vote records. The no vote from S_2 on t is added to the log, but S_1 still can't commit or abort t. When S_1 adds the vote record for v, however, it can infer two yes votes for v, one from S_3 and one from itself, so it commits v, and aborts the conflicting u.

At point Q, S_3 processes u and votes no because it voted yes for both t and v. S_3 also aborts u because it has committed t. S_3 has already received the record of t so it is ignored. S_3 records that S_2 voted yes on u, and no on t and v. S_3 now has one yes and one no for v so it cannot be committed or aborted. S_3 receives a no vote on t, but this has no effect because it already committed t when it received two yes votes.

At this point every site has seen every transaction, but sites do not know all of the votes. Eventually, due to further epidemic communication, these vote records will propagate through the network. As a result every site will commit t and v and abort u. In the meantime, x and y will be inaccessible at any site where transactions are still uncertain. Any new transaction wishing to access x or y will have to wait until the corresponding lock is released. The new transaction will read the correct committed value and will be causally ordered after t, u, and v so that it can not interfere with their serialization order.

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 [27, 22] and uses the simulation package CSIM18 [24]. All measurements in these experiments were made by running the simulation until a 95% confidence interval was achieved for each data point.

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 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 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 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. These 1000 pages could represent the "hot spots" of a larger database. 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

Parameter	*Meaning*	Value
ThinkTime	Transaction interarrival time	varies
NumSites	Number of sites	5, 10, 25
ER	Epidemic rate	3ms
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	0.3 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
LockTime	CPU time needed to handle locks for read or write request	0.006 ms

Table 1: System Parameters

locking to ensure local serializability. 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.

Parameter	*Meaning*	Value
ReadSetMin	Smallest transaction read set size	5
ReadSetMax	Maximum read set size	11
ROPercent	Percentage of read only transactions	75, 50%
WriteSetMin	Smallest write set size for update transactions	1
WriteSetMax	Maximum write set size	4
CPUPgTime	CPU time spent on a data page	1.0 ms
IntTime	Time between successive operation requests	3 ms

Table 2: Transaction Parameters

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 long or short. There is no multiprogramming level to limit the number of active transactions vying for resources and data item locks. This approach corresponds to the "open" queuing model and is appropriate for a system accepting requests from an unbounded number of users. 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. The execution of a read-only transaction is completely localized. 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.*

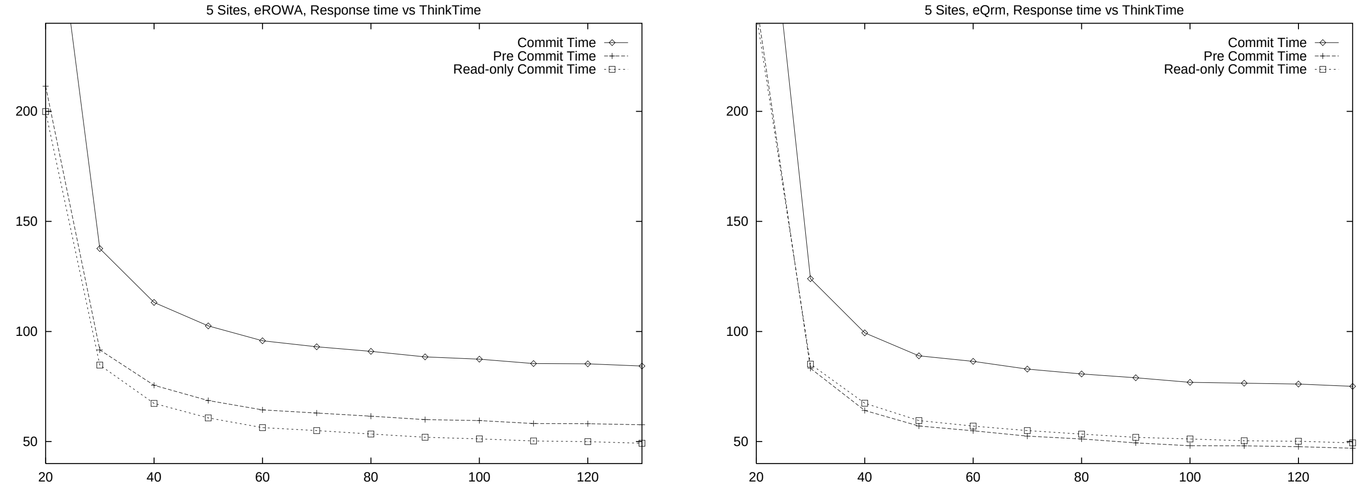
The simulation model assumes that the network is fully connected so that any site can exchange messages directly with any other site. We assume that the network is fast (100 Mbits/sec). On a 100 M bit/sec network, the amount of CPU time needed to send or receive a message was set to 0.1ms. When a site 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 sim-



(a) Protocol eROWA

(b) Protocol eQrm

Figure 11: Response time for 5 sites

ply 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 inter-arrival 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.

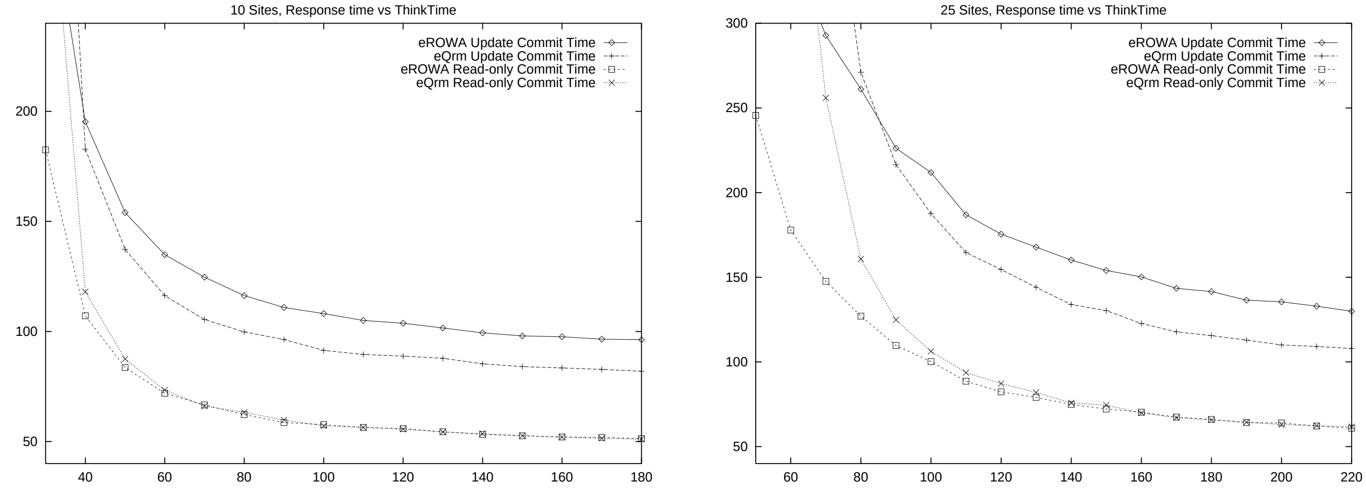
Read-only commit rate The read-only commit rate is the number of read-only transactions that are committed by the system per second.

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, which includes all of the optimizations described. 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.

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 11, 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



(a) 10 sites

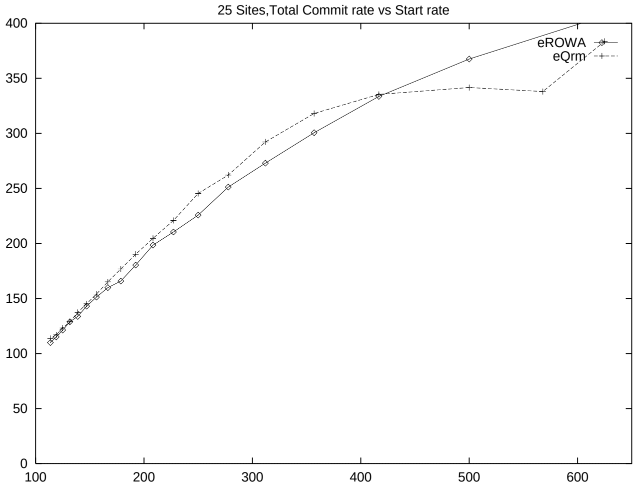
(b) 25 sites

Figure 12: Response times vs ThinkTime

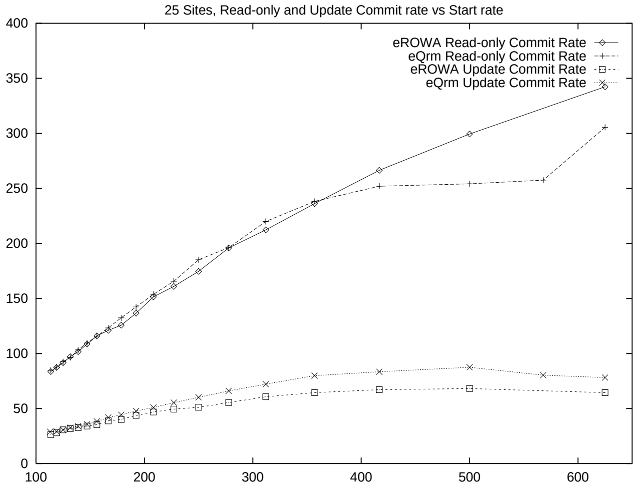
pre-commit. In eQrm, the update transactions acquire all needed locks but do not do the data writes or write to the log disk before pre-committing. Only when an eQrm transaction has enough votes to commit does it do the data writes and force write to the log disk. Thus, the pre-commit time is less than 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 12(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 12(b) we report the response time for an epidemic system with 25 sites. It is important to keep in mind that a transaction generation rate of 125 transactions per second would be represented by a ThinkTime of 80ms in a 10 site system and a ThinkTime of 200 in a 25 site system. In this larger system, two interesting trends start to manifest. First, that 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. For example, at a ThinkTime of 130ms, for eROWA an average of 88ms elapse between the time update transactions are pre-committed and the time they are committed. This time differential, which is a measure of the time needed to propagate the pre-commit records throughout the system and collect enough



(a) Total Commit Rate



(b) Read-only and Update Commit Rate

Figure 13: Commit rates for 25 sites

information to commit the transaction, is 65ms for eQrm. Thus eQrm provides 26% improvement in update commit time. 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 in more detail the throughput, or commit rate, of the system under varying loads. In Figure 13, 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 13(a), while in Figure 13(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 150 transactions per second (see Figure 13(a)). In particular, at a start rate of 156.2, the total commit rate is 151.2 transactions per second or 96.7% for eROWA and 154.0 transactions per second or 98.6% for eQrm. At a start rate of 250, the total commit rate for eROWA is 225.6 or 90.2%. The total commit rate for eQrm is 245.4 or 98.1%. So far, eQrm is clearly superior and it continues to be better up to a start rate of 400 transactions per second.

At 500 transactions started per second, the total commit rate is 367.5 (73.5%) for eROWA and 341.6 (68.3%) for eQrm. At 625 transactions per second, the total commit rate is 406.7 (65.1%) for eROWA and 383.6 (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 13(b), we see the read-only and update commit rate for the two protocols. At a transaction start rate of 156.2 transactions per second, the read-only commit for eROWA is 116 representing slightly over 75% of the committed transactions. The update commit rate is 35 transactions per second which is slightly less than 25%. The read-only commit rate for eQrm is 114.0 representing 75% of the committed transactions. The update commit rate is 37.4 transactions per second which 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 read-only commit for eROWA is 174.6 representing 77% of the committed transactions. The update commit rate is 51.1 which is only 22.6%. 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. At a start rate of 250, the read-only commit is 185.5 representing 75.4% of the committed transactions. The update commit rate is 60.2 transactions per second which is 24.5%. Thus, eQrm maintains the original balance between update and read-only transactions whereas eROWA was beginning to favor read-only transactions at this rate.

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 is 367.5; 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 254.1 representing 74.4% of the committed transactions and the update commits are 87.5 or 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.

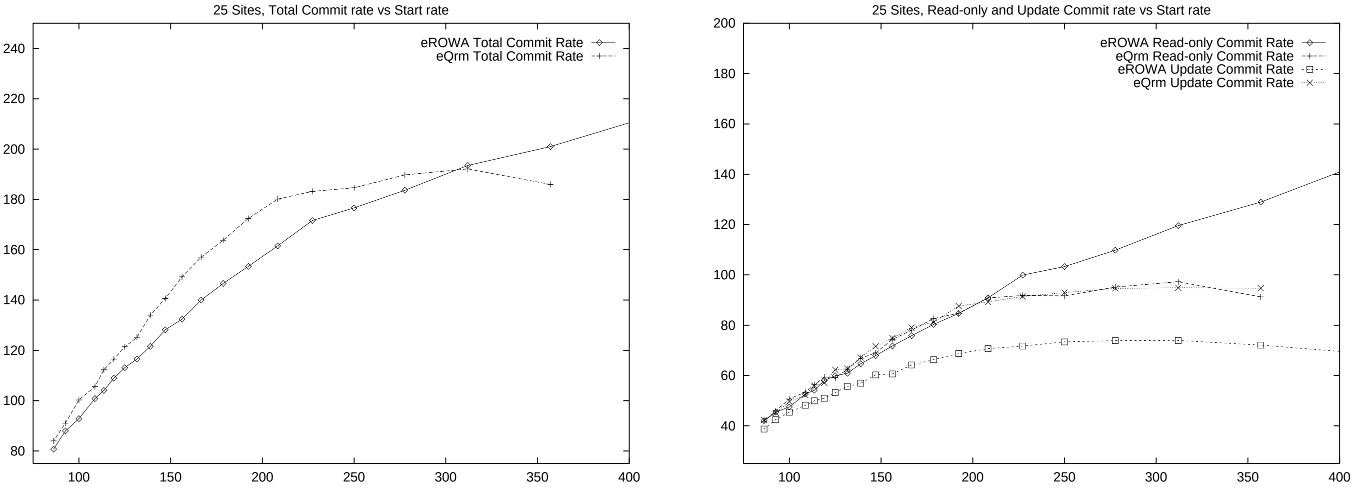
Thus eQrm does not favor one type of transaction over the other. In fact, at a start rate of 500 transactions per second, the system has reached its thrashing point and performance is starting to degrade due to data contention. At this point, eQrm maintains the percentage of committed update transactions while eROWA simply aborts update transactions. This explains the lower overall response time for eROWA observed in a heavily loaded system in Figure 12.

4.3 Increasing the Proportion of Update Transactions

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 was increased to 50% of all transactions. Increasing the proportion of update transactions will increase the contention for data and resources at a given transaction start rate. It will also give us the opportunity to see if the eQrm protocol can maintain the transaction mix at 50% update. The total commit rate for eROWA and eQrm when there are 50% read-only transactions and 50% update transactions is given in Figure 14(a). Encouragingly, the total commit rate achieved by eQrm is superior to eROWA at transaction start rates up to 300 transactions per second. Furthermore, eQrm maintains the mix of committed transactions at about 50% update and 50% read-only (see Figure 14(b)). Above that rate, eQrm begins thrashing and total throughput decreases. The total commit rate for eROWA continues to increase, sacrificing update transactions as can be seen in Figure 14(b). Read-only transactions are "easier" to commit because they share locks with each other and use only the local disk and CPU resources.

4.4 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



(a) Total Commit Rate

(b) Read-only and Update Commit Rate

Figure 14: Commit rates for 25 sites, 50% Update

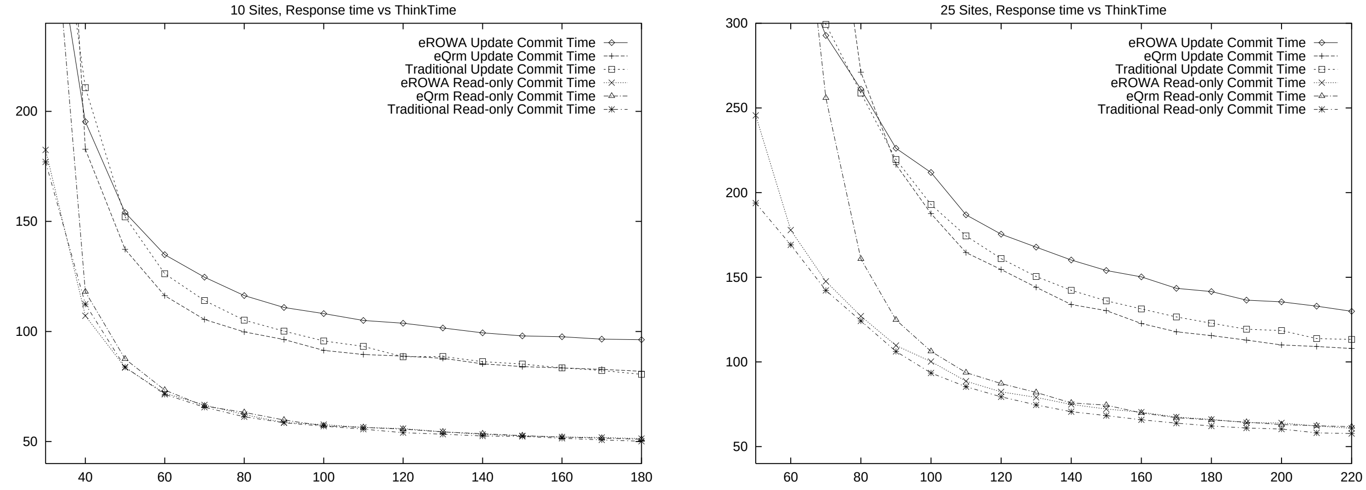
sites, it lets the transaction perform the data write and proceed with its next operation. When the transaction has completed all its operations, the home site DBMS starts an atomic commit protocol such as a two phase commit [8].

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 [5] 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).

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 15(a)) and 25 sites (Figure 15(b)). The response times for the traditional approach are similar to the epidemic approach for read-only transactions.

The results for update transactions are more interesting. At low system load, the effects of data and resource contention are 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). As can be seen in Figure 15, the traditional protocol outperforms eROWA for update transactions in both 10 and 25 site systems. The epidemic quorum protocol, eQrm, however, outperforms the traditional protocol for 25 sites at a ThinkTime greater than 80 ms (a heavy load for 25 sites - over 300 transactions started per second) and for 10 sites at a ThinkTime less than 120 ms.

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



(a) 10 sites

(b) 25 sites

Figure 15: Response time compared to Traditional Protocol

home site to commit the transaction. A remote site forces its log later when it commits the transaction.

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 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 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.

5 Conclusion

Efficient, fault tolerant management of replicated data has been a difficult problem especially in the context of disconnected and mobile environments. 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. In the quorum based epidemic protocol, quorums are used for synchronization, thus allowing update transactions to commit once a majority of sites are aware of the update. However, updates are still propagated to all copies, making local execution of read-only transactions possible. Furthermore, our experiments show that the epidemic quorum approach has several advantages:

- Even for high system load, the mix of submitted transactions matches the mix of committed transactions.
- The overhead of replication, in terms of response time, does not increase significantly as the number of copies increases.
- The system throughput and especially that of committed update transactions is maintained up to the thrashing point and is better than the read-one write-all approach.

Hence, we believe that the epidemic quorum approach holds the promise of efficiently supporting replicated databases which must preserve full serializability in an update anywhere environment. 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] A. Adya and B. Liskov. Lazy Consistency Using Loosely Synchronized Clocks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, August 1997.

[2] D. Agrawal and A. J. Bernstein. A Nonblocking Quorum Consensus Protocol for Replicated Data. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):171–179, April 1991.

[3] D. Agrawal and A. El Abbadi. Storage Efficient Replicated Databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):342–352, September 1990.

[4] 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.

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

[6] T. Anderson, Y. Breitbart, H.F. Korth, and A. Wool. Replication, consistency and practicality: Are these mutually exclusive? In *Proceedings of the ACM SIGMOD International Conference of Management of Data*, pages 484–495, June 1998.

[7] P. Aristedes and A. El Abbadi. Fast Read-Only Transactions in Replicated Databases. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 246–253, February 1992.

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

[9] 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.

[10] A. Chan and R. Gray. Implementing Distributed Read-Only Transactions. *IEEE Transactions on Software Engineering*, 11(2):205–212, February 1985.

[11] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 1–12, August 1987.

[12] M. J. Fischer and A. Michael. Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. In *Proceedings of the First ACM Symposium on Principles of Database Systems*, pages 70–75, May 1982.

[13] H. Garcia-Molina and G. Weiderhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems*, 7(2):209–234, June 1982.

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

[15] 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.

[16] R.G. Guy, J.S. Heidemann, W. Mak, Jr T.W. Page, G.J. Popek, and D. Rothmeier. Implementation of the Ficus File System. In *Proceedings Summer USENIX Conference*, pages 63–71, June 1990.

[17] A. A. Heddaya, M. Hsu, and W. E. Weihl. Two Phase Gossip: Managing Distributed Event Histories. *Information Sciences: An International Journal*, 49(1,2,3):35–57, October/November/December 1989. Special issue on Databases.

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

[19] 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.

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

[21] B. Liskov and R. Ladin. Highly Available Services in Distributed Systems. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 29–39, August 1986.

[22] M. Livny M. Carey. Conflict Detection Tradeoffs for Replicated Data. In V. Kumar, editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice Hall, 1996.

[23] 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.

[24] Mesquite Software, inc., 8500 North Mopac, Suite 825, Austin, TX 78759. *The CSIM Simulation Engine.*

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

[26] 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.

[27] 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.

[28] M. Rabinovich, N. H. Gehani, and A. Kononov. Scalable update propagation in epidemic replicated databases. In *Proceedings of the International Conference on Extending Data Base Technology*, pages 207–222, 1996.

[29] 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.

[30] O. T. Satyanarayanan and D. Agrawal. Efficient Execution of Read-Only Transactions in Replicated Multiversion Databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):859–871, October 1993.

[31] Robert C. Steinke. Epidemic Transactions for Replicated Databases. Master's thesis, University of California at Santa Barbara, Department of Computer Science, UCSB, Santa Barbara, CA 93106, 1997.

[32] W. E. Weihl. Distributed Version Management of Read-only Actions. *IEEE Transactions on Software Engineering*, 13(2):55–64, January 1987.

[33] 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.