Epidemic Algorithms for Replicated Databases

JoAnne Holliday, *Member*, *IEEE*, Robert Steinke, Divyakant Agrawal, *Member*, *IEEE*, and Amr El Abbadi, *Member*, *IEEE*

Abstract—We present a family of epidemic algorithms for maintaining replicated database systems. The algorithms are based on the causal delivery of log records where each record corresponds to one transaction instead of one operation. The first algorithm in this family is a pessimistic protocol that ensures serializability and guarantees strict executions. Since we expect the epidemic algorithms to be used in environments with low probability of conflicts among transactions, we develop a variant of the pessimistic algorithm which is optimistic in that transactions commit as soon as they terminate locally and inconsistencies are detected asynchronously as the effects of committed transactions propagate through the system. The last member of the family of epidemic algorithms is pessimistic and uses voting with quorums to resolve conflicts and improve transaction response time. A simulation study evaluates the performance of the protocols.

Index Terms—Database replication, distributed databases, epidemic communication.

1 Introduction

With the proliferation of computer networks, PCs, and workstations, new models for workplaces are emerging. In particular, organizations need to provide ready access to corporate information to users who may be geographically remote and to handle a volume of access requests that might be too high for a single server. One way to provide access to such data is through replication. However, traditional synchronous solutions for managing replicated data usually handle only a small number of replicas and are not suitable for a widely distributed environment where the network connecting the replicas may be unreliable, not fully connected, and subject to intermittent delays. As the need for replication grows, several vendors have adopted asynchronous solutions for managing replicated data [22], [29]. For example, Lotus Notes uses value-based replication in which updates are performed locally and a propagation mechanism is provided to apply these updates to other replica sites. In addition, a version number is used to detect inconsistencies and resolution of inconsistencies is left to the users. Although the Lotus approach works reasonably well for single object updates (i.e., environments such as file-systems), it fails when multiple objects are involved in a single update (i.e., transaction-oriented environments). In particular, more formal mechanisms are needed for update propagation and conflict detection in asynchronous replication of databases.

Asynchronous replication has been deployed successfully for maintaining control information in distributed systems and computer networks. For example, name-servers, yellow pages, server directories, etc., are maintained redundantly on multiple sites and updates are incorporated in a lazy manner [35] through gossip messages [27], [16], epidemic propagation, and antientropy [12]. In the epidemic model, update operations are executed locally at any single site. Later, sites communicate to exchange up-to-date information. In this way, updates pass through the system like an infectious disease, hence the name epidemic. Thus, users perform updates on a single site without waiting for communication and the system can schedule communication at a later convenient time. These algorithms rely on the application-specific update operations being commutative and maintain the causal ordering that exists between operations.

Epidemic algorithms were originally designed to satisfy a level of consistency weaker than serializability [8]. Primarily, they preserve the causal order of update operations. For some applications, this is sufficient for correctness. Otherwise, designers optimistically assume that conflicts will be rare and can be handled using application specific compensation. Fundamentally, the goal of epidemic algorithms is to ensure that all replicas of a single data item converge to a single final value. For transaction processing, this is insufficient because transactions create dependencies among the values of different data items. In particular, in a database context, the execution of a set of transactions must be equivalent to a total order. Consider two transactions, t_1 and t_2, that are executed concurrently at different database sites. Furthermore, t_1 reads a value of an object x and updates object y, whereas t_2 reads y and writes x. This is a classical example of a nonserializable execution involving transactions t_1 and t_2. Any epidemic protocol that propagates only write operations to update the values of objects cannot detect this inconsistency.

In this paper, we present a family of epidemic algorithms for maintaining replicated data in a transactional framework. Many nondatabase applications, especially on the

- J. Holliday is with the Department of Computer Engineering, Santa Clara University, Santa Clara, CA 95053. E-mail: jholliday@acm.org.
- R. Steinke is with the Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109. E-mail: Robert.Steinke@jpl.nasa.gov.
- D. Agrawal and A. El Abbadi are with the Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106. E-mail: {agrawal, amr}@cs.ucsb.edu.

Manuscript received 26 July 2000; revised 14 June 2001; accepted 25 July 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 112609.

Internet, can have a large degree of replication, supporting hundreds or even thousands of replicas. Database applications currently are limited by the performance of the replica update protocol to small numbers of replicas on fast, reliable networks or to schemes where the replicas are divided into primary and secondary replicas, and only the primary copies can execute update transactions [14]. Our algorithms are intended for use on a wide area network or Internet where the number of replicas is small to moderate and all replicas can execute update transactions. The algorithms are based on the causal delivery of log records where each record corresponds to one transaction instead of one operation. The first algorithm in this family is a pessimistic protocol that ensures serializability and guarantees strict executions. The next protocol is optimistic in that transactions commit as soon as they terminate locally and inconsistencies are detected later and the resolution of inconsistencies is left to the application. The details of specific conflict compensation schemes are orthogonal to the ideas of this paper. These protocols were first presented in [4]. The final member of the protocol family is pessimistic and uses voting and quorums [13] to resolve conflicts. This protocol was introduced in [20]. The paper is organized as follows: In Section 3, we present the epidemic model of replication. In Section 4, we develop a suite of epidemic algorithms for transaction processing. The algorithms in Sections 4.2 and 4.4 guarantee serializability by checking for both read-write as well as write-write conflicts. The optimistic variant of this algorithm is presented in Section 4.3. Section 5 presents a performance evaluation. Some of these results also appear in [17], [20] and two technical reports [18], [19] have additional performance experiments. Section 6 concludes the paper. The Appendix contains proofs of correctness.

2 Related Work

Ladin et al. [25] 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 [32] and Ficus [15] were developed for weakly consistent systems. However, these systems do not support transactional semantics, 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 [29] uses a 2-tier replication scheme; however, inconsistencies may arise and a variety of reconciliation rules are provided to merge conflicting updates.

Earlier protocols based on epidemic communication considered single operation applications such as replicated dictionaries. Bayou [30] 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. Bayou is flexible in that servers can be created and retired dynamically. However, the server creation pattern affects the size of the Bayou version vectors. Unlike our protocol, Bayou does not detect read/write conflicts, which is necessary to ensure serializability. The epidemic approach or its variant, referred to as asynchronous logging, has been adopted by several database companies like Oracle, Sybase, and Informix. However, when used for multioperation transactions, these techniques are not safe and do not ensure serializability. Rabinovich et al. [31] have recently proposed an epidemic algorithm for transaction management in replicated databases. This protocol only maintains and propagates write operations for eventual consistency of data, hence it fails to detect read-write conflicts and, therefore, allows nonserializable executions in a transactional framework. However, they propose an interesting method of reducing overhead due to the necessary version comparisons.

To ensure one-copy serializability, lazy propagation protocols have recently been introduced by Anderson et al. [7], Breitbart and Korth [10], and Breitbart et al. [9] for database replication. These protocols impose a graph structure on the sites and classify copies into primary and secondary copies. A transaction can update data item x only if the transaction originated at the primary site for x. Also, the replication graph, which represents the propagation of transactions through the system, must be acyclic. Our protocols impose no structure on the network or the sites and allow update transactions to originate at any site. That is, there are no primary or secondary sites, but, rather, all sites have equal status.

3 Database and Epidemic Replication Model

Consider a distributed system consisting of n sites $S_1, S_2, \dots, S_n$, each maintaining a copy of all items in the database. The communication system may be unreliable, i.e., messages may arrive in any order, take an unbounded amount of time to arrive, or may be lost entirely. However, we assume that the communication system ensures that the messages are not corrupted. An event model [26] 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 [26], which is a partial order on all operations in E. The happened-before relation is the transitive closure of the following two conditions:

- Local Order Condition. Events occurring at the same site are totally ordered.
- Global Order Condition. Let e_1 be a send event and e_2 be the corresponding receive event then $e_1 \rightarrow e_2$.

Epidemic algorithms use this execution model to maintain replicated data such as dictionaries, name-servers, distributed calendars, and files [35], [27], [12], [16], [3], [31]. In the epidemic model of communication, one site contacts another and sends it information about events that the sending site believes the receiving site is not yet aware of. Through this kind of pairwise communication, knowledge of events eventually propagates to all sites. The communication model is such that it preserves the potential causality among events captured by the happened-before relation. Minimally, if two events are causally ordered, their effects should be applied in that order at all sites [35], [27],

[12], [16]. Epidemic algorithms generally are implemented using vector clocks [28] or event logs [33] to ensure this property. Vector clocks are an extension of Lamport clocks [26] and ensure the following property for events e and f: $\forall e, f \in E\ e \rightarrow f$ iff $Time(e) < Time(f)$. The ith component of the vector $Time(e)$, which is written $Time_i(e)$, indicates the scalar value of the local clock at site i. In a log-based approach, each site keeps a log of database specific operations by maintaining a log record of each event. Database sites exchange their respective logs to keep each other informed about the operations that have occurred on their sites. This information exchange ensures that, eventually, all database replicas incorporate all the operations that have occurred in the system. Due to the unreliable nature of the communication medium, a record must be included in every message until the sender knows that the recipient of the message has received that record.

Wuu and Bernstein [35] combine the logs and vector clocks to solve the distributed dictionary problem efficiently. Each site S_i keeps a two-dimensional timetable T_i (sometimes called a matrix clock), which has a row for the vector clock of each site, such that 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). That is, the kth row of T_i is what S_i has recorded about S_k's knowledge of events in the system. Thus, the timetable can be used to define the following predicate for a record t corresponding to some event:

$$HasRecvd(T_i, t, S_k) \equiv\ T_i[k, Site(t)] \geq Time_{Site(t)}(t),$$

which is referred to as the *timetable property*. $Site(t)$ is the site at which the event t occurred and, so, $Time_{Site(t)}(t)$ is the scalar clock value assigned by $Site(t)$ when t occurred. Thus, $HasRecvd(T_i, t, S_k)$ will be true if timetable T_i has recorded that site S_k's knowledge of events at $Site(t)$ are at least as recent as the time 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 timetable 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 which is referred to as the *log property* and is stated as follows with respect to a local copy of the log L_i at site S_i: $\forall e, f \text{ if } (e \rightarrow f) \wedge (f \in L_i) \text{ then } e \in L_i$. That is, if event e precedes event f and event f is in the log at site i, then event e must be in the log at site i. The correctness of the algorithm can be established by using both the log and the timetable property.

4 Epidemic Transactions for Replicated Databases

There are two elements of a replicated database that are not fully supported by current epidemic techniques: transactional integrity and correctness. Transactional integrity means that transactions can have more than one operation and these operations must be executed atomically. Correctness means that replicated databases must enforce one-copy serializability [8]. In this section, we develop an algorithm that incorporates transactions into the epidemic framework.

4.1 From Causality to Serializability

Our system model is based on the epidemic model with the following extensions. Each site must have a local concurrency control mechanism that ensures serializability, so we assume that each site uses strict two-phase locking (2PL) to execute transactions. Transactions execute their read and write operations using local copies of the data. At termination, the operations of a transactions are stored in the log as a single record and disseminated to other sites. Since no global synchronization is performed during transaction execution, it is possible for two different database sites to execute conflicting operations. In the context of databases with read and write operations on objects, two operations *conflict* if they originate from different transactions, both are on the same object, and at least one of them is a write operation.

To enforce serializability, we note that all transactions are already partially ordered by the happened-before relation because of the log property. In particular, the log property ensures that if two transactions are causally related, then their relative order of execution will be the same at all sites. On the other hand, epidemic algorithms lack the mechanism to deal with conflicting concurrent operations. Most epidemic-based applications circumvent this problem by only permitting commutative operations on the database. To adapt the epidemic model of communication to transactions, we need to deal with the problem of conflicting operations of concurrent transactions. There are two different approaches to handle this problem which can be classified as a *pessimistic* approach or an *optimistic* approach. In the pessimistic approach, we allow transactions to execute concurrently but prevent any conflicting operations. In the optimistic approach, we detect conflicts after they have occurred and leave the problem of conflict resolution to the application.

Conflicting transactions are two or more transactions that are concurrent and have conflicting operations. In order to detect conflicting transactions, we need to identify the read and write sets of all transactions and we need to be able to detect concurrent transactions. Read and write sets can be easily constructed locally as transactions execute and then included with a log record of updates. Vector clocks can be used to determine the set of concurrent transactions. Vector clocks have the property that two timestamps based on vector clocks are ordered if and only if the corresponding events (transactions in our case) are causally related. Thus, the vector clock-based timestamps of concurrent events (equivalently, transactions) are incomparable. Hence, we use vector clock timestamps and log records containing the read set and write set for identifying conflicting transactions asynchronously. In this way, all of the operations of a transaction correspond to a single event in the dictionary problem and in the log.

4.2 A Pessimistic Epidemic Algorithm for Transaction Processing

We first present an algorithm that corresponds to a read one copy, write all copies replication scheme using epidemic techniques for communication. When initiated, a transaction t executes on a single site S_i. It acquires appropriate read and write locks, writes values to the database, and precommits. At precommit time, t acquires a timestamp using the vector clock of S_i, which is the ith row of S_i's timetable, i.e., $T_i[i, *]$, with the ith component incremented by one. This timestamp assignment ensures that t dominates all those transactions that have already precommitted on S_i regardless of where they were initiated. In this way, t can differentiate concurrent transactions from those that causally preceded it. A precommit record of t is then inserted in the copy of the event log at S_i containing the following information:

1. The site at which t originally executed, denoted $Site(t) = S_i$.
2. The timestamp assigned to t, denoted $TS(t) = T[i, *]$.
3. The read set, $RS(t)$, and write set, $WS(t)$ of t. Note that $WS(t)$ includes the values written by t.

After inserting the precommit record, t releases all its read locks and maintains its write locks as would be done as part of any atomic commit protocol.

The basic idea of this algorithm is to execute a transaction locally and commit the transaction globally by using the epidemic communication model. During the commitment phase, when other sites learn about the precommit request on behalf of a transaction, they must first check if there exist any conflicting transactions and then incorporate the updates of t. This is done as follows: When site S_j receives a log record of t as part of an epidemic message, it identifies in its copy of the log if there exists any transaction t' that has not yet committed and satisfies the following two conditions:

- $TS(t')$ is incomparable to $TS(t)$ (denoted as $TS(t') <> TS(t)$), which indicates that t' executed concurrently with t.
- The intersections of $RS(t)$ and $WS(t')$ or $WS(t)$ and $WS(t')$ or $WS(t)$ and $RS(t')$ are nonempty. This indicates that t and t' execute conflicting operations.

If there exists such a transaction t', then t and t' are aborted at S_j. An abort record (the precommit record with a flag indicating abort) is inserted on behalf of these transactions in the copy of the event log at S_j and all of their locks are released. Otherwise, S_j obtains write locks locally on behalf of t and applies the updates of t to the local copy of the database. Then, it appends the precommit record of t to the local log and updates the local copy of the timetable to reflect the receipt of t.

As logs are exchanged between sites as part of the epidemic process, eventually a site will have enough information to terminate transaction t. If it receives an abort record of t in an epidemic message, it simply aborts t by releasing all its write locks and restoring the before-images of all the affected objects. On the other hand, if the site knows that all the sites in the network have precommitted t, t is committed and its write locks are released. The commitment model used in this protocol is very similar to the decentralized atomic commitment protocol [8]. The correctness of the algorithm depends on the following facts: Each site provides local serializability such that causal order is reflected in the log record timestamps. If two concurrent transactions arrive at the same site, that fact can be detected when the second transaction to arrive checks the log. After a transaction precommits on a site, no new concurrent transactions can be initiated on that site by the nature of causality. A transaction must precommit on all sites, including the initiating sites of any concurrent transactions before it can be committed. Concurrency is a necessary condition for conflict. Therefore, if conflicting transactions exist, they will be detected by the logs before a transaction can precommit on all sites.

How does a site know that all the sites in the network have precommitted t so that t can be committed? When $HasRecvd(T_i, t, S_k)$ is true, site S_i knows that site S_k has received t. This means that S_k has precommitted or aborted t, and S_i has received the timetable information in a message which causally succeeded the precommit or abort of t at S_k. In the above protocol, S_i must have therefore received the precommit or abort record inserted at S_k by this time. Hence, precommits can be deduced from the timing information and the lack of an explicit abort record. When $\forall k\ HasRecvd(T_i, t, S_k)$ is true and S_i has not received an abort record for t, then t can be committed at S_i and the corresponding record of t is garbage collected from the local log, L_i. In fact, explicit abort records are not required. An abort record for t is inserted in the log at S_k if there exists a conflicting transaction t' in L_k. When, at the initiator of t, S_i, $HasRecvd(T_i, t, k)$ holds, t''s log record must also be in L_i. Since t and t' are concurrent and originated at different sites, t' was inserted and processed at S_i after t precommitted. Therefore, when t' is processed at S_i, it will induce the abort of t at S_i. Thus, the abort record of t at S_k indicating that t was aborted due to t' at S_k is redundant since that fact has already been learned by the initiator site of t. We can extend this argument to any site in the network since the epidemic protocol uniformly disperses the information in the network. As a result, only one record, the initial precommit record, needs to be created for each transaction. The propagation of that record and the return propagation of timetable information will detect conflicting transactions in its path. We include a field, denoted *aborted*, in the precommit record as an optimization. If a precommit record of a transaction t arrives at site S_j and S_j aborts t because of a conflicting transaction t', the aborted flag of t is set. Subsequently, when S_j forwards the precommit record of t to S_k, S_k uses the flag to avoid precommitting t locally. However, t is still processed against the log at S_k to ensure that any transactions that conflict with t are aborted at S_k.

We define predicates for the commit and abort decisions using the information that is included in the log and the timetable of every site. The abort predicate (Fig. 1a) holds when t should be aborted. We define the commit predicate (Fig. 1b) as the condition for a transaction successfully committing at a site. The resulting epidemic algorithm for executing transactions on a replicated database is shown in Figs. 3 and 4. Fig. 3 illustrates the handling of transaction

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

(a)

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

(b)

Fig. 1. Transaction predicates. (a) Condition for aborting t at S_i. (b) Condition for committing t at S_i.

execution locally at a site. Although the algorithm is illustrated using read and write sets, it can be easily modified so that the read and write sets are constructed incrementally as the execution of a transaction proceeds. The interaction of a transaction with the epidemic algorithm occurs when the transaction precommits (i.e., makes its writes recoverable at S_i) and inserts a precommit record in the log. Both T_i and L_i are updated in a critical section to ensure atomicity of updates to these two data-structures by local concurrent transactions. After precommitting, the transaction can release all the read locks it obtained, however, write locks are retained until the transaction terminates. The data structure used to store precommit records appears in Fig. 2.

The send function (Fig. 4) is used by site S_i to disseminate the precommit records of local transactions as well as remote transactions about which S_i has become aware of through message propagation. Periodically, S_i sends part of its log to the other sites with its two-dimensional time-table. The frequency of propagation, as well as the destination to which messages, are sent are application dependent and can be tuned appropriately. When S_i sends a message to S_k, it does not send any records that S_i knows that S_k already knows about by employing the $HasRecvd$ predicate.

In the receive procedure, transactions are processed one at a time in the order of the received log. The receiving site first checks if it already has the precommit record for transaction t. This can be done simply by checking whether $HasRecvd(T_i, t, S_i)$ is true. If the site has not already received the record, then it must check against concurrent transactions in the log for conflicting operations. S_i determines if there are concurrent conflicting transactions in its log. All such transactions, as well as t, are aborted in this case. The newly received transaction has its record marked as aborted and its updates are not applied, but its record is inserted into the log and the timetable is updated to show that it has been received. This is necessary to propagate the knowledge of the conflict to other sites. If there are no prior conflicting transactions, S_i acquires write locks of t to update the appropriate objects locally. If there are any on-going local transactions that hold conflicting locks, such transactions are aborted and t is granted the locks. This is referred to in the figure as function $ForceWriteLocks$. However, there could be causally preceding global transactions that hold conflicting locks. $ForceWriteLocks$, in this case, enqueues t's lock request. Since on-going local transactions will causally follow t, we only need to abort them if such transactions are reading or writing objects that must have been written by t. This is the reason why t only needs to obtain its write locks at S_i, but does not need read locks. Finally, S_i updates the appropriate data items, increases the value of $T_i[i, Site(t)]$ to reflect the fact that t is now known to S_i, appends t to log L_i, and precommits. After all records are processed, the procedure updates the other rows of the time-table. Next, the precommit records in L_i are checked to determine if they can be committed locally by using the $Commit$ predicate (Fig. 1b), where $TS_{Site(t)}(t)$ indicates the component of the vector timestamp $TS(t)$ contributed by $Site(t)$, the originating site for transaction t. All log records of committed or aborted transactions that are known to all other sites are garbage-collected from L_i. The proof of correctness of the algorithm appears in the Appendix.

An interesting aspect of the above algorithm is that it does not require distributed deadlock detection. To see that this is true, note that, if transaction t is waiting for a lock held by transaction t', then $t' \rightarrow t$. It is impossible for t' and t to be concurrent because they would both have been aborted when t checked the log before attempting to acquire locks. $t \rightarrow t'$ is impossible because this violates causal message delivery. Also, t' cannot be a local transaction because it

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

Fig. 2. A transaction record.

```
Transaction(RS, WS, f(x)):
begin
    GetReadLocks(RS);
    values := f(read (RS));
    GetWriteLocks(WS);
    WriteValues(WS, values);
    begin mutex
        T_i[i, i] := ++clock_i;
        L_i := L_i ∪ {⟨RS, WS, values, i, T_i[i, *]⟩};
        Pre-Commit;
    end mutex
    ReleaseReadLocks(RS);
end;
```

Fig. 3. The pessimistic epidemic algorithm for executing transactions at N_i.

```
Send(m) to N_k :
begin
    NP := {t | t ∈ L_i ∧ ¬HasRecvd(T_i, t, N_k)};
    send ⟨NP, T_i⟩ to N_k;
end;

Receive(m) from N_k :
begin
    let m = ⟨NP_k, T_k⟩ ;
    foreach {t | t ∈ NP_k ∧ ¬HasRecvd(T_i, t, N_i)} do
        begin mutex
        if {∃t' ∈ L_i | t'.time <> t.time ∧ (t'.WS ∩ t.WS ≠ φ ∨
                  t'.WS ∩ t.RS ≠ φ ∨ t'.RS ∩ t.WS ≠ φ)} then
            Abort(t);
            Abort(t');
            t.aborted = true;
            t'.aborted = true;
        else if (¬t.aborted) then
            ForceWriteLocks(t.WS);
            WriteValues(t.WS, t.values);
        endif
        T_i[i, t.node] := t.time[t.node];
        L_i := L_i ∪ {t};
        if (¬t.aborted) then
            Pre-Commit(t);
        endif
        end mutex
    endfor
    begin mutex
        ∀K, J T_i[K, J] := max(T_i[K, J], T_k[K, J]);
        foreach t ∈ L_i do
            if Commit(t, N_i) then Commit(t); then
        endfor;
        L_i := {t | t ∈ L_i ∧ ∃j | ¬HasRecvd(T_i, t, N_j)};
    end mutex
end;
```

Fig. 4. Epidemic propagation of transaction records from other nodes to N_i.

would be aborted by procedure $ForceWriteLocks$. Waits for dependencies between precommitted transactions must be a subgraph of the happened-before relation, which is acyclic, and precommitted transactions cannot wait on local transactions. Therefore, precommitted transactions cannot be involved in deadlock. Any transaction holding locks on more than one site must be precommitted and, therefore, any cycle in the global waits-for graph must involve only local transactions at a single site and can be detected and resolved locally. (See Appendix, Theorem 1.)

4.3 Increasing the Optimism of Epidemic Transactions

In this section, we use the basic epidemic algorithm for processing transactions in databases and introduce two modifications that will increase the asynchrony and concurrency among epidemic transactions. We first observe that the basic epidemic algorithm holds all write locks globally until the precommit record of a transaction is known to have been disseminated to all sites in the system. This is done to ensure that other transactions do not read uncommitted data. Since we are assuming a low-conflict environment, this may not be a significant problem. Given this assumption, we can relax the requirement of holding locks until transaction termination. This algorithm still ensures both serializability and recoverability [8]. We then use this modified algorithm to develop a more radical version of the epidemic algorithm which is more in line with contemporary approaches that are being used commercially to deal with replication [22], [29]. In this approach, transactions are optimistically committed as soon as they terminate at the initiating site. Epidemic propagation is used to disseminate the effects of these transactions so that the replicated data becomes mutually consistent and to detect conflicting transactions. Such transactions are reported to the application level for application dependent resolution.

4.3.1 Optimistic Releasing of Locks

The sole purpose of holding write locks after precommit time is to confine the effects of a transaction abort to the transaction itself. Write locks are held until commitment to avoid cascading aborts and to ensure strict execution of transactions. In order to ensure that aborting a transaction does not influence previously committed transactions, we must require that, for every transaction t that commits, its commit follows the commit of every other transaction from which t read. Such executions are called *recoverable* (RC). Recoverability, however, does not guarantee freedom from cascading aborts. Cascading aborts can be prevented by requiring that every transaction reads committed values. Executions that satisfy this requirement are said to *avoid cascading aborts* (ACA). Finally, an execution is called *strict* (ST) if all read and write operations are executed on committed values [8].

If we modify the epidemic algorithm so that write locks are released at precommit, we can still guarantee serializability as well as recoverability. However, releasing locks early exposes uncommitted values to other transactions. Thus, for recoverability, if a transaction t reads from an uncommitted transaction t', t can commit only if t' has committed. For a transaction t, any transaction that it reads from must have causally preceded it. Therefore, any site which could precommit t must have received and either precommitted or aborted all causally preceding transactions. For the initiating site of t, this is true by the definition of causality. For all other sites, this is true by causal message delivery. The epidemic algorithm has to be modified such that if t aborts at a site, then it must abort all transactions that read from t and are in the log. Furthermore, if t has read from a transaction that has been aborted at S_i, then t must be aborted. This is accomplished by including an additional field in the log record, referred to as $readfrom$, for transaction t which contains the identity of all transactions from which t reads. For t to commit, it must have precommitted on all sites. This implies that all transactions which t reads from are also precommitted on all sites and, thus, have also committed. This is a sufficient condition for recoverability.

In traditional databases, ACA executions are desirable because cascading aborts may result in lower throughput in the system. By releasing locks early, the epidemic algorithm introduces the vulnerability to cascading aborts, so this protocol variation should be used only when conflicts are expected to be rare. On the other hand, by releasing locks early, unnecessary blocking due to locks held during dissemination of precommit of transactions is eliminated while transaction execution remains serializable.

4.3.2 Optimistic Commitment of Epidemic Transactions

In the previous protocols, transactions completed execution locally and delayed the commitment until they are aware that their precommit has been processed at all sites in the network (and, hence, all their effects have been incorporated on all replicas). In contrast, the optimistic epidemic algorithm commits a transaction locally with the optimistic assumption that no conflict will arise as the commit record of this transaction disseminates through the network. If a conflict occurs, the epidemic algorithm detects it and reports it to the application level for conflict resolution.

We now briefly describe the changes that are needed to modify the epidemic algorithm to optimistically commit transactions early. A more detailed discussion appears in [4]. In the case of transaction execution of Fig. 3, the main change is that, instead of inserting a precommit record, a transaction executes as if it is a centralized database and, hence, inserts a commit record and releases all of its locks. The significant difference in the receive procedure for optimistic operation is that transactions are committed at the time they were precommitted in the conservative algorithm and, as a result, cannot be aborted. The abort flag is renamed $inconflict$ and, when conflict is detected, the algorithm raises an exception by calling $ResolveConflict$. The values written by the transaction must be applied to the database whether or not the transaction is in conflict because those values are already committed at other sites.

Existing databases such as Oracle have developed application dependent reconciliation rules for conflict resolution [29]. Additionally, there are metarule paradigms to specify which rules to apply and in what order if more than one rule is applicable. For example, Jagadish et al. [21] describe a metarule language and inference procedures to determine if a particular set of metarules is unambiguous. Using these two techniques, it should be possible to create an automated conflict resolution procedure.

4.4 Epidemic Quorums

The pessimistic epidemic algorithm (referred to as ROWA) described in Section 4.2 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* [13]. Quorums are sets of sites such that the intersection of any two quorums is nonempty. For example, a majority quorum is any set that contains a majority of sites. When there is a set of conflicting transactions, sites will vote to determine which one commits. Conflicting transactions which have not yet received enough votes to commit or abort must be allowed to coexist on a site, unlike in the ROWA epidemic algorithm in which they were aborted as soon as the conflict was detected.

4.4.1 The Epidemic Quorum Protocol

We propose enhancing the pessimistic algorithm by using voting and quorums to resolve commit decisions. This is similar to the technique developed by Keleher [23] except that our approach ensures serializability among multi-operation transactions, whereas Keleher treats only single-operation requests. In the epidemic-quorum algorithm, 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. To cope with this, we introduce the idea of an *antiquorum* as a set of sites without which a transaction cannot acquire a quorum. More formally, an antiquorum is any set of sites that intersects with all quorums. Any quorum is an antiquorum, but an antiquorum is not necessarily a quorum because we do not require antiquorums 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 antiquorum 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 antiquorum 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 antiquorum and t is aborted.

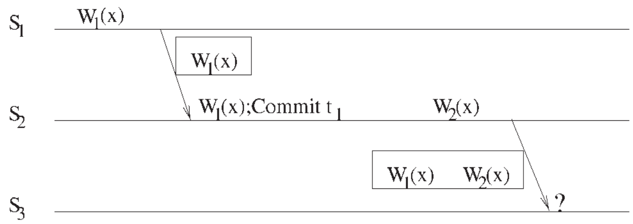
When S_i has received t, but has not received enough information to commit or abort t, t is said to be *uncertain* at S_i. This poses the problem of what to do with uncertain conflicting transactions. To preserve causality, when t arrives at S_i, it acquires write locks and applies its writes before 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. 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.

To allow conflicting uncertain transactions to hold locks on the same data item, we adapt an idea from multigranularity

Lock Held

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

(a)



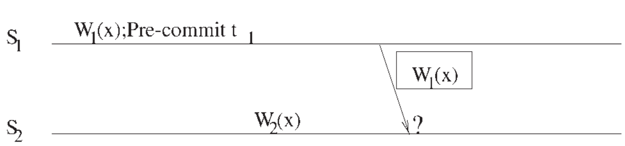
(b)

Fig. 5. Using intention to write locks. (a) IW lock conflict table and (b) situation $*_1$.

locking called *intention to write* locks [8]. The conflict table for intention to write (IW) locks is given in Fig. 5a. IW locks do not conflict with other IW 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 IW locks on all data items written by the original transaction and precommits. When a local transaction precommits, 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 IW locks, and behaves like a remote transaction. This prevents local transactions that have not precommitted from accessing the data item, but allows conflicting, uncertain transactions to precommit and simultaneously reserve access to the data item. If two transactions hold IW 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 IW 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 which one transaction causes the abort of another transaction. The first situation is referred to as $*_1$ in the lock conflict table and is demonstrated in Fig. 5b. Transaction t_1 writes data item x on site S_1 and precommits ($W_1(x)$.) 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 precommits. Now, t_1 is known to two sites (a 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 precommits ($W_2(x)$). Now, when the records of t_1 and t_2 are transmitted to site S_3. they will both acquire IW locks on x and precommit. When they attempt to commit, they will have to convert their IW locks to write locks to write their values to x. Will this cause a problem because each transaction is prevented by the other transaction's intention to write lock? This is not a problem because transactions must commit in causal order even if they are received in the same message as required to make the serialization order based on causal order. The order of transactions in a message preserves the order of transactions in the log because messages are generated from the log. In this case, t_1 preceded t_2 in the message and in the log, so t_1 converts its lock first and is able to commit. In general, 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 IW 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 IW 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 IW locks.

The second situation is referred to as $*_2$ in the lock conflict table and is demonstrated in Fig. 6a. Transaction t_1 writes x on S_1 and precommits. Concurrently, transaction t_2 writes x on S_2, but a message containing the log record of t_1 arrives before t_2 precommits. 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 precommit 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



(a)

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

(b)

Fig. 6. Situation $*_2$ and a vote record. a) Situation $*_2$. (b) A vote record.

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

Fig. 7. The quorum epidemic algorithm for executing transactions at S_i.

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. Deadlock cycles can only involve local transactions waiting for locks from other local transactions at the same site and these deadlocks can be detected locally.

4.4.2 Implementing the Epidemic Quorum Algorithm

The data structure used to represent transaction records (Fig. 2) is the same one that was used previously. As before, 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 precommits ($time$). Fig. 6b 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.

The algorithm for processing a transaction at its initiating site is given in Fig. 7. Each site keeps a scalar local clock ($clock_i$), a two-dimensional time-table (T_i), a log of transaction records (L_i), and a log of vote records (V_i). A transaction starts execution at its initiating site by acquiring locks, performing computation, and writing values to the database. Then, before the transaction precommits, it acquires a mutex, puts a record in the transaction log

```
Send(m) to S_k :
begin
    begin mutex
        SP := {t | t ∈ L_i ∧ ¬HasRecvd(T_i, t, S_k)};
        VP := {v | v ∈ V_i ∧ ¬HasRecvd(T_i, v, S_k)};
        send ⟨SP, VP, T_i⟩ to S_k;
    end mutex
end;
```

Fig. 8. Epidemic propagation of transaction records from S_i to S_k.

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. 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.

The algorithm for sending log records and processing received log records is given in Figs. 8 and 9. 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. Vote records can be sent in any order. 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

```
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;
```

Fig. 9. Epidemic propagation of transaction records from other nodes to S_i.

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 precommits. When l precommits, 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 precommitting 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 conflicting transactions. In Fig. 9, $v = VoteFor(t, i)$ 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 precommits. Then, the receive procedure releases the mutex before it handles the next transaction. Any nonconflicting local transactions waiting for the mutex can now precommit. 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.

4.4.3 Read-Only Transactions in the Quorum Protocol

In our protocol, read-only transactions can be executed locally without the need of global synchronization and still maintain "external consistency" [34]. In Weihl's definition [34], external consistency means that the values read by each read-only transaction are the result of a serial execution of some subset of the update transactions in that computation. In our protocol, all update transactions plus any single read-only transaction always appear to be one copy serializable, but any history considering more than one read-only transaction might not. This is because an update transactions can commit as soon as it receives yes votes from a quorum of sites and read-only transactions can then read the items it has written on sites that have received and committed the update transaction. Each pair of update transactions has a common site in their quorums, but, for three transactions, there may be no site which receives all three transactions before they start committing. Thus, three update transactions and three read-only transactions may not be one-copy serializable. A detailed proof of external consistency is given in the Appendix.

5 Performance Evaluation

In order to evaluate the performance of these epidemic algorithms, we used a detailed simulation of a distributed replicated database. The simulation is based on accepted database modeling techniques [5]. This article [5] provided the basic architecture for our simulation as well as values for many of the parameters. All measurements in these experiments were made by running the simulation until a 95 percent 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 timetable 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 with all the reads done before all the writes. The time between successive operation requests within a transaction is a parameter, IntTime, set at 3 ms. Each site has a copy of the 1,000 page database. This is a relatively small database and was chosen so that we could study the effects of data contention more easily. A page is 2 Kbytes, which is the locking granularity as well as the data disk granularity. The DBMS uses strict two-phase locking to ensure local serializability. A *wait-for-graph* handles local deadlocks.

The system parameters of the model are given in Fig. 10a along with their values. The parameters governing the generation and behavior of transactions are given in Fig. 10b. The generation of new transactions is governed

Parameter	*Meaning*	Value
MinDiskTm	Min data disk time	4 ms
MaxDiskTm	Max disk access time	14ms
CPUInitDisk	CPU time to access disk	0.3 ms
LogDiskTm	Time for forced log write	8 ms
LogPageSize	Log records per page	100
HitRate	Probability of cache hit	0.9
LockTime	CPU time to request lock	0.006ms

(a)

Parameter	*Meaning*	Value
ReadSetMin	Smallest read set size	5
ReadSetMax	Maximum read set size	11
WriteSetMin	Smallest write set size	1
WriteSetMax	Maximum write set size	4
CPUPgTime	Time spent on a data page	1.0ms
IntTime	Time between requests	3 ms

(b)

Fig. 10. Parameters. (a) System parameters and (b) transaction parameters.

by the parameter InterarrivalTime. 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. Unless otherwise stated, the percentage of read-only transactions is 75 percent. This is a reasonable assumption since, in most database applications, most transactions are queries. The readset size is between seven and 11 read operations for read-only transactions. Update transactions have a read set size between five and eight read operations and a write set size of one to four write operations.

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 0.1 ms. When a site is ready to initiate an epidemic session with another site, it chooses that site at random from the remaining sites. The messaging rate is given in milliseconds and tells how often a site is allowed to initiate an epidemic communication. The performances of all of the epidemic protocols were sensitive to variation in this rate, so it was decided to fix it at two milliseconds so that the relative performance of the protocols could be studied. 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.

- **Precommit time**. If an update transaction precommits, it records the elapsed simulation time since it made its first read request of the system. This is the mean value for all update transactions.
- **Commit or Update commit time**. If a transaction commits (final commit), it records the elapsed simulation time since it made its first read request of the system. Only update transactions are included in this measure.
- **Read-only commit time**. Read-only transactions do not do a precommit, they simply commit. The commit time of read-only transactions is recorded separately from update transactions.
- **InterarrivalTime**. This is the mean of the exponential distribution for the per site transaction interarrival time. As it is made shorter (smaller), the load on the system increases.
- **Start rate**. The transaction start rate is the number of new transactions started per second.

We refer to the pessimistic epidemic algorithm as read-one write-all or eROWA and the quorum algorithm as 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.

To further assess the performance of the epidemic protocols, we modeled a traditional eager update protocol with our simulator. 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 and perform the write, it responds with an acknowledgment. When the home site receives acknowledgments from all sites, it lets the transaction 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]. Since this 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 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 InterarrivalTime (transaction interarrival rate).

We chose this locking-based, eager update protocol to compare with rather than a more optimistic protocol such as one with local execution controlled by locking followed by a synchronous remote execution and two-phase commit [24]. Performance studies of centralized concurrency control protocols [6] have established that optimistic approaches outperform traditional locking protocols only when resources are abundant. Another complexity that arises with regard to optimistic concurrency control protocols is in extending them for distributed environments. In particular, since optimistic protocols do not perform any synchronization during the execution phase, the commitment of a distributed transaction needs to be coordinated at all database sites. Thus, under distributed optimistic concur-

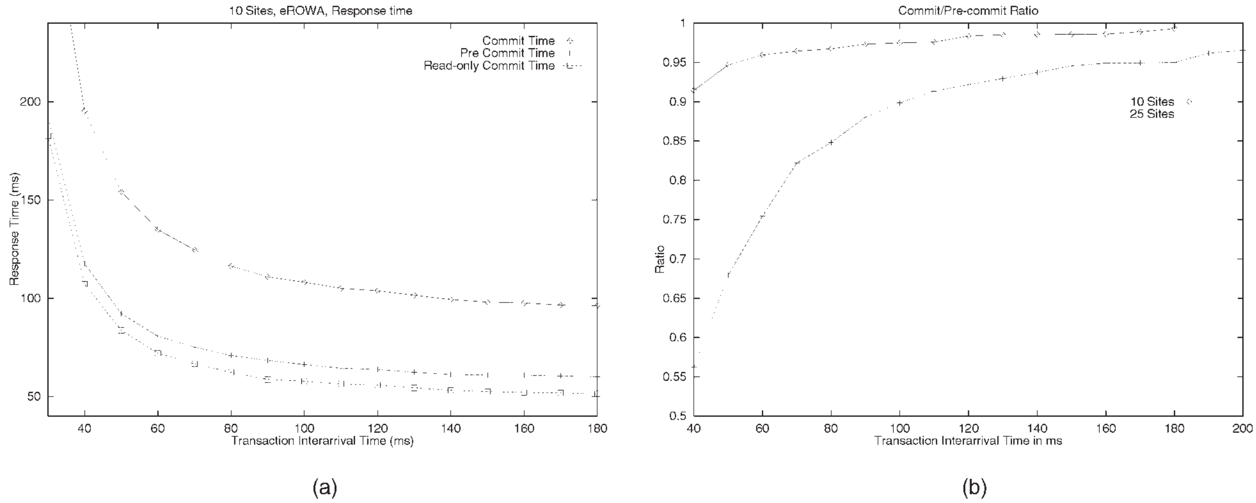


Fig. 11. Response time and commit ratio. (a) Response time for 10 sites. (b) Commit/precommit ratio.

rency control [11], a transaction necessarily involves at least two phases. During the first phase, the relative serialization order (and validation) is determined at all participating database sites and then the coordinator has to ensure that relative serialization orders are consistent with each other. Otherwise, the transaction needs to be aborted. Given such complexities associated with the distributed optimistic concurrency control protocol, commercial database management systems have been reluctant to embrace optimistic methods for concurrency control even for centralized environments. It is for these reasons that we assume a lock-based pessimistic concurrency control mechanism for our comparisons.

Due to space limitations, it is not possible to include the results of all the performance experiments in this paper. The technical reports [18], [19] have the results of additional experiments, including the use of a hit rate of 80 percent, 90 percent, and 100 percent, longer running transactions (by increasing the interoperation time), and different network configurations and communication rates. We believe that the included results are the most significant and representative. The 90 percent hit rate may be high for a server cache hit rate, however, experiments done with other hit rates confirm our expectations that these parameters tend to affect all of the propocols studied uniformly and do not affect the comparison of the protocols.

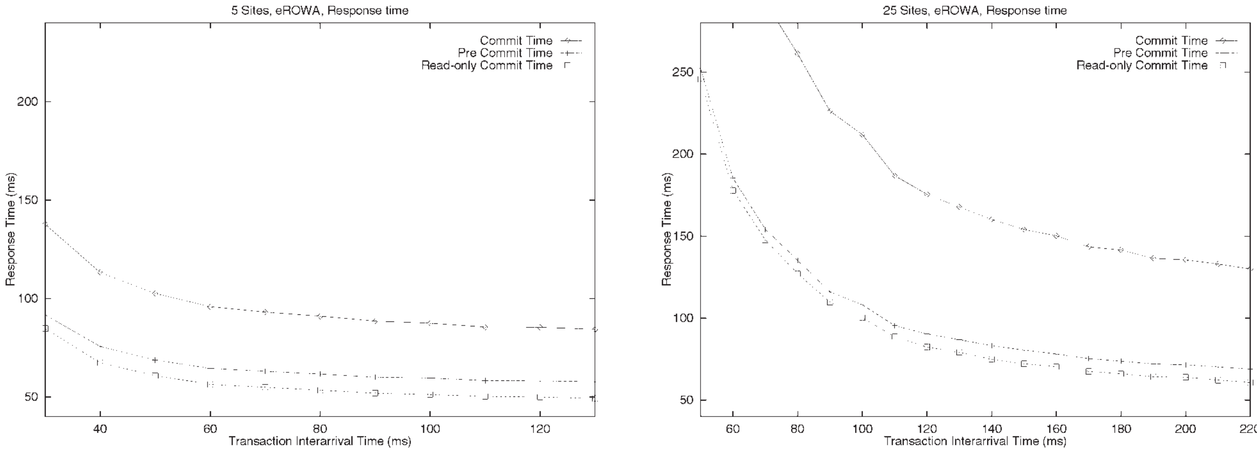
5.1 The ROWA Protocol

We performed experiments to analyze the response time with different degrees of replication and varying proportions of read-only transactions. We also performed measurements to determine how likely it is that a precommitted transaction will successfully commit. These measurements could indicate favorable conditions for using the optimistic protocol (Section 4.3).

5.1.1 Response Time Analysis

In our first set of experiments, we analyze the response time for both read-only and update transactions as a function of increasing InterarrivalTime. We consider a system with 10 sites (i.e., 10 copies). The results in Fig. 11a show the commit time for read-only transactions as well as both the precommit and commit time for update transactions. Both the x and y-axis are in milliseconds. As expected, a greater load on the system, represented by a smaller InterarrivalTime, causes an increase in the time required for a transaction precommit as well as a commit. Since read-only transactions execute locally, they are not adversely affected by the change in work load except at very low InterarrivalTime (i.e., high load) when the rate of concurrently executing transactions at each site is so high that conflicts are frequent and there is competition for resources. In fact, it is quite easy to account for the response time of read-only transactions. Since each transaction has an average of nine operations requested three milliseconds apart, the issuing of the operations takes over 27 ms on the average. In addition, 1.0 ms of CPU time is consumed for processing each operation, adding 9 ms to the transaction time. Disk I/O takes up 8.37 ms (nine operations at a 90 percent hit rate with an average 9.3 ms disk access time) for a total of 44.37 ms. At a low load, e.g., InterarrivalTime = 180 ms, a read-only transaction commits in about 51.0 ms. Hence, a total of 6.6 ms is spent on database management functions such as log writes, lock table management, and deadlock detection that take place during the lifetime of the transaction as well as competition with other transactions for resources.

Update transactions take longer than read-only transactions to precommit since they must force write update data to the recovery log disk, requiring approximately 8 ms, and the average cost of a write is slightly higher than the cost of a read operation. However, as with read-only transactions, an update transaction precommits based on local execution and requires no communication. Therefore, the response time for the precommit of update transactions closely follows the response time of read-only transactions. Committing update transactions, however, requires communication with all the other sites in the system. For a transaction to commit at the home site, the site must know that all sites have precommitted that transaction. This delay is approximately 36 ms at InterarrivalTime = 180 ms. The additional time to commit is spent on disk I/O (a site which



(a) (b)

Fig. 12. Response time for five and 25 sites. (a) Five sites and (b) 25 sites.

receives a precommit record must do the writes before putting it in its log and sending it on) and on communication costs.

5.1.2 Varying Degree of Replication

In this experiment, we compared systems with five (Fig. 12a), 10 (Fig. 11a), and 25 (Fig. 12b) copies on different sites. We were particularly interested in the communications overhead introduced by the additional sites as opposed to the advantage of being able to handle more read-only transactions. These graphs show the effect of increasing the number of sites. Recall that a transaction interarrival time of 100 ms means a transaction is started every 100 ms at each site. Notice that, as the number of sites increases, the relative distance between precommit and commit time increases, indicating additional overhead.

In a 10 site system with 200 new transactions per second (InterarrivalTime = 50), the precommit time is 92.1 and the commit time is 154.0. This means the overhead introduced by the network to enable the transactions to commit and ensure serializability is (154 - 92)/92 = 67 percent. In a 25 site system with 200 transactions per second (InterarrivalTime = 120), the precommit time is 90.3 and the commit time is 175.4. This means the overhead is 94 percent.

5.1.3 Precommits as an Indicator of Commit

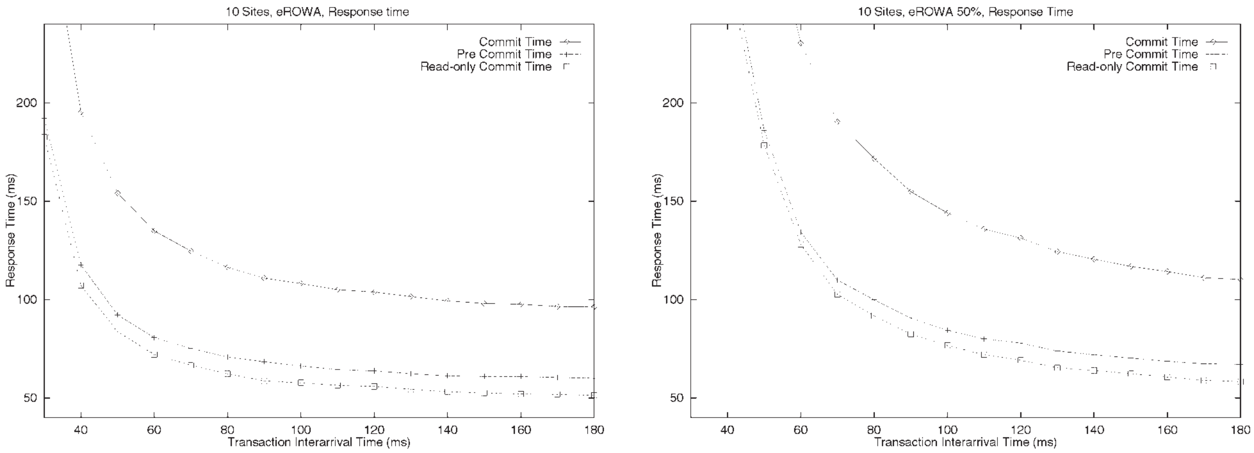
All read-only transactions commit locally and release all their locks without the need for any communication. Update transactions, on the other hand, first precommit and then later commit at the home site when it is known that the transaction has precommitted at all sites. When a transaction precommits, it releases all its read locks, however, the write locks are held until commit time. Optimistic protocols, such as those discussed in Section 4.3, which allow update transactions to release all locks at precommit time (and maybe even commit) by optimistically assuming that the transaction will actually succeed in committing, are appropriate in some systems. Such a system should have low conflict rates and the application should be able to recover from any incorrect ramification of the optimistic decision. Note that if transactions release all locks at precommit time and, later, a conflict is detected, serializability can still be maintained, but the aborting of transactions may result in cascading aborts [8]. The more radical approach of actually committing the transaction at precommit time (and, thus, the complete execution of the transaction is local) is actually more in line with current commercial systems that use replication.

Since both cascading aborts and the application of reconciliation rules to resolve the effect of committing conflicting transactions should be minimized as much as possible, it is important to determine in an optimistic environment how frequently precommitted transactions will not succeed in committing. We therefore measured the ratio of committed to precommitted update transactions.

Presumably, when the data contention was low, successful precommit would be a good indication that the transaction would go on to successfully commit. We investigated 10 and 25 site systems. The results (Fig. 11b) show that, in a 10 site system with a transaction interarrival time of 60 ms or more, 95 percent of the update transactions that precommit go on to commit. This is quite encouraging as this is a reasonable operating load. The commit/precommit ratio for a 25 site system was less encouraging, but, given the longer transaction lifetime caused by increased communication costs, not entirely unexpected. We therefore conclude that, in a small system with moderate system load, an optimistic approach that uses the precommit as a *probabilistic* indicator of commitment is actually a viable option and deserves further investigation.

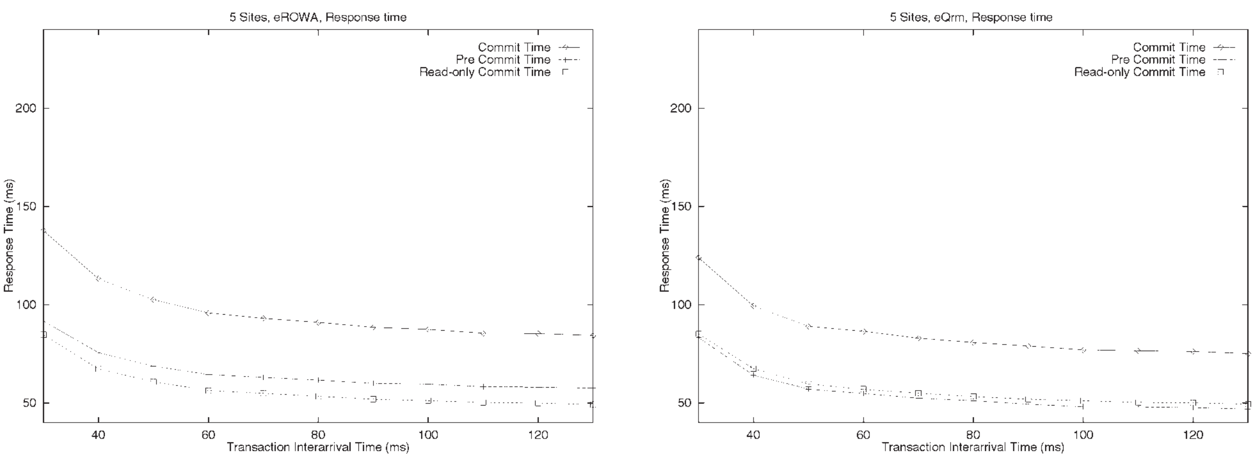
5.1.4 Varying the Transaction Mix

In order to study the effects of updates on system performance, we varied the percentage of update transactions generated. Our earlier results that indicate that the response time of read-only transactions is unaffected by replication break down when the proportion of update transactions is increased to 50 percent (Fig. 13b) from 25 percent (Fig. 13a). The read-only commit time at 50 percent update is greater due to the increased data contention (more update transactions means more write locks are held for a long time forcing readers to wait). At



(a) (b)

Fig. 13. Ten sites with different proportion of read-only transactions. (a) Ten sites, 75 percent Read-only. (b) Ten sites, 50 percent Read-only.



(a) (b)

Fig. 14. Response time for five sites. (a) Protocol eROWA and (b) protocol eQrm.

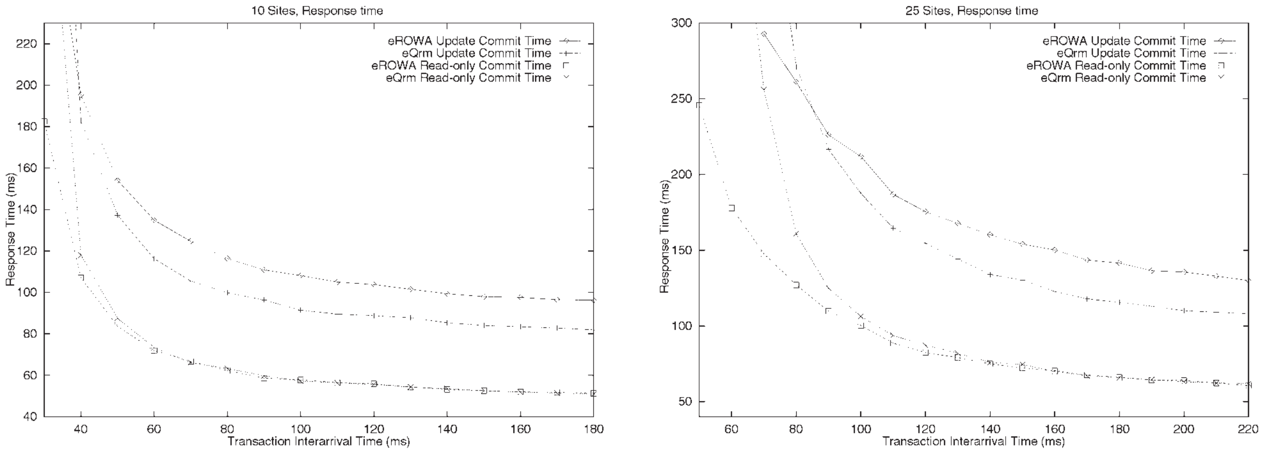
high load (low InterarrivalTime), the system is overwhelmed by data contention to a much greater degree than a 75 percent read-only environment. We also conducted experiments with a higher percentage of read-only transactions. We observed that increasing the percentage of read-only transactions does not significantly influence the latency incurred in committing update transactions globally. However, due to decreased contention on resources, there is an overall performance gain for all transactions.

5.2 The Quorum Protocol

This section explores the quorum variation of the epidemic protocol, eQrm, and contrasts it with both the ROWA epidemic protocol and the traditional eager protocol. We analyze the response time and the throughput of eQrm as compared with the epidemic ROWA. We show the effect of increasing the proportion of update transactions on throughput. The response times are compared with the epidemic ROWA and the traditional eager protocol for a 10 and 25 site system.

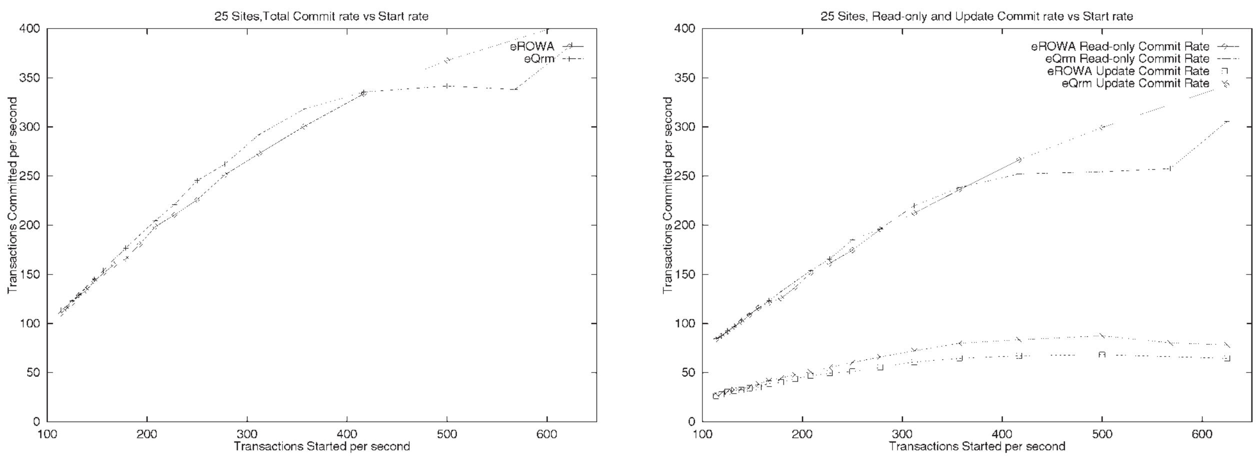
5.2.1 Response Time Analysis

In a system with five sites, as shown in Fig. 14, the read-only commit time is about the same for eROWA and eQrm. In eROWA, the precommit 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 precommit. In eQrm, the update transactions acquire all needed locks, but do not do the data writes or write to the log disk before precommitting. 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 precommit 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 precommit response time of update transactions closely tracks the read-only commit time, we drop it from our future graphs.



(a) (b)

Fig. 15. Response times versus interarrival time. (a) 10 sites and (b) 25 sites.



(a) (b)

Fig. 16. Commit rates for 25 sites. (a) Total commit rate and (b) read-only and update commit rate.

We then conducted experiments to study the performance of epidemic transaction protocols as the number of sites in the system increases. In Fig. 15a, 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 five site system, although the quorum protocol, eQrm, provides substantial benefit. Recall that an InterarrivalTime of 50ms for five sites means that 100 transactions are entering the system per second. This same transaction generation rate would be represented by an InterarrivalTime of 100ms in a 10 site system.

In Fig. 15b, 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 an InterarrivalTime of 80ms in a 10 site system and an InterarrivalTime of 200 in a 25 site system. 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 InterarrivalTime less than 80 ms. 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 an InterarrivalTime of 130ms, for eROWA, an average of 88ms elapse between the time update transactions are precommitted and the time they are committed. This time differential, which is a measure of the time needed to propagate the precommit records throughout the system and collect enough information to commit the transaction, is 65 ms for eQrm. Thus, eQrm provides 26 percent improvement in update commit time. In a heavily loaded system (InterarrivalTime less than 80ms), the response time for committing update transactions using eROWA outperforms 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.

5.2.2 *Throughput Analysis*

We now discuss in more detail the throughput, or commit rate, of the system under varying conditions. In Fig. 16, 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 Fig. 16a, while, in Fig. 16b, 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 Fig. 16a). In particular, at a start rate of 156.2, the total commit rate is 151.2 transactions per second or 96.7 percent for eROWA and 154.0 transactions per second or 98.6 percent for eQrm. At a start rate of 250, the total commit rate for eROWA is 90.2 percent and the total commit rate for eQrm is 98.1 percent. 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 73.5 percent for eROWA and 68.3 percent for eQrm. At 625 transactions per second, the total commit rate is 65.1 percent for eROWA and 61.4 percent 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 Fig. 16b, 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 percent of the committed transactions. The update commit rate is 35 transactions per second which is slightly less than 25 percent. The read-only commit rate for eQrm is 114.0 representing 75 percent of the committed transactions. The update commit rate is 37.4 transactions per second which is about 25 percent. Both protocols are maintaining the original transaction mix of 75 percent read-only and 25 percent 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 77 percent of the committed transactions. The update commit rate is only 22.6 percent. 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 75.4 percent of the committed transactions. The update commit rate is 24.5 percent. Thus, eQrm maintains the original balance between update and read-only transactions, whereas eROWA starts to favor 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 is 73.5 percent; however, the read-only transaction commits now represent 81.4 percent of the committed transactions and the update commits are only 18.6 percent. At a start rate of 500, the read-only transaction commit rate for eQrm is 74.4 percent of the committed transactions and the update commits are 25.6 percent. 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. At a start rate of 500, 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 Fig. 15.

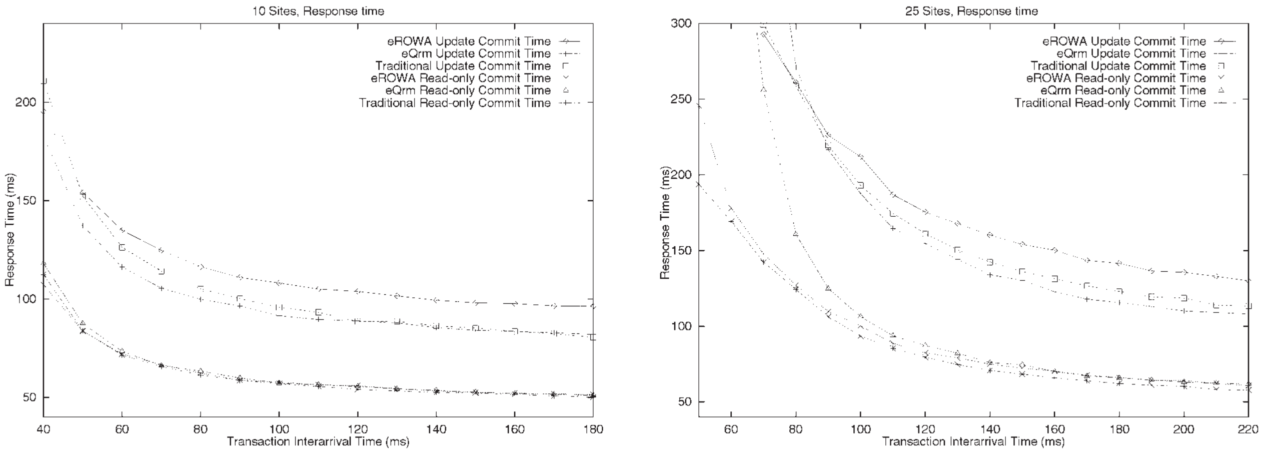
To further validate our observation that eQrm maintains the percentage of committed update transactions even in heavily loaded systems, we conducted an experiment where the proportion of update transactions was increased to 50 percent of all transactions. Increasing the proportion of update transactions gave us the opportunity to see if the eQrm protocol can maintain the transaction mix at 50 percent update. 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 percent update and 50 percent read-only. Above that rate, eQrm begins thrashing and total throughput decreases. The total commit rate for eROWA continues to increase, sacrificing update transactions as needed. Read-only transactions are "easier" to commit because they share locks and use only local resources.

5.2.3 Comparison with Traditional Methods

In this section, we explore the advantages of epidemic-based updates versus the traditional eager approach. A simple traditional update protocol allows for local execution of read-only transactions, just like the epidemic protocol. When an update transaction does a write, the home site DBMS must acquire write locks for that data page at each replica site. The home site DBMS sends a message to each other site requesting a write lock. When the remote site is able to grant the lock, it responds with an acknowledgment. When the home site receives acknowledgments from all other sites, it lets the transaction perform the data write and proceed with its next operation. When the transaction has completed all its operations, the home site DBMS starts an atomic commit protocol such as a two-phase commit [8].

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 (Fig. 17a) and 25 sites (Fig. 17b). 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 Fig. 17, 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 an InterarrivalTime greater than 80 ms (a heavy load for 25 sites—over 300 transactions started per second) and for 10 sites at an InterarrivalTime



(a) (b)

Fig. 17. Response time compared to traditional protocol. (a) Ten sites and (b) 25 sites.

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 precommit record enabling the home site to commit the transaction. A remote site forces its recovery log later, when it commits the transaction.

6 Conclusion

Many techniques are currently available to support the weak consistency of replicated data. These systems process updates as individual operations or copy updated values of individual data items. This makes them insufficient for use in applications with transaction-based semantics. We have developed a family of epidemic algorithms that propagates updates as whole transactions. This allows sites to maintain the atomicity of transactions as well as detect nonserializable executions. The algorithms can be used in a conservative manner that prevents nonserializable executions, or in an optimistic manner that merely detects nonserializable executions for application specific compensation.

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 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. A temporary disconnection or unavailability of a member that would halt activity in a traditional eager system might go unnoticed in an epidemic approach. Our approach is therefore appropriate for e-commerce on the Internet. Since the members connect for a short time to exchange epidemic messages, then disconnect, this model is also well suited for supporting users in mobile and disconnected environments.

Appendix A

General Concepts Used in Proofs

A transaction should be considered to "occur" at precommit time. For example, local serialization order at each site is the same as precommit order at that site. Additionally, events in the epidemic model related to a transaction take place at that transaction's precommit time. A transaction receives a vector timestamp when it precommits at its initiating site. A transaction is considered to be received at a remote site when it attempts to precommit on that site. This attempt may result in a precommit or abort because of conflicting transactions. Regardless, the transaction is considered to be received at that point. The term "precommit time" will be used to designate a precommit attempt which may result in a precommit or abort. By extending Wuu and Bernstein's log and time table algorithm, the following properties are enforced:

- In our algorithms, the operations to precommit, abort, assign a timestamp, and place a record in the log for a transaction are always performed together in a critical section where that transaction has exclusive access to the time table and log. Therefore, they can be considered as an atomic action.
- When a transaction, t, precommits on its initiating site, it is assigned a vector timestamp and placed in the local log. The vector timestamp is assigned such that it dominates exactly those transactions that were initiated on or received by t's initiating site before t. Using the definition that a transaction occurs or is received at precommit time, this is exactly the set of

transactions that precommitted/aborted on t's initiating site before t.

- Transactions are received by a site in some order that respects causal order and are precommitted and placed in the log on that site in the order they are received.
- On its initiating site, a transaction, t, is placed in the log when it precommits. At this time, the set of transactions in the log is all the transactions that have precommitted/aborted on that site before t. This is exactly the set of transactions that causally preceed t. Therefore, locally initiated transactions are also placed in the log in causal order so the whole log respects causal order.
- We assume a local concurrency control protocol at each site which uses precommit as the serialization point for each transaction. The order in which transactions are placed in the log is the same order in which they precommit on that site which is the same as serialization order on that site. Therefore, serialization order on all sites respects causal order.
- Finally, the epidemic model assumes that there is sufficient communication connectivity so that all log and time table information is eventually received by all sites.

Appendix B
Correctness of ROWA Epidemic Transactions

Lemma 1. *Conflicting transactions cannot commit on any site.*

Proof. To commit a transaction, a site must have knowledge that the transaction has been received by all sites. Consider two conflicting transactions, t and t'. To commit t, a site must know that the initiating site of t' has received t. This information must come from an epidemic message which is causally preceeded by the initiating site of t' receiving t. To be conflicting, t and t' must be causally concurrent. This means that t' must have precommitted at its initiating site before t arrived and so must also causally preceed any message containing the information that the initiating site of t' has received t.

Therefore, any message containing the information that the initiating site of t' has received t cannot be received before receiving the records for both t and t'. Whichever of these records arrives second will cause both transactions to abort before either could commit. If the log records of t or t' are received in the same message as the information that the initiating site of t' received t, then all log records of that message are processed and abort conflicting transactions before time table information is processed to commit any transaction. So, the conflict will still be detected and both transactions aborted. □

Lemma 2. *All conflicting transactions will be aborted on all sites.*

Proof. Neither transaction can ever commit on any site. Whenever two conflicting transactions both arrive at a site, the second one to arrive will find the other one in the log and both will abort. By the epidemic assumptions, both records will eventually arrive at all sites and abort at all sites. □

Lemma 3. *All nonconflicting transactions that have precommitted at their initiating site will be committed on all sites.*

Proof. Once a transaction has precommitted on its initiating site, the only way it can be aborted is by the existence of a conflicting transaction so nonconflicting transactions will never be aborted. By the epidemic assumptions, the log record of a nonconflicting transaction will arrive at all sites. When a transaction log record is received at a remote site, its lock requests are forced and it never waits on another transaction. Therefore, all nonconflicting transactions will precommit at all sites. The time table information that a nonconflicting transaction has arrived at all sites will also eventually arrive at all sites and the transaction will commit at all sites. □

Theorem 1. *Epidemic transactions do not suffer from distributed deadlock.*

Proof. All transactions that have precommitted on their initiating site will eventually commit or abort. The only transactions that wait for transactions that have not precommitted at their initiating site are other transactions initiated and not yet precommitted at that site. Therefore, deadlock cycles can only exist locally among transactions initiated and not yet precommitted at a site. □

Lemma 4. *The serialization graph at all sites must be identical and a subgraph of causal order.*

Proof. The set of committed transactions on all sites must eventually be the same set of all nonconflicting transactions that precommitted on their initiating sites. If two transactions have conflicting operations, they must be causally related and they must be serialized in causal order at all sites. Therefore, the serialization graph must be the same on all sites and a subgraph of causal order. □

Lemma 5. *The serialization graph of all sites is acyclic.*

Proof. It is a subgraph of causal order which is acyclic. □

Theorem 2. *Epidemic transactions enforce one copy serializability.*

Proof. After aborting conflicting transactions, all sites must reflect the same acyclic serialization graph. Therefore, the whole system appears as a one copy database with a serializable history. □

Appendix C
Correctness of Quorum Epidemic Algorithm

All of the properties derived from Wuu and Bernstein's log and time table algorithm still hold.

Lemma 6. *Once a transaction has precommitted on its initiating site, it will become either commitable at all sites or abortable at all sites.*

Proof. By the epidemic assumptions, the transaction's log record will be received by all sites. When a transaction's log record is received, its intention to write locks are forced and voting on this transaction takes place immediately without waiting for any other transaction. Again by the epidemic assumptions, the vote records will be received by all sites. If the set of *yes* votes constitutes a quorum, the transaction will become commitable at all sites. Otherwise, by definition, the set of *no* votes constitutes an antiquorum and the transaction will become abortable at all sites. □

Lemma 7. *For any pair of conflicting transacitons, at most one will become committable.*

Proof. Every site votes such that it does not vote yes for two conflicting transactions. If two conflicting transactions both have a quorum of yes votes, their quorums must have at least one site in common. This means that site voted yes for two conflicting transactions which is a contradiction. □

Theorem 3. *Quorum epidemic transactions do not suffer from distributed deadlock.*

Proof. Every transaction that has precommitted at its initiating site must become committable or abortable. An abortable transaction can be immediately aborted. A committable transaction must wait for all causally preceeding transactions to commit before itself committing, but this waits-for relationship can never be cyclic because causal order is acyclic. Also, commitable transactions never wait on transactions that have not yet precommitted at their initiating site. The only transactions that wait for transactions that have not precommitted at their initiating site are other transactions initiated and not yet precommitted at that site. Therefore, deadlock cycles can only exist among transactions initiated and not yet precommitted at a site. □

Lemma 8. *The same set of transactions will commit at all sites.*

Proof. The same set of transactions will become committable. A committable transaction will never be involved in deadlock so it will eventually commit. □

Lemma 9. *The serialization graph at all sites must be identical and a subgraph of causal order.*

Proof. The set of committed transactions on all sites must eventually be the same. For any pair of conflicting transactions, at most one commits, so, if two committed transactions have conflicting operations, they must be causally related and they must be serialized in causal order at all sites. Therefore, the serialization graph must be the same on all sites and a subgraph of causal order. □

Lemma 10. *The serialization graph of all sites is acyclic.*

Proof. It is a subgraph of causal order which is acyclic. □

Theorem 4. *Quorum epidemic update transactions enforce one copy serializability.*

Proof. All sites must reflect the same acyclic serialization graph. Therefore, the whole system appears as a one copy database with a serializable history. □

Appendix D

Correctness of Local Execution of Read-Only Transactions

A read-only transaction can execute and commit locally on its initiating site without communication and the system will still enforce external consistency. The intuition behind this fact is that read-only transactions can be serialized before any conflicting transactions without affecting the values which those transactions should have read.

Add a read only transaction to the serialization graphs at all sites and assign conflict edges as follows: At its initiating site, assign conflict edges based on the actual order of execution between the read-only transaction and other read-write transactions. At all other sites, assign edges as if each read operation had happened immediately after the write from which it read. This way, every serialization graph will be consistent with the values read by the read-only transaction.

Lemma 11. *In all serialization graphs, conflict edges to the read-only transaction can only come from causally preceeding transactions.*

Proof. On its initiating site, this must be true because the local concurrency control protocol ensures that any transaction with a write operation preceding the read-only transaction's read operation must be serialized before it and the vector timestamp assignment ensures that any transaction locally serialized before the read-only transaction causally preceeds it.

On other sites, the write immediately before a read was read by the read-only transaction. The transaction which performed that write must causally precede the read-only transaction. Any read-write transactions with conflicting operations must be causally ordered and serialized in that order so any writes to the same variable before that write must come from a transaction that causally precedes the read-write transaction which wrote to the read-only transaction and, so, must also causally preceed the read-only transaction. □

Lemma 12. *In all serialization graphs, conflict edges from the read-only transaction can only go to causally concurrent or succeeding transactions.*

Proof. On its initiating site, the local concurrency control protocol enforces that any write operations to the same variable that occur after a read operation by the read-only transaction must come from a transaction serialized after the read-only transaction. Timestamp assignment enforces that such a transaction cannot causally precede the read-only transaction. On other sites, imagine there was a write to the same variable that happened after the simulated position of the read operation, but came from a transaction that causally preceded the read-only transaction. This transaction would have to causally succeed the transaction from which the read-only transaction read. This means that, on the read-only transaction's initiating site, the write operation would have to occur between the read-only transaction's read operation and the write from which that operation read. This is a contradiction. □

Lemma 13. *After adding a read-only transaction to all sites' serialization graphs, all serialization graphs must be the same.*

Proof. The set of committed transactions at all sites is the same, so the set of committed operations is the same and the set of conflicting pairs of operations is the same. The only issue is the direction of the conflict edges. If a conflict edge is between two read-write transactions, it must respect causal order. If a conflict edge is between the read-only and a read-write transaction, it must be to the read-only transaction if the read-write transaction causally precedes it or from the read-only transaction otherwise. Therefore, all conflict edges must be the same in all serialization graphs, so all graphs must be the same. □

Lemma 14. *The serialization graph of all sites is acyclic.*

Proof. Among the read-write transactions, edges in the serialization graph must respect causal order, which is acyclic, so any cycle must contain the read-only transaction. If a cycle contains the read-only transaction, then a read-write transaction, t, points to a read-only transaction, r, which points to another read-write transaction t', and t' transitively points to t through a chain containing only causally related read-write transactions. So, t' causally preceeds t and t causally preceeds r which implies that t' causally preceeds r, but r has conflict edges to t' which is a contradiction. □

Theorem 5. *Local read-only transactions enforce external consistency.*

Proof. For the set of all read-write transactions and any single read-only transaction, the serialization graph at all sites is identical and acyclic, so it enforces one-copy serializability. □

ACKNOWLEDGMENTS

This research was partially supported by the US National Science Foundation under grant numbers EIA98-18320, CCR97-12108, IIS98-17432, and IIS99-70700.

REFERENCES

[1] A. Adya and B. Liskov, "Lazy Consistency Using Loosely Synchronized Clocks," *Proc. ACM Symp. Principles of Distributed Computing,* pp. 73-82, Aug. 1997.

[2] D. Agrawal and A.J. Bernstein, "A Non-Blocking Quorum Consensus Protocol for Replicated Data," *IEEE Trans. Parallel and Distributed Systems,* vol. 2, no. 2, pp. 171-179, Apr. 1991.

[3] D. Agrawal and A. El Abbadi, "Storage Efficient Replicated Databases," *IEEE Trans. Knowledge and Data Eng.,* vol. 2, no. 3, pp. 342-352, Sept. 1990.

[4] D. Agrawal, A. El Abbadi, and R. Steinke, "Epidemic Algorithms in Replicated Databases," *Proc. ACM Symp. Principles of Database Systems,* pp. 161-172, May 1997.

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

[6] R. Agrawal, M.J. Carey, and M. Livny, "Models for Studying Concurrency Control Performance: Alternatives and Implications," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* pp. 108-121, May 1985.

[7] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool, "Replication, Consistency and Practicality: Are These Mutually Exclusive?" *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 484-495, June 1998.

[8] P.A. Bernstein and E. Newcomer, *Principles of Transaction Processing.* Morgan Kaufmann, 1997.

[9] Y. Breitbart, R. Komondoor, R. Rastogi, and S. Seshadri, "Update Propagation Protocols for Replicated Databases," *Proc. 1999 ACM SIGMOD Int'l Conf. Management of Data,* pp. 97-108, June 1999.

[10] Y. Breitbart and H.F. Korth, "Replication and Consistency: Being Lazy Helps Sometimes," *Proc. ACM Symp. Principles of Database Systems,* pp. 173-184, May 1997.

[11] S. Ceri and S. Owicki, "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases," *Proc. Sixth Berkeley Workshop Distributed Data Management and Computer Networks,* pp. 117-129, Feb. 1982.

[12] 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," *Proc. Sixth ACM Symp. Principles of Distributed Computing,* pp. 1-12, Aug. 1987.

[13] D.K. Gifford, "Weighted Voting for Replicated Data," *Proc. Seventh ACM Symp. Operating Systems Principles,* pp. 150-159, Dec. 1979.

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

[15] R.G. Guy, J.S. Heidemann, W. Mak, T.W. Page Jr., G.J. Popek, and D. Rothmeier, "Implementation of the Ficus File System," *Proc. Summer USENIX Conf.,* pp. 63-71, June 1990.

[16] A.A. Heddaya, M. Hsu, and W.E. Weihl, "Two Phase Gossip: Managing Distributed Event Histories," *Information Sciences: An Int'l J.,* special issue on databases, vol. 49, nos. 1, 2, 3, pp. 35-57, Oct./Nov./Dec. 1989.

[17] J. Holliday, D. Agrawal, and A. El Abbadi, "Database Replication Using Epidemic Communications," *Proc. European Conf. Parallel and Distributed Systems (EUROPAR 2000),* pp. 427-434, Aug. 2000.

[18] J. Holliday, D. Agrawal, and A. El Abbadi, "Database Replication Using Epidemic Update," Technical Report TRCS 00-01, Dept. of Computer Science, Univ. of California at Santa Barbara, Jan. 2000. Also available at http://www.cs.ucsb.edu/research/trcs/index.shtml, 2002.

[19] J. Holliday, R. Steinke, D. Agrawal, and A. El Abbadi, "Epidemic Quorums for Managing Replicated Data," Technical Report TRCS 99-32, Dept. of Computer Science, Univ. of California at Santa Barbara, 1999. Also available at http://www.cs.ucsb.edu/research/trcs/index.shtml, 2002.

[20] J. Holliday, R. Steinke, D. Agrawal, and A. El Abbadi, "Epidemic Quorums for Managing Replicated Data," *Proc. 19th IEEE Int'l Performance, Computing, and Comm. Conf. (IPCCC 2000),* pp. 93-100, Feb. 2000.

[21] H.V. Jagadish, A.O. Mendelzon, and I.S. Mumick, "Managing Conflicts between Rules," *Proc. 1996 ACM Symp. Principles of Database Systems,* pp. 192-201, 1996.

[22] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozie, and I. Greif, "Replicated Document Management in a Group Communication System," *Proc. Second Conf. Computer Supported Cooperative Work,* Sept. 1988.

[23] P. Keleher, "Decentralized Replicated-Object Protocols," *Proc. 18th ACM Symp. Principles of Distributed Computing,* pp. 143-151, Apr. 1999.

[24] H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Systems,* vol. 6, no. 2, pp. 213-226, June 1981.

[25] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing High Availability Using Lazy Replication," *ACM Trans. Computer Systems,* vol. 10, no. 4, pp. 360-391, Nov. 1992.

[26] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.

[27] B. Liskov and R. Ladin, "Highly Available Services in Distributed Systems," *Proc. Fifth ACM Symp. Principles of Distributed Computing,* pp. 29-39, Aug. 1986.

[28] F. Mattern, "Time and Global States of Distributed Systems," *Proc. 1988 Int'l Workshop Parallel and Distributed Algorithms,* 1989.

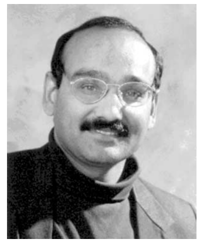
[29] Oracle, *Oracle7 Server Distributed Systems: Replicated Data,* Oracle part number A21903, Oracle, Redwood Shores, CA, Mar. 1994. Also available at http://lina.cli.di.unipi.it:8000/WG73-doc/server/sd273/toc/html, 2002.

[30] K. Petersen, M. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers, "Flexible Update Propagation for Weakly Consistent Replication," *Proc. 16th ACM Symp. Operating Systems Principles,* pp. 288-301, 1997.

[31] M. Rabinovich, N.H. Gehani, and A. Kononov, "Scalable Update Propagation in Epidemic Replicated Databases," *Proc. Int'l Conf. Extending Data Base Technology,* pp. 207-222, 1996.
[32] 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 Trans. Computers,* vol. 39, no. 4, pp. 447-459, Apr. 1990.
[33] F.B. Schneider, "Synchronization in Distributed Programs," *ACM Trans. Programming Languages and Systems,* vol. 4, no. 2, pp. 125-148, Apr. 1982.
[34] W.E. Weihl, "Distributed Version Management of Read-Only Actions," *IEEE Trans. Software Eng.,* vol. 13, no. 1, pp. 55-64, Jan. 1987.
[35] G.T. Wuu and A.J. Bernstein, "Efficient Solutions to the Replicated Log and Dictionary Problems," *Proc. Third ACM Symp. Principles of Distributed Computing,* pp. 233-242, Aug. 1984.

JoAnne Holliday received the bachelor's degree in physics in 1971 from the University of California at Berkeley and the master's degree in industrial engineering from Northeastern University in 1976. She received the PhD degree from the University of California at Santa Barbara in August 2000. She is the Clare Boothe Luce Assistant Professor of computer engineering at Santa Clara University. She worked as a computer programmer/analyst for corporations such as Raytheon, Honeywell, and Unisys before deciding to return to academia. Her research interests are in the area of distributed databases and innovative uses of the Internet. She is a member of the IEEE and the IEEE Computer Society.

Robert Steinke received the BS and MS degrees in computer science from the University of California, Santa Barbara, and the PhD degree in computer science from the University of Colorado, Boulder. He is a software developer at the Jet Propulsion Laboratory currently working on distributed collaboration systems for mission control applications.



Divyakant Agrawal received the PhD degree from the State University of New York at Stony Brook. He is a professor of computer science at the University of California, Santa Barbara. His research interests are in the area of distributed systems, databases, multimedia information storage and retrieval, digital libraries, and Web-based technologies for building large-scale information systems. He is a member of the IEEE and the IEEE Computer Society.

Amr El Abbadi received the undergraduate degree in computer science from Alexandria University and the PhD degree in computer science from Cornell University. In 1987, he joined the Department of Computer Science at the University of California, Santa Barbara, where he is currently a professor. He is currently an editor of *Information Processing Letters* (*IPL*). He was vice chair of the 1999 International Conference on Distributed Computing Systems, vice chair for the International Conference on Data Engineering 2002, and the Americas program chair for the 2000 International Conference on Very Large Data Bases (VLDB). He is a member of the IEEE and the IEEE Computer Society.