Epidemic Algorithms in Replicated Databases* (Extended Abstract)

D. Agrawal
Dept. of Computer Science
University of California
Santa Barbara, CA 93106
agrawal@cs.ucsb.edu

A. El Abbadi
Dept. of Computer Science
University of California
Santa Barbara, CA 93106
amr@cs.ucsb.edu

R. C. Steinke
Dept. of Computer Science
University of California
Santa Barbara, CA 93106
steinke@cs.ucsb.edu

Abstract

We present a family of epidemic algorithms for maintaining replicated data in a transactional framework. 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 in which locks are released as soon as transactions finish their execution locally. However, this optimistic releasing of locks introduces the possibility of cascading aborts while ensuring serializable executions. The last member of this family of epidemic algorithms is motivated from the need for asynchronous replication solutions that are being increasingly used in commercial systems. The protocol 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.

*This research was partially supported by LANL under grant number 6863V0016-3A, by CALTRANS under grant number 65V250A, by NASA under grant number NAGW-3888, and by the NSF under grant numbers IRI94-11330, CDA94-21978, and CCR95-05807.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee

PODS '97 Tucson Arizona USA

Copyright 1997 ACM 0-89791-910-6/97/05 ..$3.50

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 or may not always be connected to the database. One way to provide access to such data is through replication. However, traditional synchronous solutions for managing replicated data [Sto79, Tho79, Gif79] can not be used especially in such a distributed, mobile, and disconnected environment. As the need for replication grows, several vendors have adopted asynchronous solutions for managing replicate data [KBH+88, Ora]. 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. 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 the context of asynchronous replication.

Asynchronous replication has been deployed successfully for maintaining control information in distributed systems and computer networks. For example, nameservers, yellow pages, server directories, etc. are maintained redundantly on multiple sites and updates are incorporated in a lazy manner [FM82, WB84] through gossip messages [LL86, HHW89], epidemic propagation and anti-entropy [DGH+87]. 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 per-

form 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 are designed to satisfy a level of consistency weaker than serializability [BHG87]. 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. Epidemic techniques are useful but are too weak for supporting transaction processing. 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 between 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 non-serializable 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. Rabinovich et al. [RGK96] 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 non-serializable executions in a transactional framework.

In this paper we present a family of epidemic algorithms for maintaining replicated data in a transactional framework. 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 in which locks are released as soon as transactions finish their execution locally. However, this optimistic releasing of locks introduces the possibility of cascading aborts while ensuring serializable executions. The last member of this family of epidemic algorithms is motivated from the need for asynchronous replication solutions that are being increasingly used in commercial systems. The protocol 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. Resolution of inconsistencies is left to the application, and the details of specific conflict compensation schemes are orthogonal to the ideas of this paper.

The paper is organized as follows. In the next section, we present the epidemic model of replication. In Section 3, we develop an epidemic algorithm for transaction processing that guarantees serializability by checking for both read-write as well as write-write conflicts. The optimistic variants of this algorithm are presented in Section 4. Section 5 concludes with a discussion of our results.

2 The Epidemic Model of Replication

We consider a distributed system consisting of n sites $S_1, S_2, ..., S_n$ 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 message are not corrupted. Each site performs three kinds of operations: send operations, receive operations, and database specific non-communication operations. An event model [Lam78] 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 [Lam78] which is a partial order on all operations in E. The happened-before relation is the transitive closure of the following two conditions:

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

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

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

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

Epidemic algorithms use this execution model to maintain replicated data such as dictionaries, name-servers, distributed calendars, and databases [FM82, AM83, WB84, LL86, DGH+87, HHW89, AE90a, AE90b, SA93, AM91, RGK96, TDP+94, TTP+95]. In general, database specific operations are executed locally and they are communicated to the other database sites by using the epidemic model of communication. 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 [FM82, AM83, WB84, LL86, DGH+87, HHW89]. Epidemic algorithms generally are implemented using vector clocks [Mat89] or event logs [Sch82] to ensure this property. Vector clocks are an extension of Lamport clocks [Lam78] and ensure the following property:

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

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 [Sch82, AM83, WB84].

Wuu and Bernstein [WB84] combine the logs and vector clocks to solve the distributed dictionary problem efficiently. Each site S_i keeps a two-dimensional time-table T_i, which corresponds to the vector clocks of all sites, 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). Thus, the time-table can be used to define the following predicate for a record t corresponding to some event:

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

which is referred to as the *time-table property*. That is, the k^{th} row of T_i is the knowledge of S_i about S_k's knowledge of events in the system. In Wuu and Bernstein's algorithm when a site S_i performs an update operation it places an event record in the log recording that operation. When S_i sends a message to S_k it includes all records t such that $HasRecvd(T_i, t, S_k)$ is false, and it also includes its time-table T_i. When S_i receives a message from S_k it applies the updates of all received log records and updates its time-table in an atomic step to reflect the new information received from S_k. When a site receives a log record it knows that the log records of all causally preceding events either were received in previous messages, or are included in the same message which is referred to as the *log property* which is stated as follows with respect to a local copy of the log L_i at site S_i:

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

The correctness of the algorithm can be established by using both the log and the time-table property. A variant of the log and time-table algorithm called the two-phase gossip protocol [HHW89] reduces the size of the two-dimensional time-table from n^2 to $2 \cdot n$.

3 Epidemic Transactions for Replicated Databases

There are two elements of a replicated database that are not fully supported by current epidemic techniques.

- Transactional integrity: Transactions can have more than one operation, and these operations must be executed atomically.
- Correctness: Replicated databases must enforce one-copy serializability [BHG87].

In this section, we develop an algorithm that incorporates transactions into the epidemic framework.

3.1 From causality to serializability

Our system model is based on the epidemic model with the following extensions. Each site is assumed to have a local concurrency control mechanism that ensures serializability. 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 of the applications used in the context of epidemic algorithms circumvent this problem by only permitting commutative operations on the database. For example, in the distributed dictionary problem, even though two insert operations are potentially conflicting (if they insert the same item), they are forced to be commutative by associating a unique identifier (based on site and time information).

In order to adapt the epidemic model of communication to transactions, we need to devise mechanisms 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 ensure that transactions can execute concurrently as long as they do not have any conflicting operations. In the optimistic approach, we detect if concurrent transactions executed conflicting operations. Conflict resolution is left to the application or to the user. For example, Oracle 7 provides a choice of twelve reconciliation rules to merge conflicting updates on replicated data [GHOS96, Ora].

We define *conflicting transactions* to be two or more transactions that are concurrent and have conflicting operations. In order to detect conflicting transactions we need to deal with two issues related to transactions. First, the read and write sets of all transactions at all sites must be identified. Second, for a given transaction the set of concurrent transactions must be determined. Since the read and write sets are needed to determine conflicts among transactions that executed concurrently, they can be easily constructed locally as transactions execute on a site. Determining the set of concurrent transactions is more complex since for a given transaction there can be potentially conflicting transactions anywhere in the network. Ascertaining this information through explicit synchronization will be prohibitively expensive. Furthermore, simple timestamping of transactions based on Lamport clocks is not sufficient since Lamport clock based timestamps guarantee order only when there is a causal relationship among transactions. These timestamps do not provide sufficient information to determine if two transactions are concurrent. Vector clocks, on the other hand, 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 the vector clock for identifying conflicting transactions asynchronously.

3.2 A naive 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 pre-commits. At pre-commit time t acquires a timestamp using the vector clock of S_i which is the i^{th} row of the S_i's time-table, i.e., $T_i[i,*]$, with the i^{th} component incremented by one. This timestamp assignment ensures that t dominates all those transactions that have already pre-committed on S_i regardless of where they were initiated. In this way t can differentiate concurrent transactions from those that causally preceded it. A pre-commit 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 executed, denoted, $Site(t) = S_i$.
2. The timestamp assigned to t, denoted $TS(t) = T[i,*]$.
3. The read set of t, $RS(t)$, and write set of t, $WS(t)$. Note that $WS(t)$ includes the values written by t.

After inserting the pre-commit 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 pre-commit 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 $TS(t') <> TS(t)$) which indicates that t' executed concurrently with t.
- The intersections of $RS(t)$ and $WS(t')$, $WS(t)$ and $WS(t')$, or $WS(t)$ and $RS(t')$ is nonempty. This indicates that t and t' execute conflicting operations.

If there exists such a transaction t', then t and t' are aborted at S_j. An abort record 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 record to the local log, and updates the local copy of the time-table to reflect the receipt of t. Finally, S_j inserts another pre-commit record of t in the log.

As the epidemic process evolves, 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 has pre-commit records from all the sites in the network, 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 [BHG87, Ske82]. 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 pre-commits on a site no new concurrent transactions can be initiated on that site by the nature of causality. A transaction must pre-commit 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 pre-commit on all sites. By aborting all conflicting transactions the algorithm guarantees one-copy-serializability.

3.3 The Pessimistic Epidemic Algorithm

In its initial formulation, the above algorithm creates a large number of log records that must be stored and transmitted to other sites. For each transaction, each site inserts a pre-commit or an abort record in the log which results in n records per committed transaction. By exploiting the properties of the two-dimensional time-table, the number of records per transaction can be reduced from n to 1. A site initiating a transaction inserts a pre-commit record in the log and communicates it to other sites via epidemic messages. Other sites also insert an explicit pre-commit or abort records to acknowledge the receipt of the original pre-commit record. However, the time-table encodes enough information to carry these acknowledgements implicitly through the timing information rather than explicitly through the log records.

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 pre-committed or aborted t, and S_i has received the time-table information in a message which causally succeeded the pre-commit or abort of t at S_k. In the above protocol, S_i must have therefore received the pre-commit or abort record inserted at S_k by this time. Hence, pre-commits can be deduced from the timing information and the lack of an explicit abort record. In this formulation, only aborts need to be explicitly acknowledged. 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, we now demonstrate that explicit abort records are also 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 pre-committed. 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 pre-commit record, needs to be created for each transaction. The propagation of that record and the return propagation of time-table information will detect conflicting transactions in its path. We include a field, denoted *aborted* in the pre-commit record as an optimization. If a pre-commit record of a transaction t arrives at site S_j and S_j aborts t on account of a conflicting transaction t', the aborted flag of t is set. Subsequently, when S_j forwards the pre-commit record of t to S_k, S_k uses the flag to avoid pre-committing t locally. However, t is still processed against the log at S_k to ensure any conflicting transactions with respect to t are aborted at S_k.

We define the following predicates using the information that is included in the log and the time-table of every site. The first predicate holds when t should be aborted:

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

Similarly, we define another predicate which holds when a transaction successfully commits at a site. This predicate is defined as follows:

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

The resulting epidemic algorithm to execute transactions on a replicated database is shown in Figures 2 and 3. Figure 2 illustrate the handling of transaction execution locally at a site. Although the algorithm is illustrated by using read and write sets, it can be easily modified so that the read set 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 pre-commits (i.e., makes its writes recoverable at S_i) and inserts a pre-commit 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 pre-committing, 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 pre-commit records appears in Figure 1.

Figure 3 illustrates the communication model used by the epidemic algorithm to execute transactions at a particular site S_i. The send function is used by site S_i to disseminate the pre-commit 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.

The receive procedure is the most interesting. Transactions are processed one at a time in the order of the received log. The receiving site first checks if it already has the pre-commit 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 time-table 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, pre-commits. After all records are processed the procedure updates the other rows of the time-table. Next, the pre-commit records in L_i are checked to determine if they can be committed locally by using the *Commit* predicate. Finally, 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 [Ste97].

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 would be aborted by procedure $ForceWriteLocks$. Waits for dependencies between pre-committed transactions must be a subgraph of the happened-before relation which is acyclic, and pre-committed transactions cannot wait on local transactions. Therefore, pre-committed transactions cannot be involved in deadlock. Any transaction holding locks on more than one site must be pre-committed, and therefore any cycles in the global waits-for graph must involve only local transactions at a single site.

4 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 pre-commit 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. However, given this assumption we can relax this requirement by avoiding the necessity of holding locks till transaction termination. This algorithm still ensures both serializability and recoverability [Had88]. 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 [KBH+88, Ora]. In this approach transactions are optimistically committed as soon as they terminate at the initiating site. Epidemic propagation is primarily used to disseminate the effects of these committed transactions to other sites so that the replicated data becomes mutually consistent. In addition, this mechanism is used to detect conflicting transactions and such transactions are

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

Figure 1: A Transaction Record

```
Persistent data:
    clock_i   :   TimeType INITIALIZED TO 0;
    T_i       :   array [1..n, 1..n] of TimeType INITIALIZED TO 0;
    L_i       :   setof Transaction INITIALIZED TO ∅;

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;
```

Figure 2: 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)
            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;
```

Figure 3: Epidemic propagation of transaction records from other nodes to N_i

reported to the application level for manual or automatic compensation which is application dependent.

4.1 Optimistic Releasing of Locks

The sole purpose of holding write locks after pre-commit 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 [Had88, BHG87].

Thus if we modify the epidemic algorithm so that write locks are released at pre-commit, 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 read from must have causally preceded it. Therefore, any site which could pre-commit t must have received and either pre-committed 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 read from. For t to commit it must have pre-committed on all sites. This implies that all transactions from which t read from are also pre-committed 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. However for applications that will use epidemic style of transaction processing, conflicts among transactions are expected to be rare. Thus, the concern that exists in traditional database settings is not very relevant here. By releasing locks early, the epidemic algorithm introduces the vulnerability to cascading aborts but we expect them to be rare resulting in insignificant impact on the performance. On the other hand, by releasing locks early, unnecessary blocking due to locks held during dissemination of pre-commit of transactions is eliminated. Although locks are released at pre-commit, transactions commitment is still as before, i.e., a transaction can commit only after it has been pre-committed successfully at all sites in the network. Hence, transaction execution remains serializable and correctness is not compromised.

4.2 Optimistic Commitment of Epidemic Transactions

Recently, several commercial systems have become interested in replication solutions primarily for supporting distributed, mobile, and disconnected users. In the context of these applications, the synchronous approach for managing replicated data (i.e., read-one write-all or quorum protocol [Tho79, Gif79]) is considered too expensive and not applicable in some cases (e.g., for disconnected operations). Instead, asynchronous replication techniques based on optimistically executing and committing transactions locally is advocated. In the presence of replicated data it is not enough to commit transactions locally rather mechanisms are needed to asynchronously apply the effects of committed transactions to other replica sites. In addition, these mechanisms should also be able to detect inconsistent or conflicting executions. For example, Lotus Notes uses value-based propagation of files to replica sites and uses version numbers to detect inconsistencies. Although this approach is adequate for single file operations it fails in the context of transactions that access multiple data objects. Rabinovich et al. [RGK96] proposed adopting transactions in the context of epidemic algorithms, which are asynchronous solutions for managing replicated data. As pointed out before, the protocol fails to capture inter-transaction dependencies resulting from read-write conflicts.

In this section, we use the epidemic model for transaction processing to introduce the notion of optimistic commitment of transactions. In the previous protocol transactions completed execution locally and delayed the commitment until they are aware that their pre-commit 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. In the case of transaction execution of Figure 2, the main change is that instead of inserting a pre-commit record a transaction executes as if it is a centralized database and hence inserts a commit record and release all its locks. The receive procedure for optimistic operation appears in Figure 4. The significant difference is that transactions are committed at the time they were pre-committed 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 [Ora]. Additionally, there are meta-rule paradigms to specify which rules to apply and in what order if more than one rule is applicable. For example, Jagadish, Mendelzon, and Mumick [JMM96] describe a meta-rule language and inference procedures to determine if a particular set of meta-rules is unambiguous. Using these two techniques it should be possible to create an automated conflict resolution procedure. Specific compensation rules and meta-rules would be application dependent and such issues are orthogonal to the ideas in this paper.

5 Conclusion

Many techniques are currently available to support 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 non-serializable executions. The algorithms can be used in a conservative manner that prevents non-serializable executions, or in an optimistic manner that merely detects non-serializable executions for application specific compensation.

References

[AE90a] D. Agrawal and A. El Abbadi. Integrating Security with Fault-tolerant Distributed Databases. *The Computer Journal*, 33(1):71–78, February 1990.

[AE90b] D. Agrawal and A. El Abbadi. Storage Efficient Replicated Databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):342–352, September 1990.

[AM83] J. E. Alchin and M. S. McKendry. Synchronization and Recovery of Actions. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.

[AM91] D. Agrawal and A. Malpani. Efficient Dissemination of Information in Computer Networks. *The Computer Journal*, 34(6):534–541, December 1991.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, 1987.

[DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 1–12, August 1987.

[FM82] M. J. Fischer and A. Michael. Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. In *Proceedings of the First ACM Symposium on Principles of Database Systems*, pages 70–75, May 1982.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.

[Gif79] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 150–159, December 1979.

[Had88] V. Hadzilacos. A Theory of Reliability in Database Systems. *Journal of the ACM*, 35(1):121–145, January 1988.

[HHW89] A. A. Heddaya, M. Hsu, and W. E. Weihl. Two Phase Gossip: Managing Distributed

```
Receive(m) from S_k :
begin
    let m = ⟨S_k, T_k⟩ ;
    foreach {t|t ∈ S_k ∧ ¬HasRecvd(T_i, t, S_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
            ResolveConflict(t);
            ResolveConflict(t');
            t.inconflict = true;
            t'.inconflict = true;
        endif
        if {∃t' ∈ L_i|t' ∈ t.readfrom ∧ t'.inconflict = true} then
            ResolveConflict(t);
            t.inconflict = true;
        endif
        ForceWriteLocks(t.WS);
        WriteValues(t.WS, t.values);
        Commit(t);
        T_i[i, t.site] := t.time[t.site];
        L_i := L_i ∪ {t};
        end mutex
    endfor
    begin mutex
        ∀K, J T_i[K, J] := max(T_i[K, J], T_k[K, J]);
        L_i := {t|t ∈ L_i ∧ ∃j|¬HasRecvd(T_i, t, S_j)};
    end mutex
end;
```

Figure 4: The receive procedure for optimistic operation for S_i

Event Histories. *Information Sciences: An International Journal*, 49(1,2,3):35–57, October/November/December 1989. Special issue on Databases.

[JMM96] H. V. Jagadish, A. O. Mendelzon, and I. S. Mumick. Managing Conflicts between Rules. In *Proceeedings of the 1996 ACM Symposium on Principles of Database Systems*, pages 192–201, 1996.

[KBH+88] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozie, and I. Greif. Replicated Document Management in a Group Communication System. In *Proceedings of the 2nd Conference on Computer Supported Cooperative Work*, September 1988.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[LL86] B. Liskov and R. Ladin. Highly Available Services in Distributed Systems. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 29–39, August 1986.

[Mat89] F. Mattern. Time and global states of distributed systems. In *Proceedings of the 1988 International Workshop on Parallel and Distributed Algorithms, Bonas, France.* North Holland, 1989.

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

[RGK96] M. Rabinovich, N. H. Gehani, and A. Kononov. Scalable update propagation in epidemic replicated databases. In *Proceedings of the International Conference on Extending Data Base Technology*, pages 207–222, 1996.

[SA93] O. T. Satyanarayanan and D. Agrawal. Efficient Execution of Read-Only Transactions in Replicated Multiversion Databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):859–871, October 1993.

[Sch82] F. B. Schneider. Synchronization in Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, April 1982.

[Ske82] D. Skeen. Non-blocking commit protocols. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 133–147, June 1982.

[Ste97] Robert C. Steinke. Epidemic Transactions for Replicated Databases. Master's thesis, University of California at Santa Barbara, Department of Computer Science, UCSB, Santa Barbara, CA 93106, 1997. In preparation.

[Sto79] M. Stonebraker. Concurrency Control and Consistency in Multiple Copies of Data in Distributed INGRES. *IEEE Transactions on Software Engineering*, 3(3):188–194, May 1979.

[TDP+94] M. Theimer, A. J. Demers, K. Petersen, M. Spreitzer, and C. H. Hauser. Dealing with tentative data values in disconnected work groups. In *Proceedings of the Workshop on Mobile Computing Systems abd Applications*, pages 192–195, 1994.

[Tho79] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transaction on Database Systems*, 4(2):180–209, June 1979.

[TTP+95] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, 1995.

[WB84] G. T. Wuu and A. J. Bernstein. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 233–242, August 1984.