Two Epoch Algorithms for Disaster Recovery

Hector Garcia-Molina and Christos A. Polyzois
Department of Computer Science, Princeton University, Princeton, NJ 08544 USA

Robert Hagmann
Xerox PARC, Palo Alto, CA 94304 USA

ABSTRACT

Remote backup copies of databases are often maintained to ensure availability of data even in the presence of extensive failures, for which local replication mechanisms may be inadequate. We present two versions of an epoch algorithm for maintaining a consistent remote backup copy of a database. The algorithms ensure scalability, which makes them suitable for very large databases. The correctness and the performance of the algorithms are discussed, and an additional application for distributed group commit is given.

1. Introduction

A remote backup (or *hot standby* or *hot spare*) is a technique used in critical applications for achieving truly continuous operation of databases. A copy of the primary database is kept up-to-date at a geographically remote site and takes over transaction processing in case the primary site fails. The geographic separation of the two copies provides significantly more failure isolation than what is available with local replication. The advantages of a remote backup are discussed in detail in [5], [12].

We focus on a particular type of remote backup, called *1-safe* [9], [12]: transactions first commit at the primary site, release the resources they hold, and are then propagated to the backup and installed in the backup copy. This means that in case of disaster, some transactions that were executing close to the occurrence of the disaster may never reach the backup, although they may have

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

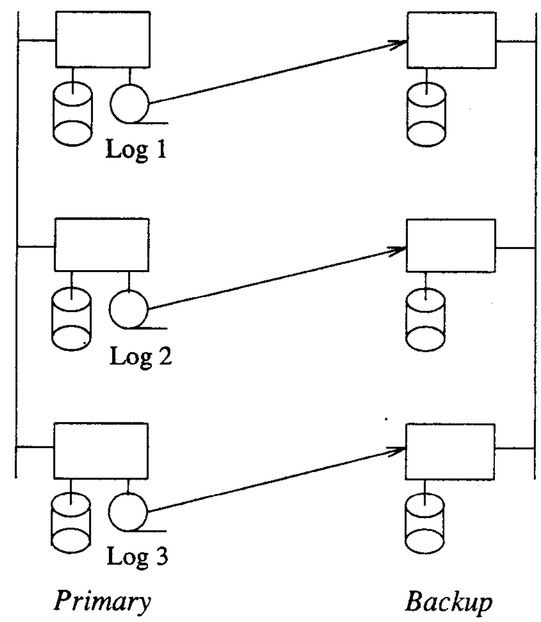

Figure 1. Architectural Framework

committed at the primary. The other option would be to run *2-safe* transactions: the primary and the backup run a two-phase commit protocol [8], [13] to ensure that transactions are either installed at both sites or are not executed at all. The commit protocol increases the response time of transactions by at least one round trip delay. For this and for other reasons [12], [5] many systems prefer to use 1-safety and lose some transactions in case of disaster rather than pay the overhead for 2-safety.

Existing 1-safe remote backup systems (e.g., Tandem RDF [15]) usually address only the simple case, with one primary and one backup computer. Data logs are propagated from the primary to the backup to enable the latter to install the same changes that were installed at the former. Even when multiple computers are allowed in existing systems, the situation is similar, because the logs are merged (implicitly or explicitly) into a single log stream. A single log scheme is undesirable, because the merging of the logs may eventually become a bottleneck in the performance of the system.[1] Instead, we would like to have many computers at each site, with multiple independent log streams from the primary to the backup (as shown in Fig. 1), so that the system can scale up to very large databases. In this figure, the log at each primary computer records the actions that take place there, just like in any system. The writes in this log are then transmitted and replayed at the corresponding backup computer.

When multiple logs are used, as in Figure 1, we actually need a distributed commit process at the backup to decide when the writes for a given transaction can be safely installed. In other words, it is not possible to simply install the writes at each backup computer as they arrive.

To illustrate this, consider three transactions, T_1 T_2, and T_3. Assume transaction T_1 wrote data at primary computer P_i. Then T_2 wrote at primary computers P_i and P_j, while later T_3 wrote at P_j and P_k. All three transactions commit at the primary. As the logs are being propagated, a disaster hits, and only the log records shown in Fig. 2 make it to the backup. (The tail of the logs is at the bottom.) BP_i is the hot standby for P_i, BP_j for P_j, and so on. The prepare record for a transaction T_x is represented by $P(T_x)$ and the commit record by $C(T_x)$, while the notation *write* T_x is used for a write action made by transaction T_x. Backup computer BP_i receives the write records for T_1, but cannot install them because the commit record, $C(T_1)$, was not received. (It does not know whether T_1 committed or not at the primary.) Site BP_j does see the $C(T_2)$ record, but it is still unable to install the T_2 writes (i.e., unable to commit T_2 at the backup). This is because the actions of T_2 did not arrive at BP_i due to the crash. (Recall that T_2 wrote at P_i and P_j.) For example, suppose

$\mathbf{BP_i}$	$\mathbf{BP_j}$	$\mathbf{BP_k}$
write T_1	write T_2	write T_3
	$P(T_2)$	$P(T_3)$
	$C(T_2)$	$C(T_3)$
	write T_3	
	$P(T_3)$	
	$C(T_3)$	

Figure 2.

[1]It is not necessarily the bandwidth of the single line to the backup that is a problem: very high bandwidth lines are available. The bottleneck could be the processing load at the multiplexing computer, which needs to receive streams of messages from the local computers, merge them (in an *appropriate* order) and repackage them for transmission to the backup. In many cases, the telecommunication protocols, both local and wide-area to the backup, are very high overhead.

T_2 is a funds transfer transaction, with the write at P_i being the withdrawal from $Account_1$ and the write at P_j being the deposit in $Account_2$. Then we do not want to execute only the deposit without the withdrawal.

Since T_2 cannot be installed, it may also be impossible to install T_3. It is possible that T_3 read data produced by T_2, so installing T_3 may compromise the consistency of the database, even though all of the write and commit records of T_3 arrived at the backup site. Not installing T_3 introduces a divergence from the real world, but seems the lesser of two evils. Transactions that cannot commit at the backup because they would violate consistency can be saved and fixed "manually" (by a human operator or a special program).

In summary, before a transaction can commit at the backup, the backup computers must run a commit protocol that detects the problems illustrated in our example. This protocol is in essence a two-phase commit among the backup computers. When a computer involved in a transaction T knows it is feasible to install T's writes (because T's commit record has been received, and all transactions that T could depend on have committed), then it sends a message to a coordinator backup computer. Once the coordinator gets acknowledgments from all participants in T, it can send a commit message telling them to actually install the writes.

The backup commit protocol can be run for each transaction individually (as done in [4], [6]), or instead, it can be run for a batch of transactions. This is the approach we follow here. Our method makes a commit decision for a group of transactions at a time, amortizing the cost over them.

The rest if the paper is organized as follows: in section 2 we give our framework and in sections 3-6 we present two versions of our algorithm and prove their correctness. In section 7 we discuss the features of the algorithms, and in section 8 we give a related application to distributed group commit for large memory computers.

2. Our Framework

2.1. Architecture

In our model there are two sites (primary and backup) with multiple computers each. (It is possible to have multiple backups for a single primary, but for ease of explanation we assume just one backup. The extension to multiple backups is straightforward.) Each computer has one or multiple processors, holds part of the (local) database and runs a DBMS. All of the computers at each site can communicate with each other through shared memory or through a local network or bus. This makes our method applicable to shared memory architectures as well as to more loosely coupled systems. Running between the two sites are several communication lines, which let computers at the primary site send copies of operations being performed to the backup computers. Control messages are also exchanged over these lines. No particular assumption is made about the delays in the network, but the bandwidth is assumed sufficient for the propagation of the logs. We assume an one-to-one correspondence between primary and backup computers. (This is again for ease of explanation. See section 9.) As in our example, we use the notation P_i for a processor at the primary and BP_i for its peer at the backup.

As failures occur at the primary, the system tries to recover and reconfigure (possibly using some local mechanisms). However, multiple and/or significant failures may slow down the primary site or even stop it entirely. At this point, a *primary disaster* is declared and the backup attempts to take over transaction processing. The declaration of the disaster will in all likelihood be done by a human administrator. This is mainly because it is very hard for the backup site to distinguish between a catastrophic failure at the primary and a break in the communication lines. In addition, the input transactions must now be routed to the backup site. (In practice, user "terminals" keep two open connections, one to the primary and one to the backup. The backup connection is on standby until a disaster occurs.)

Our failure model for this paper only considers disasters of the primary. That is, the computers at the backup never fail. During normal processing, they receive and process logs from the primary. When a disaster is declared, the backup computers finish installing the available logs, and then go into primary mode and process transactions. Our algorithms can be extended to cope with other failure scenarios (e.g., a backup computer fails and its duties are taken over by another one, or a single primary computer fails and its backup takes over its duties only). Due to space limitations, we do not address such failure scenarios here.

Regardless of the backup strategy used, a local two-phase commit protocol must be used at the primary to ensure atomicity. The coordinator for a transaction T notifies the participants that the end of the transaction has been reached. Those participants that have executed their part of T successfully make a *prepare* entry in their logs (we use the notation $P(T)$ for prepare entries) and send a positive acknowledgement (*participant − ready* message) to the coordinator. We assume that the $P(T)$ entry includes the identity of the coordinator. Participants that were not able to complete their part of T successfully write an *abort* entry in their logs and send a negative acknowledgement to the coordinator. If a positive acknowledgement is received from all participants, the coordinator makes a *commit − coordinator* entry in its log (we use the notation $CC(T)$ for this) and sends a *commit* message to all participants. The participants make a *commit − participant* entry in their logs (symbolically $CP(T)$) and send an acknowledgement to the coordinator. Sometimes we use the notation $C(T)$ for a commit entry written in a log when we do not want to specify if it was written by the coordinator or a participant.

A concurrency control mechanism ensures that the transaction execution schedule at the primary is serializable. We say a dependency $T_x \rightarrow T_y$ exists between two transactions T_x and T_y if both transactions access a common data item and at least one of them writes it [1], [10].

The logs, including the $P(T)$, $C(T)$ and the write entries (giving the new values written by the transactions) are propagated to the backup site, where the writes have to be installed in the database. The backup will in general execute a subset of the actions executed at the primary. Read actions do not modify the database, so they need not be propagated to the backup. We also assume that write actions at the backup install the same value that was installed by the corresponding write actions at the primary. We use the notation $W(T, d)$ to represent the write at the backup of data item d by transaction T.

2.2. Correctness criteria

Before we proceed with our solution, let us define more precisely what a "correct" backup is. Our first requirement for the backup is *transaction atomicity*, the second one is *consistency*.

Requirement 1: Atomicity. If $W(T_x, d)$ appears in the backup schedule, then all of the T_x's write actions must appear in the backup schedule.

Requirement 2: Consistency.[2] Consider two transactions T_i and T_j such that at the primary $T_i \rightarrow T_j$. Transaction T_j may be installed at the backup only if T_i is also installed (*local consistency*: dependencies are preserved). Furthermore, if they both write data item d, $W(T_i, d)$ must occur before $W(T_j, d)$ at the backup (*mutual consistency*: the *direction* of dependencies is preserved).

Finally, we would like the backup to be as close to the primary as possible. This is formally stated in the following requirement, which guards against a backup that trivially satisfies all of the previous requirements by throwing away all transactions:

Requirement 3: Minimum Divergence. If a transaction is not missing at the backup and does not depend on a missing transaction, then its changes should be installed at the backup.

3. Overview of the Epoch Algorithm

The general idea is as follows: periodically, special markers are written in the logs by the primary computers. These markers serve as delimiters of groups of transactions (*epochs*) that can be committed safely by the backup.

[2]This consistency criterion is stronger than the one in [4], [6]. Only the weaker criterion is actually necessary, but our algorithm guarantees this stronger version.

The primary computers must write these markers in their logs in some synchronized way. Each backup computer waits until all backup computers have received the corresponding portion of the transaction group, i.e., all backup computers have seen the next delimiter. Then, each computer starts installing from its log the changes for the transaction group. This installation phase is performed (almost) independently from other processors.

In the log, each delimiter includes an integer that identifies the epoch that is ending. We represent the delimiter as a small circle with the epoch number as a subscript, e.g., $\bigcirc_n$ is the delimiter at the end of epoch n. At the primary, each computer i keeps track of the current epoch number in a local counter $E(i)$. One computer is designated as the *master* and periodically makes a $\bigcirc_n$ entry in its log (where n is the current epoch number), increments its epoch counter from n to $n+1$ and broadcasts an $end-epoch(n)$ message to all nodes at the primary. All recipient nodes also make a $\bigcirc_n$ entry in their logs, increment their epoch counters and send an acknowledgement to the master. The master receives acknowledgements from all other nodes before it repeats the above process to terminate another epoch.[3]

It is important to note that simply writing circles does not solve the problem, i.e., there is more that has to be done. As we have described it, transactions can straddle the end-epoch markers, as shown in the sample logs of Fig. 3 (again, the last record received at the backup is at the bottom). If epoch n is committed at the backup, the updates of T_1 at BP_j are installed. However, the updates of T_1 at BP_i appear after the end-epoch marker and will not be installed. This violates atomicity.

There are two ways to avoid the undesirable situation described above. The first way is to let transaction processing proceed normally and place the delimiters more carefully with respect to the log entries for actions of transactions. The second way is to write the delimiters asynchronously but to delay some actions of some transactions,

BP_i	**BP_j**
$\bigcirc_n$	write T_1
write T_1	$P(T_1)$
$P(T_1)$	$CC(T_1)$
$CP(T_1)$	$\bigcirc_n$

Figure 3

[3]This is not necessary in one algorithm, the single mark algorithm, to be described. However, we make the assumption to simplify the discussion.

so that the log entries for these actions will be placed more carefully with respect to the delimiters. These two options give rise to two versions of the *epoch algorithm*, which are described in sections 4 and 6 respectively.

In what follows, when we want to specify the processor where an event took place and a log entry was written, we add an extra argument to the log entry. For example, the notation $\bigcirc_n(P_i)$ denotes the event when P_i writes a $\bigcirc_n$ entry in its log. Similarly, $C(T, P_i)$ denotes the event when processor P_i writes a commit entry for transaction T in its log.

We use the symbol "$\Rightarrow$" to denote the "occurs before" relation, i.e., $A \Rightarrow B$ means that event A occurred before event B [11]. When using a relation $A \Rightarrow B$, we do not distinguish whether A and B are the actual events or the corresponding entries in the log; we assume the log preserves the relative order of events (within the same computer). Do not confuse the "$\Rightarrow$" symbol with the symbol "$\rightarrow$" used for transaction dependencies. Suppose a dependency $T_x \rightarrow T_y$ exists at processor P_d between two transactions T_x and T_y. Transactions T_x and T_y were coordinated by processors P_x and P_y. (Note that P_d, P_x and P_y need not all be different processors.) We assume the following property relates the two symbols "$\Rightarrow$" and "$\rightarrow$":

If $T_x \rightarrow T_y$ at P_d, then $CC(T_x, P_x) \Rightarrow CP(T_x, P_d)$
$\Rightarrow P(T_y, P_d) \Rightarrow CC(T_y, P_y)$ (Property 1)

We prove the property in the case when strict two-phase locking is used for concurrency control. Transaction T_x does not release its locks until it commits, and transaction T_y cannot commit before T_x releases its locks, because the two transactions access some common data in conflicting modes and therefore ask for incompatible locks. Thus, P_x writes the *commit* message for T_x in its log, then P_d (if different from P_x) writes the *commit* message for T_x, the locks are released, T_y runs to completion, P_d writes the *prepare* entry for T_y, and finally P_y writes the commit message for T_y. Thus, the property holds for strict two-phase locking. Other concurrency control mechanisms also satisfy this property, but we will not discuss them here.

4. The Single Mark Algorithm

In this section we describe the first version of the epoch algorithm, where we place circles in the log more carefully. Circles are still generated as described in the previous section, but some additional processing rules are followed. When a participant processor i writes a *prepare* entry in its log and sends a $participant-ready$ message to the coordinator of a transaction, the local epoch number $E(i)$ is included in the message. Similarly, the epoch number is included in the *commit* message sent by the coordinator to the participants of a transaction. When a message containing an epoch number n arrives at its destination j, it is checked against the local epoch counter. If $E(j) < n$, it is inferred that the master has broadcast an

$end-epoch(n-1)$ message which has not arrived yet. Thus, the computer acts as if it had received the $end-epoch(n-1)$ message directly: it makes a $\bigcirc_{n-1}$ entry in its log, sets $E(j)$ to n, sends an acknowledgement to the master and *then* processes the incoming message. If the $end-epoch(n-1)$ message is received later directly from the master (when $E(j) > n-1$), it is ignored. The idea of bumping an epoch when a message with a larger epoch number is received is similar in principle to bumping logical clocks [11].

The logs (including the *circle* entries) are propagated to the backup site. As the logs arrive at a backup processor, they are saved on stable storage. The backup processor does not process them immediately. Instead, it waits until a $\bigcirc_n$ mark has been seen by all backup computers in their logs. This can be achieved in various ways. For example, when a computer receives a $\bigcirc_n$ mark, it broadcasts this fact to other computers and waits until it receives similar messages from everybody else. Alternatively, when a computer sees a $\bigcirc_n$ in its log, it notifies a master at the backup site. The local master collects such notifications from all computers and then lets everyone know that they can proceed with installing the logs for epoch n.

To install the transactions in epoch n, BP_i examines the newly arrived log entries from $\bigcirc_{n-1}$ to $\bigcirc_n$. However, there can also be entries pending from previous epochs (before $\bigcirc_{n-1}$) that need to be examined. These entries correspond to transactions that did not commit in previous epochs. Thus, at the end of epoch n, BP_i examines *all* of the log records appearing before $\bigcirc_n$ corresponding to transactions that have not been installed at BP_i. The following rules are used to decide which new transactions will commit as part of epoch n:

- For a transaction T, if $C(T) \Rightarrow \bigcirc_n$ in the log, a decision to commit T is made.
- If a transaction T does not fall in the above category but $P(T) \Rightarrow \bigcirc_n$ in BP_i's log ($P(T)$ could possibly be in some *previous* epoch), the decision whether to commit T depends on whether P_i was the coordinator for T at the primary. (Recall from the model section that the coordinator is included with every $P(T)$ log entry.) If P_i was the coordinator, T does not commit at the backup during this epoch. If some other processor P_j was the coordinator at the primary, a message is sent to BP_j requesting its decision regarding T. (BP_j will reach a decision using the rules we are describing, i.e., if BP_j finds $CC(T) \Rightarrow \bigcirc_n$ it says T committed.) If BP_j says T committed, BP_i also commits T; otherwise T is left pending (updates not installed).
- Transactions for which none of the above rules applies do not commit during this epoch.

After having made the commit decisions, BP_i reexamines its log. Again, it starts with the oldest pending entry (which may occur before $\bigcirc_{n-1}$) and checks the entries in the order in which they appear in the log. If an entry belongs to a transaction for which a commit decision has been reached, the corresponding change is installed in the database and the log entry is discarded. If no commit decision has been made for this transaction, the entry is skipped and will be examined again during the next epoch.

Note that during the first scan of the log (to determine which transactions can commit) the only information from previous epochs that is actually necessary is for which transactions a $P(T)$ entry without a matching $C(T)$ entry has been seen. If this information is maintained across epochs and updated accordingly as *commit* and *prepare* messages are encountered in the log, the first scan can ignore pending entries from previous epochs and start from $\bigcirc_{n-1}$. It is still necessary for the second scan (installing the updates) to examine all pending entries.

5. Why the Epoch Algorithm Works

In this section we show the correctness of the epoch algorithm. In particular, we prove that the first two correctness criteria mentioned in section 3 are satisfied.

Atomicity. To prove atomicity, we use the following two lemmas.

Lemma 1: If $C(T) \Rightarrow \bigcirc_n$ in the log of a processor P_i, then $CC(T) \Rightarrow \bigcirc_n$ in the log of the coordinator P_c of T.
Proof. If $P_i = P_c$, the lemma is trivially satisfied. Now suppose that $P_i \neq P_c$, that $CP(T) \Rightarrow \bigcirc_n$ in the log of P_i and that $\bigcirc_n \Rightarrow CC(T)$ in the log of P_c. The commit message from P_c to P_i includes the current coordinator epoch $n+1$. Upon receipt of this message, P_i will write $\bigcirc_n$ if it has not already done so. Thus, $\bigcirc_n \Rightarrow CP(T)$, a contradiction.

Lemma 2: If $CC(T) \Rightarrow \bigcirc_n$ in the log of the coordinator for T, then $P(T) \Rightarrow \bigcirc_n$ in the logs of the participants.
Proof. Suppose $\bigcirc_n \Rightarrow P(T)$ at some participant. When the coordinator received the acknowledgement (along with the epoch) from that participant, it bumped its epoch (if necessary) and then wrote the $CC(T)$ entry. In either case, $\bigcirc_n \Rightarrow CC(T)$, a contradiction.

Let us now see why atomicity holds. Suppose the changes of a transaction are installed by a backup processor BP_i after the logs for epoch n are received. If $C(T) \Rightarrow \bigcirc_n$ in the log of BP_i and the transaction was coordinated by P_c at the primary, by lemma 1 $CC(T) \Rightarrow \bigcirc_n$ in the log of BP_c. If BP_i does not encounter a $C(T)$ entry before $\bigcirc_n$, it must have committed because the coordinator told it to do so, which implies that in the log of the coordinator $CC(T) \Rightarrow \bigcirc_n$. Thus, in any case, in the coordinator's log $CC(T) \Rightarrow \bigcirc_n$. According to lemma 2, in the logs of *all* participants $P(T) \Rightarrow \bigcirc_n$. The participants for which $CP(T) \Rightarrow \bigcirc_n$ will commit T anyway. The rest of the participants will ask BP_c and will be informed that T can commit. Thus, if the changes of T are installed by one processor, they are installed by all participating processors.

Consistency. We prove the first part of the consistency requirement by showing that if $T_x \rightarrow T_y$ at the primary and T_y is installed at the backup during epoch n, T_x is also installed during the same epoch or an earlier one. Suppose the dependency $T_x \rightarrow T_y$ is induced by conflicting accesses to a data item d at a processor P_d. By property 1 we get $C(T_x, P_d) \Rightarrow P(T_y, P_d)$. Since T_y committed at the backup during epoch n, $P(T_y, P_d) \Rightarrow \bigcirc_n(P_d)$, which implies that $C(T_x, P_d) \Rightarrow \bigcirc_n(P_d)$. Thus, T_x must commit during epoch n or earlier (see lemmas 1, 2). For the second part of consistency: suppose $T_i \rightarrow T_j$ and they both write data item d. As we have shown in the first part, if $T_x \rightarrow T_y$ at the primary, T_x commits at the same epoch as T_y or at an earlier one. If T_x is installed in an earlier epoch, it writes d before T_y does, i.e., $W(T_x, d) \Rightarrow W(T_y, d)$. If they are both installed during the same epoch, the writes are executed in the order in which they appear in the log, which is the order in which they were executed at the primary. Since $T_x \rightarrow T_y$ at the primary, the order must be $W(T_x, d) \Rightarrow W(T_y, d)$, which is exactly what we want.

6. The Double Mark Epoch Algorithm

In the single mark algorithm, *participant−ready* and *commit* messages must include the current epoch number. The overhead, in terms of extra bits transmitted, is probably minimal. However, the commit protocol does have to be modified to incorporate the epoch numbers. This may be a problem if one wishes to add the epoch algorithm to an existing database management system. The double mark algorithm that we now present does not require any such modifications to the primary system. The double mark version works by positioning transactions more carefully in the log (with respect to delimiters).

At the primary there is again a master that periodically writes a $\bigcirc_n$ entry in its log ($E(master)=n$), sets $E(master)$ to $n+1$ and sends an *end−epoch*(n) message to all nodes. Recipients make a $\bigcirc_n$ entry in their logs, stop committing new transactions and send an acknowledgement to the master. Note that commit processing does *not* cease entirely. Transactions can still be processed; only *new* commit decisions by coordinators cannot be made, i.e., after writing the $\bigcirc_n$ entry in its log, a processor cannot write a $CC(T)$ entry for a transaction T for which it is the coordinator (receiving and processing *prepare* and *commit* messages for transactions for which it is not the coordinator is still permissible). Note that except for the master, nodes do not need to remember the current epoch in the double mark version.

When the master collects *all* acknowledgements, it starts a similar second round: it writes a $\square_n$ entry in its log (the counter is not incremented in this round) and sends a *close−epoch*(n) message to all nodes. The recipients make a $\square_n$ entry in their logs, send another acknowledgement to the master and resume normal processing (i.e., new commit decisions can now be made). The master cannot initiate a new epoch termination phase (i.e., write a new $\bigcirc_{n+1}$ entry in its log) until all second round acknowledgements have been received.

The logs (including *circle* and *square* entries) are propagated to the backup site, where they are stored on stable storage. A backup processor does not process the log entries of epoch n until all backup processors have seen $\square_n$ in the logs they receive. Then each computer BP_i examines all of the log entries before $\square_n$ (including entries pending from previous epochs[4]) to decide which transactions can commit after this epoch, according to the following rules:

- If $C(T) \Rightarrow \bigcirc_n$ in the log, a decision to commit T is made.
- If a transaction T does not fall in the above category but $P(T) \Rightarrow \square_n$ in BP_i's log ($P(T)$ could possibly be in some *previous* epoch), the decision whether to commit T depends on whether P_i was the coordinator for T at the primary. (Recall from the model section that the coordinator is included with every $P(T)$ log entry.) If P_i was the coordinator, T does not commit at the backup during this epoch. If some other processor P_j was the coordinator at the primary, a message is sent to BP_j requesting its decision regarding T. (BP_j will reach a decision using the rules we are describing, i.e., if BP_j finds $CC(T) \Rightarrow \bigcirc_n$ it says T committed.) If BP_j says T committed, BP_i also commits T; otherwise T is left pending (updates not installed).
- If none of the above rules applies to a transaction T, the transaction does not commit during this epoch.

After the commit decisions have been made, the log entries up to $\square_n$ are examined and the actions of the committed transactions are installed as in the single mark version of the algorithm.

We now show the correctness of the double mark version. In the correctness proofs we use the following property, which stems directly from the fact that the master receives all acknowledgements for *end−epoch*(n) before sending *close−epoch*(n) messages:

$$\bigcirc_n(P_i) \Rightarrow \square_n(P_j) \ \forall\ i, j, n \quad \text{(Property 2)}$$

Lemma 1: If $C(T) \Rightarrow \bigcirc_n$ in the log of a processor P_i, then $CC(T) \Rightarrow \bigcirc_n$ in the log of the coordinator P_c of T.

Proof. If $P_i = P_c$, the lemma is trivially satisfied. Now suppose that $P_i \neq P_c$. Then,

$$CC(T, P_c) \Rightarrow CP(T, P_i) \quad \text{(by two-phase commit)}$$
$$CP(T, P_i) \Rightarrow \bigcirc_n(P_i) \quad \text{(by hypothesis)}$$
$$\bigcirc_n(P_i) \Rightarrow \square_n(P_c) \quad \text{(by property 2)}$$

By transitivity, we get $CC(T, P_c) \Rightarrow \square_n(P_c)$, and since no commit decisions are allowed for coordinators between the *circle* and *square* entries, we conclude that $CC(T, P_c) \Rightarrow \bigcirc_n(P_c)$.

[4]The comment made in the single mark version about avoiding examination of entries from previous epochs when making commit decisions applies to the double mark version as well.

Lemma 2: If $CC(T) \Rightarrow \bigcirc_n$ in the log of the coordinator for T, then $P(T) \Rightarrow \Box_n$ in the logs of the participants.
Proof. Consider a participant processor P_i. Then,

$P(T, P_i) \Rightarrow CC(T, P_c)$ (by two-phase commit)
$CC(T, P_c) \Rightarrow \bigcirc_n(P_c)$ (by hypothesis)
$\bigcirc_n(P_c) \Rightarrow \Box_n(P_i)$ (by property 2)

By transitivity, we get $P(T, P_i) \Rightarrow \Box_n(P_i)$.

Atomicity. Suppose the changes of a transaction are installed by a backup processor BP_i after the logs for epoch n are received. If $C(T) \Rightarrow \bigcirc_n$ in the log of BP_i and the transaction was coordinated by P_c at the primary, by lemma 1 $CC(T) \Rightarrow \bigcirc_n$ in the log of processor BP_c. If BP_i does not encounter a $C(T)$ entry before $\bigcirc_n$, it must have committed because the coordinator told it to do so, which implies that in the log of the coordinator $CC(T) \Rightarrow \bigcirc_n$. Thus, in any case, in the coordinator's log $CC(T) \Rightarrow \bigcirc_n$. According to lemma 2, in the logs of *all* participants $P(T) \Rightarrow \Box_n$. The participants for which $CP(T) \Rightarrow \bigcirc_n$ will commit T anyway. The rest of the participants will ask BP_c and will be informed that T can commit. Thus, if the changes of T are installed by one processor, they are installed by all participating processors.

Consistency. We prove the first part of the consistency requirement by showing that if $T_x \rightarrow T_y$ at the primary and T_y is installed at the backup during epoch n, T_x is also installed during the same epoch or an earlier one. Suppose that at the primary the coordinators for T_x and T_y were P_x and P_y respectively. Since $T_x \rightarrow T_y$, by property 1 we get:

$CC(T_x, P_x) \Rightarrow CC(T_y, P_y)$

Since T_y committed at the backup, we infer from our processing rules that

$CC(T_y, P_y) \Rightarrow \bigcirc_n(P_y)$
$\bigcirc_n(P_y) \Rightarrow \Box_n(P_x)$ (by property 2)

From the above by transitivity we get $CC(T_x, P_x) \Rightarrow \Box_n(P_x)$ and since no commit decisions are made by coordinators between circle and square entries, we get

$CC(T_x, P_x) \Rightarrow \bigcirc_n(P_x)$

This implies that according to our processing rules transaction T_x must commit during epoch n or earlier. The proof for the second part of consistency is identical to the proof for the single mark version.

7. Evaluation of the Epoch Algorithms

In this section we examine the features of the epoch algorithms and discuss their performance. The algorithms are *scalable*: there is no processing component that must see all transactions. Each computer only processes transactions that access the data it holds. This makes the algorithms appropriate for very large databases.

The protocols have a low overhead and their cost is amortized over an entire epoch. There are three factors that contribute to the overhead: the overhead for the termination of an epoch at the primary, the overhead for ensuring reception of an epoch at all backup sites and the overhead for resolving the fate of transactions for which a $P(T)$ entry without a matching $C(T)$ has been seen. For the first two factors, the number of messages required is proportional to the number of computers at each site. For the third type of overhead, the average number of transactions for which a computer cannot make a decision by itself can be estimated as follows (for the single mark version): the transactions for which a $P(T)$ was written before $\bigcirc_n$ and a $CP(T)$ after $\bigcirc_n$ are those transactions whose $P(T)$ entry falls within a time window t_c before the $\bigcirc_n$ mark, where t_c is the average delay necessary for a *participant − ready* message to reach the coordinator and the *commit* answer to come back. Thus, the expected number of transactions for which information must be obtained from another computer is $w_g \times t_c$, where w_g is the rate at which a computer processes global transactions for which it is not the coordinator. If the entire system processes global transactions at a rate w_e, there are n computers at each site and a global transaction accesses data at m computers on the average, then $w_g = w_e \times (m-1)/n$. Note that the number of messages that must be sent could be less than the number of transactions in doubt, since questions to the same computer can be batched into a single message. Finally, note that all these overheads are paid once per epoch. If an epoch contains a large number of transactions, then the overhead per transaction is minimal.

Let us now compare the two versions of the algorithm. The single mark version requires one less round of messages for writing delimiters at the end of each epoch at the primary. Also, the single mark version does not suspend commits at any point. However, the transaction processing mechanism has to be modified to include the local epoch number in certain messages and to update the epoch accordingly when such a message is received. On the other hand, the double mark algorithm may require fewer modifications to an existing system: the double mark epoch termination protocol can be viewed as the commit phase of a special transaction with null body. The *end − epoch(n)* message corresponds to the message telling participants to prepare and the *close − epoch(n)* message corresponds to the message telling participants to commit. The only system interaction in the double mark protocol is the suspension of coordinator commits. On some systems this may be easy to achieve by simply holding the semaphore for the commit code. (Typically, only a single transaction can commit at a time, and there is a semaphore to control this.) Depending on the system, by holding a commit semaphore we may disable all commits, not just coordinator commits. This may be acceptable if the time between the $\bigcirc_n$ and the $\Box_n$ is short. If this is not acceptable, then a new semaphore can be added.

As we saw in our proofs, the algorithms satisfy atomicity and consistency, but they do not achieve minimum divergence. If a disaster occurs, the last epoch may not have been fully received by the backup computers. The epoch algorithms will not install any of the transactions in the incomplete epochs, even though some of them could be installed. This problem can be addressed by running epochs more frequently (to limit the number of transactions per epoch) or by having another mechanism for dealing with incomplete epochs, e.g., individual transaction commit or a mechanism like the one in [4], [6].

In [4], [6] we have proposed a dependency reconstruction algorithm for maintaining a remote backup. It is interesting to compare these two algorithms. The epoch algorithm induces less overhead, but it does not achieve minimum divergence. The dependency reconstruction algorithm achieves minimum divergence, which implies that the *takeover* time (i.e., the time between the point when a disaster occurs at the primary and the point when the backup starts processing new transactions) is shorter. We have not presented the dependency reconstruction algorithm here, but we believe that in that algorithm it may be easier to have both the primary and the backup run the same software than it is with the epoch algorithm. Having both sites run the same software is desirable, because it may need less effort to maintain it and takeover time is reduced further, since no software reloading is necessary.

Finally, we would like to note that algorithms similar to the epoch algorithm have appeared in the literature for obtaining snapshots [2] and checkpointing databases [14]. The main differences between those approaches and the epoch algorithm are:

- Our algorithm is log based.
- Minimal modifications to an existing system are necessary.
- Minimal overhead is imposed at the primary.
- Our snapshot is not consistent. Enough information is included to allow a consistent snapshot to be extracted from the propagated logs, but some work is still necessary at the backup to clean up $P(T)$ entries with no matching $C(T)$.

8. Another Application of the Epoch Algorithm: Distributed Group Commit

Group commit (for a single node) [3], [7] is a technique that can be used to achieve efficient commit processing of transactions in computer systems with a large main memory, which can hold the entire database (or a significant fraction of it). When the end of a transaction is reached, its log entries are written into a log buffer that holds the tail of the master log. The locks held by the transaction are released, but the log buffer is not flushed immediately onto non-volatile storage (to avoid synchronous I/O). When the log buffer becomes full, it is flushed and the transactions that are contained in this part of the log commit as a group. The updates made by these transactions are installed in the database after the group commit. Care must be taken to ensure that actions of transactions that are members of the same group and depend on each other are installed in a way that preserves these dependencies.

Under the above scheme transactions are permitted to read uncommitted data. However, this presents no problem, since a transaction T can only depend on transactions in the same group or previous groups, which under this processing scheme will be installed before or when T does.

It would be desirable to apply the same technique to distributed systems. However, in a multicomputer environment it is not possible for each individual processor to flush its own log independently of other processors, since that could violate transaction atomicity and therefore compromise database consistency. A simple example illustrates why: suppose transaction T completes at processors P_1 and P_2, processor P_1 flushes its log (and commits T in the database) while P_2 does not. If a failure occurs and the contents of P_2's log buffer are lost (they are in volatile memory), transaction atomicity is violated and the database enters an incorrect state.

To achieve distributed group commit without endangering the consistency of the database we can use the epoch algorithm. One can think of the main memory as being the primary site and of the disks as playing the role of the backup in the discussion of section 4. Transactions run in a main memory database and their logs are written into log buffers, but their changes are not propagated to the disk copy. Distributed transactions still use a two-phase commit protocol to achieve atomicity. $P(T)$ and $C(T)$ entries are made for all transactions that finish processing successfully and their locks are released, but the logs are not flushed.

Periodically, delimiters (e.g., *circles*) are written by all processors in their logs (epoch termination). When the delimiter is written, the log buffer is written on stable storage, but the group commit does not take place until it is confirmed that all processors have saved their log buffers on stable storage. Then, each processor starts to actually install the changes of the transactions in the disk copy of the database, in the same way way the backup processors did in section 4.

The advantages of group commit are manifold. Local transactions that execute only at one node avoid synchronous I/O and release the resources they hold as soon as they finish processing. This, in turn, causes transactions to hold resources for a shorter time, thus decreasing contention and increasing throughput. Furthermore, the cost of log I/O is amortized among many transactions. Distributed transactions still have to pay the cost of the agreement protocol to ensure atomicity. This cost may actually be a little smaller, since the individual *prepare* and *commit* decisions need not be written on stable storage, and thus responses to *prepare* and *commit* messages can be sent immediately. Distributed transactions can also benefit from the amortization of the log I/O cost among several transactions.

9. Conclusions

We have presented an efficient, scalable algorithm for maintaining a remote backup copy of a database. In this section we briefly discuss some issues that we left open in previous sections. First, the size of the epoch counters could be a problem. As time progresses, the epoch numbers become bigger and bigger. How big should the epoch counters be? If only one epoch can be pending at any time, a computer only needs to distinguish between its epoch, the epoch of a computer that is possibly one epoch ahead and the epoch of a computer that is possibly one epoch behind. Thus, a counter with 3 states that cycles through these states should be sufficient. In general, if the epochs of two nodes can differ by at most k, the epoch counter should be able to cycle through $2 \times k + 1$ states.

In the previous sections we made the implicit assumption that the primary and the backup database start from the same initial state. When the system is initialized or after a site has recovered from a disaster, one of the sites will have a valid copy and the other will be null. It is necessary to have an algorithm which will bring the null copy up-to-date, without impairing the performance at the other site. For this purpose, a method like that outlined in [6] and detailed in [4] can be used.

Finally, let us return to the one-to-one correspondence between primary and backup computers mentioned in section 2.1. It is not necessary to have the same number of computers at the two sites. If the primary and the backup have a different number of computers, one can partition the data into logical chunks. As long as the *logical* partitions are identical at the two sites, the epoch algorithms can be applied. One simply needs to keep a log for each chunk and to use the notion of a chunk instead of a computer in the discussion above.

REFERENCES

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

[2] K. M. Chandy and L. Lamport, ''Distributed Snapshots: Determining Global States of Distributed Systems,'' *ACM Transactions on Computer Systems*, Vol. 3, No. 1 (February 1985), pp. 63-75.

[3] D. J. DeWitt, R. H . Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker and D. Wood, ''Implementation Techniques for Main Memory Database Systems,'' *ACM SIGMOD*, Boston, MA, June 1984.

[4] H. Garcia-Molina, N. Halim, R. P. King and C. A. Polyzois, ''Management of a Remote Backup Copy for Disaster Recovery,'' *Princeton University Technical Report CS-TR-198-88*, Princeton, NJ, June 1989.

[5] H. Garcia-Molina and C. A. Polyzois, ''Issues is Disaster Recovery,'' *IEEE Compcon*, San Francisco, CA, February 1990.

[6] H. Garcia-Molina, N. Halim, R. P. King and C. A. Polyzois, ''Disaster Recovery for Transaction Processing Systems,'' to appear in *IEEE 10th ICDCS*, Paris, France, May 1990.

[7] D. Gawlick and D. Kinkade, ''Varieties of Concurrency Control in IMS/VS Fast Path,'' *Data Engineering Bulletin*, Vol. 8, No. 2 (June 1985), pp. 3-10.

[8] J. N. Gray, ''Notes on Database Operating Systems,'' *Operating Systems: An Advanced Course*, R. Bayer et al., editors. Springer Verlag, 1979.

[9] J. N. Gray and A. Reuter, ''Transaction Processing,'' *Course Notes from CS#445 Stanford Spring Term*, 1988.

[10] H. F. Korth and A. Silberschatz, *Database System Concepts.* New York: McGraw-Hill, 1986.

[11] L. Lamport, ''Time, Clocks, and the Ordering of Events in a Distributed System,'' *Communications of the ACM*, Vol. 21, No. 7 (July 1978), pp. 558-565.

[12] J. Lyon, ''Design Considerations in Replicated Database Systems for Disaster Protection,'' *IEEE Compcon*, 1988.

[13] D. Skeen, ''Nonblocking Commit Protocols,'' *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 133-147, Orlando, FL, June 1982.

[14] S. H. Son and A. K. Agrawala, ''Distributed Checkpointing for Globally Consistent States of Databases,'' *IEEE Transactions on Software Engineering*, Vol. 15, No. 10 (October 1989), pp. 1157-1167.

[15] Tandem Computers, *Remote Duplicate Database Facility (RDF) System Management Manual*, March 1987.