Evaluation of Remote Backup Algorithms for Transaction Processing Systems

Christos A. Polyzois

Department of Computer Science
Princeton University
Princeton, NJ 08544-2087
e-mail: cap@princeton.edu

Hector Garcia-Molina

Department of Computer Science
Stanford University
Stanford, CA 94305-2140
e-mail: hector@cs.stanford.edu

ABSTRACT

A remote backup is a copy of a primary database maintained at a geographically separate location and is used to increase data availability. Remote backup systems are typically log-based and can be classified into 2-safe and 1-safe, depending on whether transactions commit at both sites simultaneously or they first commit at the primary and are later propagated to the backup. We have built an experimental database system on which we evaluated the performance of the epoch algorithm, a 1-safe algorithm we have developed, and compared it with the 2-safe approach under various conditions. We also report on the use of multiple log streams to propagate information from the primary to the backup.

1. Introduction

The increasing reliance of enterprises on databases for their operation has created a rising demand for availability in the area of database and transaction processing systems. Local hardware replication techniques (e.g., dual processors, mirrored disks) can mask a significant number of failures and increase data availability. However, these techniques are inadequate for extensive failures (*disasters*), which may be caused by environmental hazards (fire, flood, earthquake, power outage), malicious acts or operator errors. To ensure continuous operation even in the presence of such failures, a backup copy of the primary database is often maintained up-to-date at a remote geographical location and administered separately. When disaster strikes at the primary site, the backup takes over transaction processing. The geographic separation of the two copies reduces the likelihood of the backup also being affected by the disaster.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0246...$1.50

Disaster recovery systems have received significant attention recently. In [12] Jim Gray and Andreas Reuter state that "they offer fertile ground for new algorithms, and also offer the promise of much higher availability" and that "this is the most active area of transaction processing research." Apart from taking over transaction processing in case of disaster, remote backups can find use during planned downtime of the primary system (e.g., for hardware maintenance or a software upgrade). If they have spare capacity, they can also be used to perform useful processing, e.g., for read-only queries [9].

Algorithms for maintaining a remote backup copy fall into two major categories: *1-safe* and *2-safe* [13]. In 2-safe algorithms, the primary and the backup copy are kept in lockstep; all updates to the data are applied to both copies in complete synchrony, typically through the use of an agreement protocol [11, 18, 21]. In 1-safe systems, transactions first commit at the primary site and are later propagated to the backup.

Two-safe is the way in which replicated data is traditionally handled in distributed systems [1, 3] and it offers two main advantages: conceptual simplicity (since the applications programmer is presented with a single logical view of the data) and guaranteed survival of all committed transactions in case of disaster (since changes are applied "simultaneously" to the two copies). However, 2-safe systems also have certain disadvantages [15]. The agreement protocol increases transaction response time by at least one round-trip delay. This may in turn increase resource contention and decrease throughput. Also, if the backup becomes unreachable due to transient network failures, the primary cannot commit transactions (under 2-safe, a transaction cannot commit until its updates are safely received at the backup).

As discussed in [4], 1-safe backups avoid these performance penalties at the cost of potentially losing a few committed transactions in case of disaster. Consider, for example, a transaction T that executed and committed at the primary. If a disaster occurs at the primary before the backup receives the information that will enable it to install T (typically the log entries written by T), transaction T will not survive the disaster. There are applications that can tolerate the loss of a few transactions and prefer to use 1-safe backups to gain in performance [15].

Several algorithms and commercial backup products are available [2, 15, 16, 19, 23]. Some are 2-safe, some 1-safe and some are customizable to allow both options. They are almost invariably log-based, and most of them assume a model with one primary and one backup site. All primary computers merge their logs into a single master log stream, which is propagated to the backup to allow the backup computers to install the same changes that were installed at the primary site. Even when the primary and the backup are distributed systems, the assumption of a single log stream is still made.

Unfortunately, very little has been published in the area of remote backup systems. Most publications come from industrial environments and describe particular products, focusing on their functionality and specifications rather than the underlying principles. There are very few performance evaluation studies and their scope is always limited to a particular system. We have tried to fill this gap by building an experimental transaction processing system on which we implemented various backup algorithms and tested our ideas. We have also extended the system model to allow for several log streams between the primary and backup computers.

In this paper we evaluate the performance of the epoch algorithm (a 1-safe backup algorithm we have developed) and compare it with the performance of the 2-safe scheme. We are not trying to give a definitive statement as to which backup technique is better. The answer depends heavily on the configuration, the workload and the performance requirements of a particular system. Rather, we are trying to gain insight into the various techniques and understand the basic tradeoffs between them. We hope that the results of our study will serve as guidelines for people that have to choose or tune a remote backup algorithm.

The rest of the paper is organized as follows. In Section 2 we present our model of a transaction processing system and we give a brief overview of the 2-safe scheme and the epoch algorithm, which is the 1-safe backup used in our experiments. In Section 3 we describe the transaction processing testbed we built to run our experiments and the implementation aspects of the algorithms. In Section 4 we present and analyze our experimental results and in Section 5 we give our conclusions.

2. Model and Algorithms

In our model there is a collection of computers holding the primary copy of the database and a collection of computers holding the backup copy. The computers holding each copy may actually be distributed across a number of geographical sites and the two collections need not be disjoint: the same computer may act as primary for one part of the database and as backup for another.

In order to maintain the backup copy up-to-date, the backup computers must install the same changes as the primary. To this end, logs of the transaction processing activity at the primary are sent to the backup. These logs are usually maintained at the primary anyway, for local recovery purposes. Existing algorithms and products [19, 23] use a single log stream to propagate information to the backup. Even when each copy resides on multiple computers, the logs from all computers are merged at the primary, transported to the backup, distributed, and then applied.

The single log stream approach relies on a single processor to create the master log, and hence it may not scale up [4]. It also requires network messages to send the logs from each primary computer to the log concentrator. (Similarly, the recipient of the log at the backup must distribute it among backup computers.) If the primary computers are geographically distributed, these messages may have to be sent over a wide-area network. For these reasons, it may be desirable to allow multiple *independent* log streams between the primary and the backup computers. For example, there may be a log stream between each pair of primary and backup peer processors. In this paper we study multiple primary-to-backup log streams. However, in Section 4.5, we return to the master log case and study its performance.

In 2-safe processing (with multiple logs), a transaction T executes at one or more primary computers. The updates at each computer are recorded in the local log and sent to the corresponding backup computer. The updates are not installed either at the primary or at the backup computers until T commits. The commit protocol for T thus involves all primary and all backup computers. Additional details of our implementation of this protocol are given in Section 3.

In 1-safe processing, the logs may be propagated *after* a transaction T commits at the primary. The updates for T may arrive at various backup computers at different times, and special efforts must be made to ensure that T's updates are applied atomically at the backup. Also it is important to avoid dependencies on lost transactions in case of disaster. We illustrate this with an example. There are two primary computers P_1 and P_2, their backup peers B_1 and B_2, and two transactions, T_a, which executed at P_1 and P_2 and T_b which executed at P_2. There is a data dependency $T_a \rightarrow T_b$ at the primary, because the two transactions access data in conflicting modes. Both transactions commit at the primary, but because of a disaster B_1 does not receive its part of T_a. Computer B_2 receives its part of T_a as well as T_b. Transaction T_a cannot commit, because it cannot achieve atomicity. Transaction T_b has been fully received, but it cannot commit either, because it depends on a missing transaction.

In the epoch algorithm [8], each backup computer installs the changes it receives on the log stream from its primary peer. The changes are installed in batches called *epochs*, rather than on a per transaction basis. The epochs are defined by delimiters which the primary computers place in their logs. The backups accumulate the logs they receive, but they do not apply them immediately. Instead, they wait until all backup computers have seen the next delimiter (i.e., until the next epoch has been received by everyone) and then they start applying the epoch (almost) independently of each other.

The placement of delimiters in the logs occurs as follows. Each primary computer keeps a local counter of the current epoch. One primary computer is designated as master and periodically writes a delimiter in its log, increments its epoch counter and sends a message to every other primary computer notifying them that the current epoch has terminated. Upon receipt of such a message a primary computer writes a delimiter in its log and also increments its local counter.

Since epoch delimiters are written in the logs asynchronously by the primary computers, distributed transactions may commit during different epochs at the various primary computers and will be installed in different epochs at the corresponding backup computers. Thus, transaction atomicity may be violated. To prevent this, some extra information is added to the messages of the agreement protocol (typically two-phase commit) that is used to achieve atomicity at the primary. In particular, when a primary computer responds to a prepare message received from the coordinator of a distributed transaction, it includes its current epoch number in the response. The coordinator of a distributed transaction also includes its current epoch number in the commit message sent to participants. When a computer receives a message containing an epoch number, it checks its own epoch number against that included in the message. If the local epoch number is smaller, the recipient increments it to match the number in the message, writes a delimiter in its log and then processes the incoming message. This is similar to bumping logical clocks in [14].

The backup computers wait until all have seen the next delimiter in the log stream. To ensure this, a computer designated as master can collect acknowledgements from backup computers that have seen the next delimiter. When the master receives such acknowledgements from all backups, it sends a message to all backups authorizing them to proceed with the installation of the completed epoch. This synchronization step requires $2 \times n$ messages per epoch, where n is the number of backup computers.

When a backup computer B receives the installation authorization, it starts applying changes made by transactions according to the following rules:

- If for a transaction T a commit message has been seen in the log stream of B before the delimiter, then T commits at B and its updates are installed.
- If a prepare entry (but no commit entry) has been seen in the log stream of B before the delimiter, the decision is based on the following: (a) If the primary computer that sends the log stream was the coordinator of T for the primary copy, then T does not commit during this epoch (non-distributed transactions are assumed to run at the coordinating computer). (b) If the sending computer was not the coordinator for T, then the prepare log entry contains the identity of the coordinator computer for the transaction, the one that coordinated T at the primary. We assume that from this information it is possible to determine the identity of the coordinator's backup peer, call it B'. A message is sent to backup computer B' inquiring about its decision regarding T. Computer B' replies with its decision, which is also adopted by B. (The decision of B' is reached according to the rules mentioned above.)
- In any other case the transaction does not commit at B during this epoch and is deferred for consideration in the next epoch.

3. The Testbed

Our experimental transaction processing system uses eight IBM RT-PC 6150 computers running Unix 4.3BSD. Four computers are the primary set and four are the backup. Each machine holds part of the corresponding database copy (primary or backup) on its local disk and communicates with the machines that hold the rest of the copy as well as with its remote peer over a 10Mb/sec Ethernet. The primary computers receive their workload from a DECstation 5000/125 over the network. Many of the design decisions presented below have been dictated by this environment.

We built a real system instead of a simulator for several reasons. First, a real system is a proof of concept for our algorithms. The confidence in the correctness of the algorithms is higher when the state of the primary and the backup databases can be compared and checked for consistency. Tricky details of the protocols are easy to miss with a simulator. Finally, we made fewer assumptions about the system parameters, since some of them were fixed by our environment.

Real transaction processing environments have more powerful processors and larger and faster I/O systems. Our system is a scaled-down version of a major system. The network delays from primary to backup computers may be

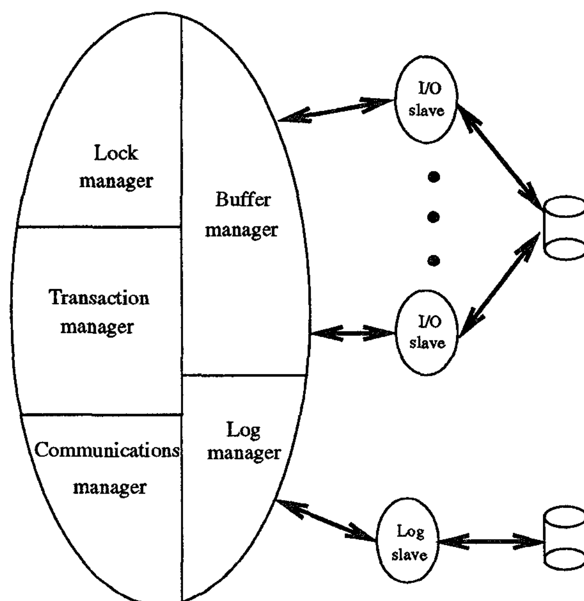


Figure 1. Structure of the testbed.

unrealistically short, since the machines are attached to the same Ethernet. Real systems have longer delays, which we can simulate by introducing a range of artificial delays.

Certain high-level database functions, such as query processing and optimization, were not implemented in our system, since they were not relevant to our study. All access requests have the form read/update/insert/delete record-id and assume that other mechanisms have converted higher-level requests into this form. On the other hand, all low-level DBMS modules are fully implemented, including logging, locking, record and buffer management, etc., since these modules play an important role in backup algorithms.

Figure 1 shows a block diagram of the main parts of the testbed. The lack of shared memory and threads in 4.3BSD forced us to build a monolithic process that implements the transaction logic and does its own internal context switch when a transaction gets suspended on an I/O request. The lock manager, the log manager and the buffer manager are also parts of this process. If these modules ran in separate address spaces, accesses to shared data (e.g., a lock request) would incur the cost of interprocess communication. Finally, the process handles the network communications with other local computers and with the remote peer.

Recovery mechanisms need to determine if modified data is in volatile storage or it has been written to disk. Consequently, we use "raw" UNIX I/O device to control the location of the data and do our own disk management. The problem with this approach is that I/O system calls block and all processing freezes for the duration of the call, so we could not overlap I/O with other processing.

To avoid this problem, we use slave I/O processes. When the central process issues an I/O request, it selects one of the slaves and passes the request (along with the data, if the request is a write) to that slave. The slave performs the actual blocking I/O operation and notifies the central process, passing back the page, if the request was a read. While the slave is blocked on the I/O system call, the central process can continue processing. Using slaves incurs the overhead of the interprocess communication, but this cost is taken into account when reporting CPU loads.

Records are always accessed through a record identifier unique within a table. Hashing maps records to pages and locates them on disk. The operations that can be performed on records are insert, delete, update and read. Insert and update requests also include the image of the record. Since we use small records (30 bytes), updates include the new image for the entire record. In a real system, records may be larger and updates would include only the modified fields. All record accesses are preceded by a request for the appropriate lock (shared or exclusive). Our lock manager also performs lock upgrades. Deadlocks are broken with timeouts. Finally, the system can create tables; these are executed serially.

The log manager uses a slave to perform raw disk I/O without blocking the central process. This is analogous to the scheme used for accessing the database. Since the log is written to a raw device, it is flushed at page boundaries. If a certain interval elapses without enough logging activity to fill the tail page of the log, it is padded to the next page boundary and flushed. Later, when additional log data is appended and the tail page fills, it is rewritten.

We tested the system by writing the log to disk and verifying it. Real systems typically have a separate log disk because the log is written sequentially and sequential disk access is an order of magnitude faster than random access. Our RT's only have one disk, which makes log I/O compete with database I/O. In our experiments, we changed the log device to a dummy device (/dev/null). All of the processing associated with the log (system calls, etc.) is performed as if the data were written to a separate disk.

The buffer manager implements a simple LRU page replacement policy. There may be a limited number of active transactions at any time, and this number is a system parameter. We found that ten transaction slots provide enough multiprogramming to exploit the capacity of our machines fully.

We give a brief description of transaction processing for the 1-safe mechanism at the primary. A transaction is read from the load-generating machine over the network and it is assigned a slot. It performs its actions (setting the appropriate locks), but its changes are made to private copies of the data. When the transaction reaches the end of its execution, it writes log entries for its modifications, a *commit* entry (if the transaction is local) in the log, and waits until that portion of the log has been flushed. Then it writes its changes to the pages in the shared buffer pool, so that they can be seen by other transactions, and releases its locks. The changes may actually be written to disk later when the buffer manager decides to flush a page.

We use a chain model for distributed transactions. Assume that a distributed transaction T is executing at node A. When T completes its processing at A, it writes a *prepare* rather than a *commit* message in the local log, it holds on to its resources (locks, transaction slot) and sends a message to the next node, say B, to start processing on behalf of T. When T finishes at B, it moves to the next node and so on, until it reaches the last node. The last node on the chain also acts as the coordinator. The commit/abort outcome of the transaction is propagated to the participants on the reverse path. The messages that trigger the execution of a distributed transaction at the next node also serve as messages for the agreement protocol: when a message is sent from A to B telling B to start executing T, the message is implicitly conveying information that all nodes in which T has executed so far are prepared to commit T. This information enables the last node on the chain to commit.

In the 2-safe mechanism, all updates of a transaction must be applied synchronously to both the primary and the backup. Thus an agreement protocol is required; we use two-phase commit. Consider a transaction T that executes at only one primary P_i. When P_i finishes executing T, it writes a *prepared* message in its log but does not release

any of the resources held by T. The logs are propagated to B_i, the backup peer of P_i. When the backup receives the logs, it extracts T's actions. The *prepared* entry in the log serves as both a delimiter for T's actions and an indicator that the primary is prepared to commit T.

Transaction T has already run at the primary, so the backup performs only the writeback phase to externalize T's modifications. The backup must also ensure that changes to the same data item by different transactions are applied locally in the same order as at the primary. To do so, the backup sets locks on the data items to be modified by T. The locks that have been obtained by the primary site would suffice to ensure the same sequencing at both sites, provided that these locks would not be released until after both the primary and the backup had installed the changes. This scheme would require additional synchronization between the primary and the backup and would cause the primary to hold locks longer.

After the locks are obtained, the backup writes log entries for T's actions and a "commit T" message in a local log. The backup then acts as the coordinator for T between P_i and itself. The backup waits until the log is flushed to disk and then sends a *commit* message about T's fate to the primary. Upon receipt of the *commit* message, the primary commits T, externalizes its changes, and release its locks. The backup independently does likewise.

In the case of a 2-safe transaction that must access data at more than one primary site, the transaction finishes processing at one site and moves on to the next. As the backups receive the logs of their primary peers, they mirror their peer's processing. If a transaction T moved from primary P_i to P_j, when backup B_i receives the logs for T, it writes a prepare message in its log and notifies B_j, which implies that all backups on T's execution chain up to and including B_i are prepared. Assume that after P_j transaction T accesses data at P_k, and that P_k is the last computer on T's execution chain at the primary. When B_k receives the logs from P_k, all primaries are prepared. When it receives the message from B_j, all backups are also prepared. Thus, all participants are prepared to commit T, so B_k can act as the global coordinator for T. In general, the backup peer of the transaction's last hop acts as the coordinator for all primaries and all backups.

4. Experimental Results

One of the goals of our experiments was to compare the primary load against the backup load in order to estimate the overhead of maintaining a remote backup. A second goal was to compare the behavior of 1-safe and 2-safe backup techniques under various conditions in order to determine the performance cost of 2-safe processing. One expects 2-safe processing to be more expensive, since it provides a higher level of reliability, but *how much* more expensive? Is the performance improvement with 1-safe large enough to merit the potential loss of a few transactions? A third goal was to study the performance impact of the epoch length on the epoch algorithm. Finally, we experiment with single versus multiple log streams.

Each computer in the base configuration of our system holds a 40 MB database; 35 MB hold data and 5 MB are empty and provide overflow pages. The 1-KB pages are half-full with 30-byte records; all records belong to the same table. The buffer cache is 4 MB. Log entries are 16 bytes; inserts and updates are followed by the full image of the record. These base settings are varied in the experiments presented in subsequent sections.

Each primary processes 15,000 transactions. Runs typically take 20-25 minutes. Epoch lengths are 15 seconds. Each transaction accesses exactly 4 records, and the access pattern is uniform. The fraction of read-write transactions is 30%, 70% are read-only. Read-write transactions do at least one write action and the rest of their accesses are half reads and half writes. Write actions can be inserts, deletes, or updates with equal probability. Distributed transactions are a series of local transactions at more than one primary. Distributed transactions access data at a number of computers that is uniformly distributed between 2 and 4, the maximum number of primaries. In the base case, 28% of the transactions executed at each computer were distributed.[1] For this workload and running the epoch algorithm, the I/O and CPU utilization were 98% and 50% respectively. Average response time for local read-write transactions was 700 msec and the throughput of each primary computer was 10.6 transactions per second. There are 4 primaries, so the total throughput is 43 transactions per second.

4.1. I/O Utilization

A backup performs only the fraction of the primary I/O that corresponds to writes. For the base settings, the I/O utilization at the backup is 29%. However, there is an important difference in I/O load between the 2-safe and the epoch algorithm. Assuming input transactions are processed at the primary at a uniform rate, the I/O load at the backup is also uniform for the 2-safe case, e.g., 29%. In contrast, in the epoch algorithm the load comes in bursts: when an epoch is fully received at the backup, the I/O subsystem sees a large volume of I/O, which keeps the I/O subsystem at 100% utilization for 29% of the duration of an epoch. For the rest of the epoch the I/O subsystem is idle.

The periodic nature of I/O in the epoch algorithm may have positive performance implications for systems with more sophisticated I/O subsystems. Long I/O queues can lead to better disk scheduling and increased I/O throughput [20, 22]. In both the 2-safe and the epoch algorithm the spare I/O capacity at the backup can be used for useful processing (e.g., to answer read-only queries). However, the idle periods in the epoch algorithm obviate the need for synchronization between updates and reads, so that the read-only queries can run without locking between

[1] This fraction cannot be set directly: a computer can only adjust the number of distributed transactions it generates; however, it must also execute distributed transactions that have originated elsewhere. Thus, we count the number of distributed transactions during execution and determine their fraction at the end of a run.

epochs when the database is consistent.

4.2. CPU Utilization

Figure 2 shows the CPU utilization[2] at the primary for the epoch algorithm (solid line) and the 2-safe algorithm (dashed line) as a function of the ratio of read-only and read-write transactions. When there are few read-write transactions (10%), the CPU requirements are almost the same for both algorithms. As the fraction of read-write transactions increases, the CPU requirements increase faster for the 2-safe than for the epoch.

Both algorithms must do more logging and initiate more I/O for more write actions, which explains the trend shown in Figure 2. The 2-safe algorithm must also pay for the agreement protocol with the backup, which accounts for its faster increase in CPU demand.

In our system, transactions pay a relatively high CPU overhead when initiating I/O (see Section 3). Nonetheless, our base case may underestimate the CPU requirements of a real system's primary. Real systems perform tasks such as terminal handling, input validation, query processing, actual computation within queries, etc. To cover these costs, we added some CPU processing to each action executed by each transaction and observed the behavior of the system under each algorithm.

The graph in Figure 3 shows the throughput of the epoch and the 2-safe algorithms as the CPU processing per action increases. The horizontal axis is the CPU processing added for each access to a record in milliseconds. Both algorithms suffer as the CPU load increases because the CPU becomes saturated, but the epoch algorithm sustains higher CPU loads before its performance drops. The graph for the I/O utilization in the two algorithms as the CPU

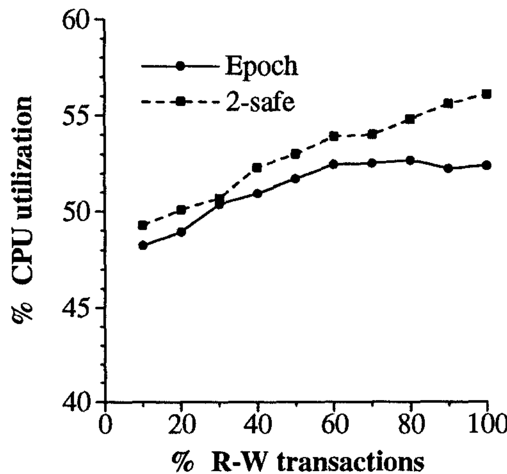


Figure 2. Primary CPU utilization

[2]In our system, the monolithic process polls the slave I/O and log processes even during idle periods. CPU utilization is the actual utilization minus the time spent polling.

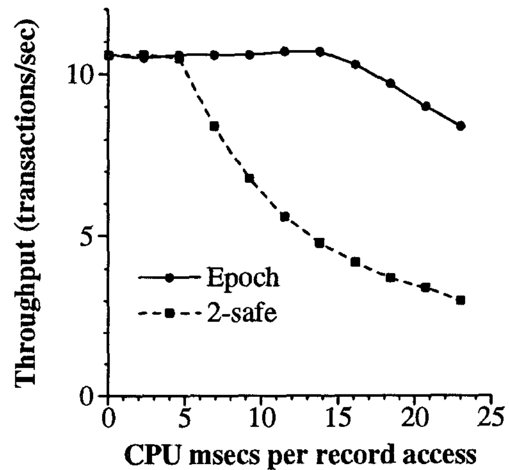


Figure 3. CPU-bound performance

load increases is similar to the graph in Figure 3 because unsaturated I/O is proportional to throughput.

Consider the backup's CPU requirements in the I/O-bound base case. The backup only executes write actions in both algorithms, thus its CPU requirements are lower than those at the primary, as shown in Figure 4, which plots the fraction of the CPU cycles at the primary that is required at the backup for the epoch algorithm against the read-write transaction ratio. The CPU fraction rises with the read-write transaction ratio since the backup has to replicate a rising fraction of the primary's work. The curve for the 2-safe algorithm is similar.

While our base case may underestimate the primary's CPU requirements, it does not underestimate the backup's CPU requirements because a real backup system would not perform any more tasks than our system. Thus, Figure 4 may overestimate the ratio.

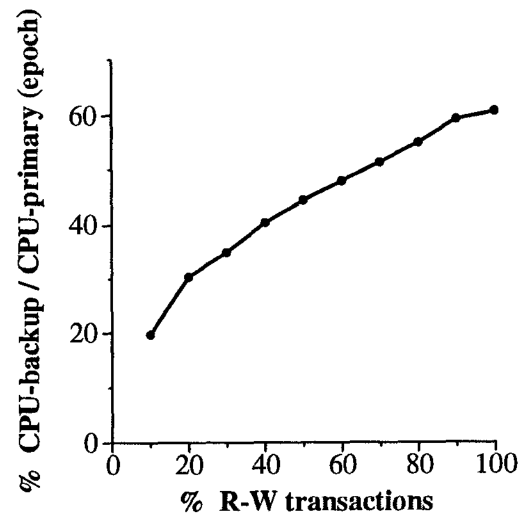


Figure 4. Backup CPU utilization relative to primary

Finally, Figure 5 shows that the CPU requirements of the epoch algorithm at the backup are always less than those of the 2-safe algorithm. Figure 5 plots $\frac{CPU_{2-safe} - CPU_{epoch}}{CPU_{epoch}} \times 100$ versus the read-write transaction ratio, where CPU_z is the CPU usage of algorithm z at the backup.

4.3. Network Utilization

We separate network traffic into two classes: one for messages that propagate the log information to the backup and one for messages for the agreement protocols and other forms of synchronization (e.g., the epoch termination messages). Only the number of messages in the second class varies with the algorithm, so we restrict our attention to those messages.

For a read-write transaction ratio of 30%, we varied the percentage of distributed transactions executed and observed the number of messages generated by primaries and backups. In this experiment, distributed transactions accessed data at two primaries. Figure 6 shows the average number of messages generated by a computer per 10 transactions at the primary and the backup for the epoch algorithm (top) and the 2-safe algorithm (bottom). As the percentage of distributed transactions increases, so do the messages generated by the primaries in both algorithms due to the agreement protocol. The epoch algorithm pays for $N-1$ messages per epoch, where N is the number of primary computers (4). In these experiments, the epoch duration was 15 seconds, so there were 3 messages every 15 seconds at the primary. This overhead is small to negligible; Section 4.6 discusses other aspects of epoch length selection.

At the backup, the two algorithms exhibit different behavior. The 2-safe algorithm requires the same number of messages at the backup as at the primary to run the agreement protocol between backups, plus messages for

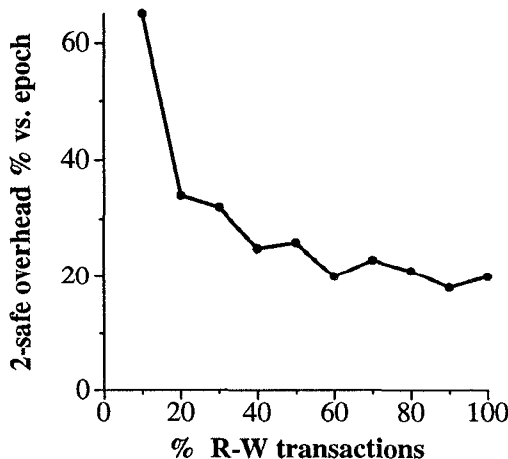
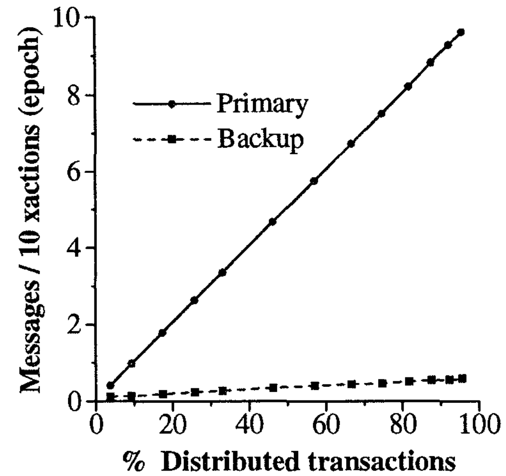


Figure 5. Overhead of 2-safe backup vs. epoch backup



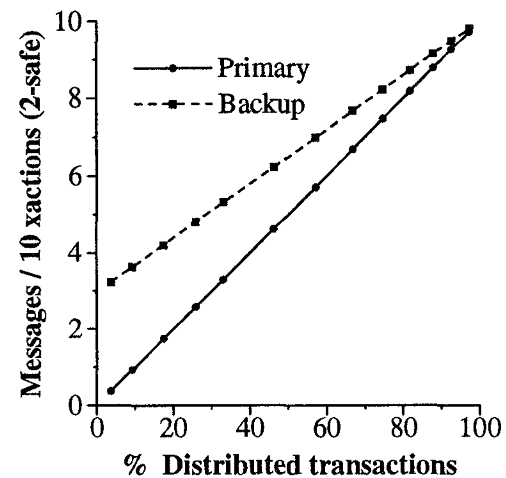


Figure 6. Messages generated by epoch and 2-safe

agreement between primaries and backups (when a backup, acting as coordinator, sends a commit message to its primary peer). The latter messages are necessary even for read-write transactions that access data at only one primary. In contrast, in the epoch algorithm, the backup computers exchange a fixed number of messages to ensure that an epoch has been received by all, and this number is independent of the fraction of distributed transactions. There is also a small number of messages to negotiate the fate of transactions whose prepare and commit messages straddle the epoch marker. The number of these messages increases slowly with the fraction of distributed transactions since the probability of such straddling increases.

Figure 7 illustrates the savings in the number of messages achieved by the epoch algorithm over the 2-safe algorithm; it displays the ratio of the number of messages generated at the backup by the 2-safe over the number of messages generated by the epoch as a function of the distributed transaction percentage. For example, when 20% of the transactions are distributed, the 2-safe backup needs 20 times as many messages as the epoch backup. These savings are due mainly to the batching of transactions in the

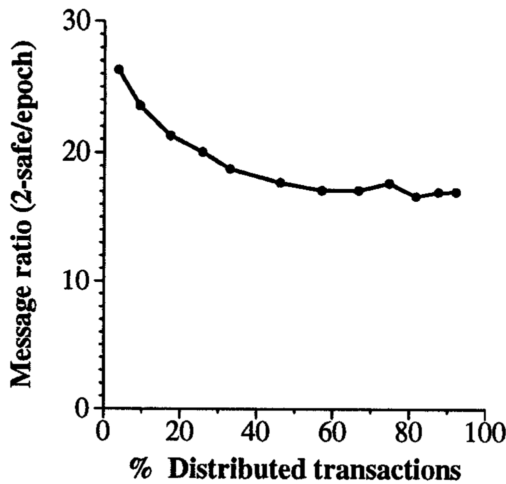


Figure 7. Backup message ratio (2-safe/epoch)

epoch algorithm. In the 2-safe scheme, there is no batching of transactions at the backup because the primary and the backup are coupled. If the 2-safe primary did not commit transactions until an entire batch was received and acknowledged by the backup, response time and throughput would suffer.

The actual cost difference may be higher than shown in Figure 7. We count all messages generated at the backup as having equal cost. In the epoch algorithm, all messages are between backups, whereas in the 2-safe algorithm messages are also sent to the primary. If the backups are physically close to each other, but separated from the primaries, messages from a backup to its primary peer may be more expensive than those to other backups.

4.4. Network Delays

We emulated longer primary-to-backup delays by having senders hold messages before sending them over the Ethernet to their remote peers.

In the epoch algorithm, the primary is decoupled from the backup, so the performance at the primary is almost immune to network delays. The only change is the time by which the backup lags the primary. In the 2-safe algorithm, transaction response time increased by the round-trip delay. Throughput dropped by 7.5% for a round-trip delay of 1 second and 28.9% for a round-trip delay of 2 seconds. For fast networks, the round trip delay will be below 1 second, so that throughput loss may be insignificant.

The small decrease in throughput is due to a pipelining effect: the level of multiprogramming was high enough so that while some transactions were blocked waiting for the acknowledgement from the backup, other transactions could do useful processing and keep the critical components of the system (in our case I/O) busy.

However, there are cases where this pipelining effect is not feasible and the impact of delays is more dramatic. Contention is an example. In the presence of contention, a transaction waiting for a commit acknowledgement can prevent other transactions from running because it holds resources (e.g., a lock) that they need.

Contention is a major problem in database systems and special efforts are made to reduce it. For example, IMS [10] first checks data without locking and obtains locks only if the check succeeds, which decreases lock hold times and alleviates contention. More recent attacks on contention hotspots include Reference [17]. In real systems, hotspots can arise when many transactions need to write the same data. For example, a branch balance may have to be updated for every transaction at a bank.

We introduced various levels of contention in our system by creating workloads with hotspots, i.e., heavily accessed records. Each transaction accessed exactly one hotspot record and three ordinary records. When there were more than one hotspot record in the database, the transaction picked one at random with uniform probability. Contention level was determined by the number of hot records and the fraction of read-write transactions. The three levels of contention tested were low (40% read-write transactions, 3 hot records), medium (50% read-write transactions, 2 hot records) and high (40% read-write transactions, 1 hot record). Even in the high contention case, there is some parallelism: read-only transactions can run in parallel with each other and while one transaction holds a lock on the hotspot, other transactions can access non-hotspot data.

Figure 8 displays the ratio of epoch throughput over 2-safe throughput as a function of network round-trip delays for the three levels of contention. Even for round-trip delays as low as 250 msec, the epoch algorithm achieves throughput 2.5 times as high as the 2-safe algorithm.

4.5. Single vs. Multiple Log Streams

We modified our system to use a central log stream: six computers act as primary transaction processors and a

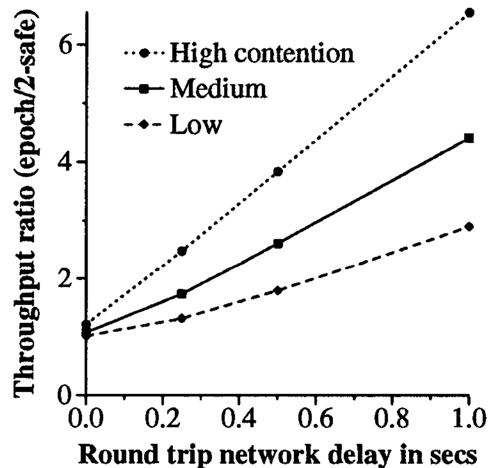


Figure 8. Throughput ratio (2-safe/epoch)

seventh computer acts as the log concentrator. In this experiment, we focused only on the primaries and ignored backups. The concentrator merges the log entries of all primaries and sends them to yet another machine, which simulates the machine that would receive the logs at the backup; in our case, it discards them. A primary writes its log entries in a local log buffer. To flush the log, it sends the log tail to the concentrator. The concentrator writes these entries in its master log and sends an acknowledgement to the creator of the entries after they have been flushed to disk.

Since log writes take longer, the response time of transactions increases with a log concentrator. The throughput remains almost the same because of the pipelining effect mentioned in the previous section. The six primaries use most of the CPU capacity of the concentrator. We did not have more processors, but we expect that if the number of primaries increases, there will be significant delays in the response of the concentrator, transaction response time will increase, and throughput will decrease. This behavior indicates that log concentration does not scale well and that it can be expensive in terms of hardware: merging the logs requires a dedicated processor, which could otherwise be used to process transactions. On the other hand, central logging works well for a small number of processors and it requires only a single log device for all primaries instead of a separate log device per computer.

A log concentrator can also hurt throughput when there is contention. We verified this effect experimentally by running the medium contention workload mentioned in the previous section on our log concentrator testbed. The throughput with multiple log streams was 36% higher than the throughput with the concentrator. Interestingly, contention caused the CPU utilization of the concentrator to drop. Some transactions get blocked waiting for locks to be released and do not generate logs, so the concentrator receives logs at a lower rate and its CPU utilization drops.

4.6. Epoch Length Selection

Selecting an appropriate epoch length is a potential disadvantage of the epoch algorithm. Fortunately, the performance of the algorithm is nearly insensitive to the epoch length. The epoch algorithm requires some synchronization between all primaries when an epoch is terminated and some synchronization between backups when an epoch is received. We varied the epoch length from 2 to 20 seconds, and observed only tiny changes in CPU utilization (1%) at both the primary and the backup, which indicates that the CPU overhead of synchronization in the epoch algorithm is negligible. The I/O utilization was also independent of the epoch length at both the primary and the backup.

The time interval between the moment a primary writes the epoch marker in its log until all backups receive and acknowledge the epoch was 150 msec. For wide-area networks, one should add the difference in round-trip delay between Ethernet and the wide-area network. As expected, this interval did not change with the epoch length.

The number of messages exchanged by backups was the only aspect of the system affected by the epoch length. Since installing an epoch requires a fixed number of messages, longer epochs require fewer messages. Figure 9 shows the ratio of the messages required at the backup over the messages required at the primary for various epoch lengths.

The epoch length also affects the lag between the backup and the primary. For longer epochs, there are more transactions in the last, possibly unfinished, epoch in case of disaster, and these transactions must be processed before the system can accept new transactions. Thus, epoch length affects takeover time. However, it does not affect the window of vulnerability for transactions: a transaction is vulnerable only until it is received at the backup, not until its epoch is installed. If a disaster occurs and an epoch has been partially received at the backup, the transactions in the incomplete epoch can be processed individually. On the average, half an epoch's worth of transactions will have to be processed in this way, and the time required is proportional to the length of an epoch.

Finally, the epoch length may also affect the performance of hybrid 1-2-safe schemes, where most transactions are run as 1-safe and some important transactions are run as 2-safe. As pointed out in [5, 19], a 2-safe transaction may have to wait for all of its 1-safe predecessors to commit at the backup before it can commit. If the epoch algorithm is used to process 1-safe transactions in a hybrid system, the response time of 2-safe transactions depends on the epoch length.

5. Conclusions

We have presented a performance study of disaster recovery mechanisms, including a 2-safe and a 1-safe approach. With a 1-safe mechanism, a limited number of transactions may be lost after a disaster. However, a 1-safe

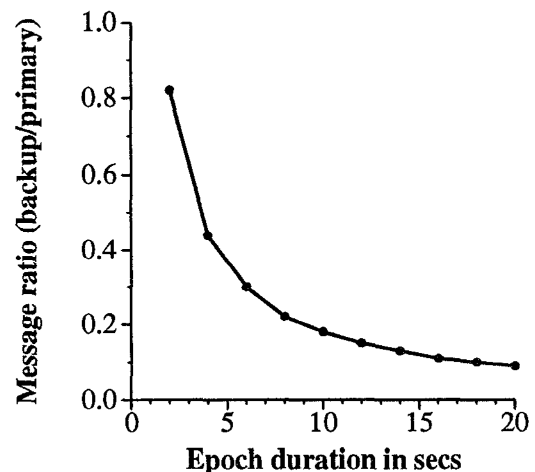


Figure 9. Message ratio (backup/primary) for epoch

mechanism like the epoch algorithm effectively decouples primary and backup processing and significantly reduces CPU and message overhead in many cases. On the other hand, a 2-safe approach may be preferable if transaction loss is intolerable or if networks and processors are fast and lock contention is not serious. Our experiments also illustrated the types of loads present at backup computers and at log concentrators.

Our next step is to use the testbed to study other disaster recovery algorithms (e.g., [6], [7]) and strategies for read-only transaction processing at the backup. We also plan to evaluate database initialization algorithms that are used to recover from a disaster.

REFERENCES

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

[2] Burkes, D., and Treiber, K. Design Approaches for Real Time Recovery. Presentation at the *Third International Workshop on High Performance Transaction Systems,* Pacific Grove, CA, September 1989.

[3] Garcia-Molina, H., and Abbott, R. K. Reliable distributed database management. In *Proceedings of the IEEE, Special Issue on Distributed Database Systems,* pp 601-620, May 1987.

[4] Garcia-Molina, H., and Polyzois, C. A. Issues in Disaster Recovery. In *Proceedings of IEEE Compcon,* San Francisco, CA, February 1990, pp 573-577.

[5] Garcia-Molina, H., and Polyzois, C. A. A Generalized Disaster Recovery Model and Algorithm. In *Proceedings of the Fourth International Workshop on High Performance Transaction Systems,* Asilomar, CA, September 1991.

[6] Garcia-Molina, H., Halim, N., King, R. P., and Polyzois, C. A. Management of a Remote Backup Copy for Disaster Recovery. *ACM Transactions on Database Systems,* Vol. 16, No. 2 (June 1991), pp 338-368.

[7] Garcia-Molina, H., Halim, N., King, R. P., and Polyzois, C. A. Overview of Disaster Recovery for Transaction Processing Systems. In *Proceedings of IEEE 10th ICDCS,* Paris, France, May 1990, pp 286-293.

[8] Garcia-Molina, H., Polyzois, C., A. and Hagmann, R. Two Epoch Algorithms for Disaster Recovery. In *Proceedings of 16th VLDB,* Brisbane, Australia, August 1990, pp 222-230.

[9] Garcia-Molina, H., and Polyzois, C. A. *Processing of Read-Only Queries at a Remote Backup.* Technical Report CS-TR-354-91, Department of Computer Science, Princeton University, December 1991.

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

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

[12] Gray, J. N., and Reuter, A. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1991.

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

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

[15] Lyon, J. Design Considerations in Replicated Database Systems for Disaster Protection. In *Proceedings of IEEE Compcon,* San Francisco, CA, 1988.

[16] Lyon, J. Tandem's Remote Data Facility. In *Proceedings of IEEE Compcon,* San Francisco, CA, February 1990, pp 562-567.

[17] Mohan, C., and Narang, I. Solutions to Hot Spot Problems in a Shared Disks Transaction Environment. In *Proceedings of the Fourth International Workshop on High Performance Transaction Systems,* Asilomar, CA, September 1991.

[18] Mohan, C., and Lindsay, B. Efficient Commit protocols for the Tree of Processes Model of Distributed Transactions. In *Proceedings of 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing,* Montreal, Canada, August 1983.

[19] Mohan, C., Treiber, K., and Obermarck, R. *Algorithms for the Management of Remote Backup Databases for Disaster Recovery.* IBM Research Report RJ 7885R, IBM Almaden Research Center, June 1990.

[20] Seltzer, M., Chen P., and Ousterhout, J. Disk Scheduling Revisited. In *Proceedings Winter 1990 USENIX,* 1990.

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

[22] Staelin, C., and Garcia-Molina, H. *Clustering Active Disk Data to Improve Disk Performance.* Technical Report CS-TR-283-90, Department of Computer Science, Princeton University, September 1990.

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