MANAGEMENT OF A REMOTE BACKUP COPY FOR DISASTER RECOVERY

Richard P. King
Hector Garcia-Molina
Nagui Halim
Christos A. Polyzois

CS-TR-198-88

December 1988

Revised June 1989

Management of a Remote Backup Copy for Disaster Recovery

Richard P. King
Nagui Halim

IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Hector Garcia-Molina
Christos A. Polyzois

Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

A remote backup database system tracks the state of a primary system, taking over transaction processing when disaster hits the primary site. The primary and backup sites are physically isolated so that failures at one site are unlikely to propagate to the other. For correctness, the execution schedule at the backup must be equivalent to that at the primary. When the primary and backup sites contain a single processor, it is easy to achieve this property. However, this is harder to do when each site contains multiple processors and sites are connected via multiple communication lines. We present an efficient transaction processing mechanism for multiprocessor systems that guarantees this and other important properties. We also present a database initialization algorithm that copies the database to a backup site while transactions are being processed.

1. Introduction

In critical database applications, the halting of the computer system in case of failure is considered unacceptable. Instead, it is desirable to keep an up-to-date backup copy at a remote site, so that the backup site can take over transaction processing until the primary site recovers. Such a remote backup database (or *hot standby),* normally of the same capacity as the primary database, should be able to take over processing immediately. Furthermore, when the primary site recovers, the backup should provide it with a valid version of the database that will reflect the changes made by transactions processed while the primary was not operational; this will enable the primary site to resume normal processing. Finally, the overhead at the primary site for the maintenance of the remote backup copy should be kept low.

A remote backup copy has advantages over other forms of fault tolerance. To illustrate, consider a common form of local replication: mirrored disks. It can happen that a repairman trying to fix one of a pair of mirrored disks accidentally damages the good disk, which is physically located next to its faulty mirror image [5]. Thus, because failures tend to propagate, local replication may sometimes be inadequate. The remote backup copy technique decouples the systems physically, so that failures are isolated and the overall system is more reliable. We have mentioned only hardware failures so far; physical isolation can also contain failures caused by operator errors or software bugs. For example, an operator may mistakenly destroy the database by reformatting the disks that hold it. This has actually been reported [5]. However, it will be much harder for the operator to destroy the database stored remotely and under the control of a separate operator. Similarly, software bugs triggered by particular timing events at the primary site will probably not occur at the backup. The backup will have bugs of its own, but these are likely to occur at different times. Thus, remote backup copies provide a relatively high degree of failure isolation and data availability, and are actually used in practice [4].

Backup systems can track the primary copy with varying degrees of consistency. An *order-preserving* backup ensures that transactions are executed in the same logical order they were run at the primary. This is the only approach we will consider here. Non order-preserving backups are sometimes used in practice [2], but may lead to inconsistencies between the primary and the backup.

Along another dimension, backup systems can run *1-safe* or *2-safe* transactions [7], [10]. 2-safe transactions are atomic: either their updates are reflected at both the primary and the backup, or they are not executed at all. A conventional two-phase commit protocol can be used to provide 2-safety [6], [12]. One major drawback of 2-safe protocols is that they increase transaction response time by at least one primary-backup round trip delay. According to [10], this may exceed one second in practice. These delays force transactions to hold locks longer, increasing contention and decreasing throughput.

To avoid these delays, many applications use 1-safety only: transactions first commit at the primary and then are propagated to the backup. A disaster can cause some committed transactions to be lost. These losses only occur when a disaster hits and are ''economically acceptable'' in applications with ''very high volumes of transactions with stringent response time requirements. Typical applications include ATM networks and airline reservations systems [10].'' The backup algorithm we will present in this paper is intended for very high performance applications and only provides 1-safety.

In this paper we make three main contributions:

[1] We formally define the concept of 1-safe transactions and its implications. Commercial systems claim they provide 1-safe transactions, but they never state precisely what it means to "lose some transactions". For instance, it is important to specify that if a transaction is lost, any transactions that depend on it cannot be processed at the backup.

[2] We present a fully decentralized and scalable backup management algorithm that does not rely on a single control process. All commercial products concentrate the logs of every transaction into a single control process and master log. Such a process becomes a bottleneck in larger systems. Furthermore, their solutions are not amenable to parallelization: if the control process is split into a set of processes, correctness can no longer be guaranteed.

[3] We present an efficient database initialization algorithm. It does not rely on first making a fuzzy dump and then bringing it up-to-date with the log. Instead, the fuzzy dump and the log playback occur concurrently, making the initialization simpler and faster.

2. Review of existing systems

Systems for maintaining a backup copy are commercially available. For example, Tandem provides a Remote Duplicate Database Facility, RDF [14] and IBM markets an Extended Recovery Facility, XRF [9]. These packages provide a set of utilities for dumping databases, monitoring them, and propagating modifications to a backup database. It is not our intention here to describe the full packages; we are only interested in the algorithms used for maintaining and initializing the backup database.

We discuss RDF briefly, which is typical of commercial systems. At the primary site, undo/redo log entries for every transaction are written on a master log. (If several processes or processors are executing transactions, they all communicate with a logging process that manages the master log.) As this log is written, a copy is sent to a *control process* at the backup site.

When a transaction commits at the primary site, it is assigned a *ticket* (a sequence number) that represents its commit order relative to other transactions. This is the order in which transactions must install their updates at the backup. Note that the notion of a ticket may not be explicitly implemented; it may be implicit in the position of a commit message in the log. The log entries are received and installed at the backup by the control process in the same order they were generated. When a commit record for a transaction T_1 is encountered, all transactions with earlier tickets have already been safely received at the backup, so it is safe to commit T_1.

The actual writes are performed by a collection of *writing processes*. Each writing process is assigned a part of the database (e.g., a disk volume). The control process distributes the writes to the appropriate writing processes, which install the updates in the order they receive them.

To initialize the backup database, a fuzzy dump is made at the primary, usually onto tape. While the dump is in progress, the master log is sent to the backup, where it is accumulated on disk. The tape is carried over to the backup (or transmitted if it is small) and loaded onto the database. Finally, the saved log is played back against the database, bringing it up to date. The log playback is done through the control process and ticketing described above.

The ticket assignment process plays a critical role, since every transaction has to go through it. Having this central bottleneck is clearly undesirable. It would be much better to have multiple master logs created and received by multiple control processes running on multiple machines at each site. Unfortunately, it is now much harder to know when a transaction can be committed at the backup and what its ticket is. But before we can illustrate this difficulty and our solution, we must step back and define more formally an architectural framework and correctness criteria.

3. Architecture

To make our discussion concrete we define an architecture, shown in Figure 1. We will try to place as few restrictions as possible on this model, in order to make it widely applicable. By *site* we mean all of the computer equipment at one of the locations where the database resides. Each site (primary and backup) consists of a number of *stores* and a number of *hosts*. The data actually resides in the stores, which are assumed to have reliable, non-volatile storage. The stores also have enough processing power to perform basic database operations (such as accessing and modifying records), to keep logs, etc. The hosts are responsible for processing the transactions; they communicate with the users, the stores at the local site and the hosts at the remote site.

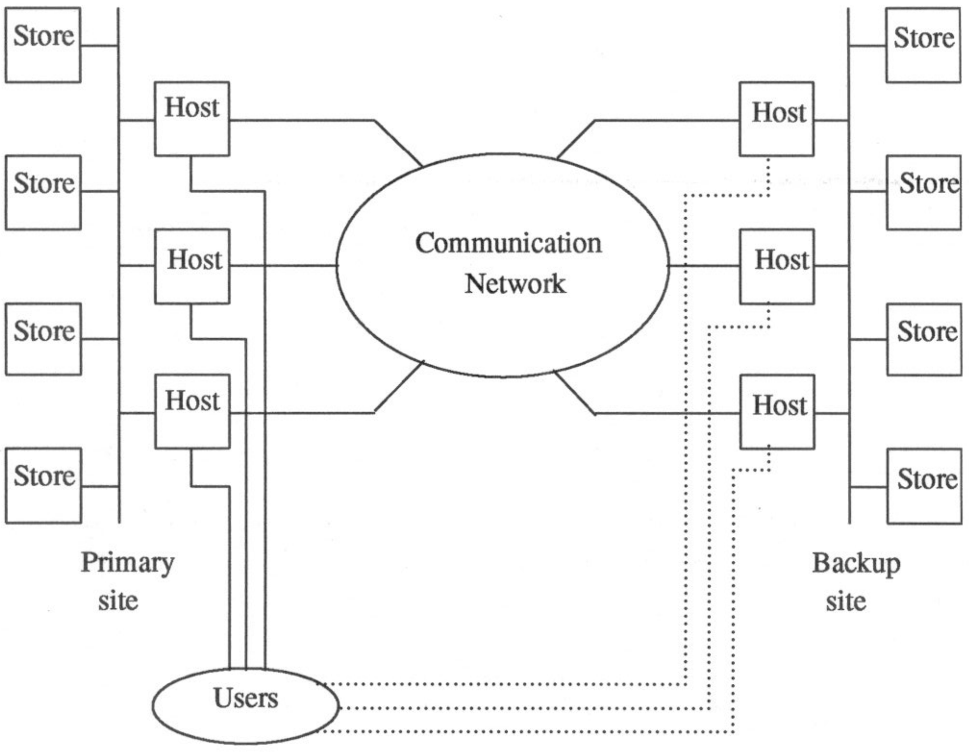

Figure 1. System Architecture

The stores and hosts need not necessarily be disjoint processors. The same processor could perform both tasks, by dividing its CPU cycles between them. Stores and hosts might actually be implemented as two layers of software in the same system. For example, Bernstein et al. [1] define a database system architecture with a *transaction manager* (corresponding to our notion of host) and a *data manager* (corresponding to our notion of store). Similarly, the RSS store manager of system R could implement our stores and the RDS system our hosts.

All of the stores and hosts at one site can communicate with each other through shared memory or through a local network or bus. The method we will present applies to shared memory architectures (e.g., DEC Firefly, Encore Multi-Max) as well as more loosely coupled systems (e.g., a VaxCluster, a Tandem Dynabus or a set of Camelot workstations connected via an Ethernet).

Running between the two sites are several communication lines, which let hosts at the primary site send copies of operations being performed to the backup hosts. Control messages are also exchanged over these lines. The existence of enough bandwidth to propagate the changes is assumed; however, communication delay is not a critical issue here, since we are assuming 1-safe transactions. We remind the reader that the capacity of the stores at the remote site must be at least equal to the one at the primary site.

The database model we will use is a simple version of the relational model, but it is realistic enough to allow us to study the major issues. The database contains a set of *tables,* and each table contains a set of *records.* The tables have unique *names* and the records have unique *record ids.* Requests can be made to create or delete tables and to insert, select, update or delete records. Each of these requests must provide the appropriate parameters. For example, in order to update a record, one must provide the id of that record along with the new version of it. The store will, upon request, create and maintain indices on tables, with any subset of the fields forming the key of the index. Basic transaction processing functions, such as locking, abort and commit operations, are also supported by the store. Such requests are always associated with a transaction identifier, which is established with a begin-transaction request during the initialization phase of a transaction.

The tables are partitioned among the stores. This could be done by hashing names, by table look up or some other method; the details are not important, but as we will show in section 10, the partition must be logically identical at the two sites.

Let us take a brief look at how normal processing would proceed at the primary site without the existence of the backup. A host gets a transaction from a user and assigns it a transaction id (host id followed by sequence number). Before a transaction issues the first request to a particular store, it must send a begin-transaction request to this store. Then the transaction is executed, by issuing the appropriate requests to the store(s) where the corresponding data reside; the requests contain the transaction id and the necessary parameters. When all of the requests have been executed, the host initiates a two-phase commit protocol that ensures that the transaction is either committed or aborted at all stores. The stores, on the other hand, execute all of the database operations that are issued to them, produce the necessary logs, set the appropriate locks, etc. [6]. We are assuming strict two-phase locking is used for concurrency control. Note also that no global transaction sequence numbers indicating the commit order are generated.

(Generating them would create a bottleneck.)

We assume a fail-stop model [13] for host and store processors. We also assume that each host to host connection offers a datagram service [15]. Messages are delivered correctly, but not necessarily in order.

As failures occur at the primary, the system tries to recover and reconfigure. (The details for this are given in Section 9.) However, multiple 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 a 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. For simplicity we assume that when a primary disaster occurs the hardware at the backup site is fully operational. (Certain failures at the backup could be tolerated during a primary disaster, but this will not be discussed here.) A *backup disaster* is similarly declared when failures impair the backup site. We assume that the primary site is operational during a backup disaster.

A site is always in one of three modes: primary, backup or recovering. At most one of the sites can be in the primary mode at any time. Under normal operation, processing of the transactions takes place at the site operating in primary mode and sufficient information is sent to the site operating in backup mode to allow it to install the changes made at the primary site. When a primary disaster is declared, the site previously operating in backup mode starts operating in primary mode and all of the transactions are directed to it. When the failed site comes up, it enters a special recovering mode, which will allow it to get a consistent, up-to-date copy of the database and (perhaps later) resume normal processing. The same recovering mode is used for creating the backup copy at system initialization. Note that the primary and backup roles are interchangeable between the two sites.

4. Evaluation Criteria

In this section we describe the requirements for 1-safe transactions more precisely. The transaction processing mechanism at the primary ensures that the execution schedule *PS* of a set of transactions *T* is equivalent to some serial schedule. Schedule *PS* induces a set of dependencies on the transactions in *T* .

We say $T_a \rightarrow T_b$ (in *PS*) if both transactions access a common data item and at least one of them writes it [8]. Dependencies can be classified into write-write (W-W), write-read (W-R) and read-write (R-W) depending on the actions that generate them.

The backup site will execute a subset of the actions in *PS*. Let this schedule be *BS*. Read actions are not performed at the backup since they do not alter the database. The write actions that are executed simply install in the database the value that their counterpart installed at the primary. Because of failures, not all write actions of *PS* may appear in *BS*.

Requirement 1: Atomicity. If one action of a transaction T_x appears in *BS*, then all write actions of T_x appearing in *PS* must appear in *BS*. This disallows partial propagation of a transaction. ○

Let R be the set of transactions whose writes are in *BS*, $R \subseteq T$.

Requirement 2: Mutual Consistency. Assume T_x and T_y are in R. Then, if $T_x \rightarrow T_y$ in *BS*, it must be the case that $T_x \rightarrow T_y$ in *PS*. This guarantees that the backup schedule is "equivalent" to the primary, at least as far as the propagated *write* actions are involved. (Since no read actions take place at the backup, this requirement does not apply to R-W or W-R dependencies.) ○

Let M be the set of transactions that were not fully propagated to the backup before a failure and hence were not installed. In addition to these transactions, there may be other transactions that we do not want to install at the backup. For example, suppose that when T_x and T_y execute at the primary, T_x writes a value read by T_y. If T_x is not received at the backup (i.e., $T_x \in M$), we do not want to install T_y either, even if it was properly received. If we did, the database would be inconsistent.

To illustrate this, say that T_x is the transaction that sells a ticket to an airline customer. It inserts a record giving the customer's name, date, flight involved, payment information, and so on. Transaction T_y checks-in the passenger at the airport, issuing a seat assignment. The updates produced by T_y cannot be installed at the backup without those of T_x: there would be no passenger record to update. Thus, we have the following requirement:

Requirement 3: Local Consistency. No transaction in R should depend on a transaction in M. That is, suppose there is a transaction $T_a \in M$ and there is a sequence of W-W and W-R dependencies (*not* R-W) in *PS*:

$$T_a \rightarrow T_b \rightarrow T_c \rightarrow \cdots \rightarrow T_n.$$

Then none of T_a, T_b, ..., T_n is allowed to be in R. If C is the set of transactions that depend in this way on M transactions, then $R \cap (M \cup C)$ should be empty. ○

Note that R-W dependencies do not cause violations of the local consistency constraint. If $T_a \rightarrow T_b$ and the only dependencies between these two transactions are R-W, then the values installed by T_a cannot possibly affect the values installed by T_b. Thus, one can install at the backup updates made by T_b and have a consistent database, even if T_a does not reach the backup.

Requirement 4: Minimum divergence. The backup copy should be as close to the primary as possible, i.e., the backup site should commit as many as possible of the transactions it is given, as long as none of the above correctness constraints is violated. In other words, if a received (therefore not belonging to M) transaction does not depend on any transaction in M (i.e., does not belong to C), then it has to belong to R, i.e., $T = R \cup M \cup C$.

Requirement 5: Possibility for parallelism. Although it is rather difficult to give a measure for parallelism, one can see that an acceptable solution to the problem should exploit the potential for parallelism. For example, suppose that transactions at the backup site are processed sequentially. If the primary site allows parallel execution of transactions, it will have a higher throughput than the backup. The backup will inevitably be unable to keep pace with the primary, and the backup copy will become out of date.

In closing this section we make four observations. First, we have implicitly assumed that the primary and backup schedules PS and BS run on the same initial database state. In section 8 we present a procedure for initializing the backup database so that this property holds.

Our second observation is that read only transactions do not modify the state of the database and therefore need not be propagated to the backup site.

The third observation is that most replicated data mechanisms described in the literature (e.g., [1], [3]) would not allow what we call missing transactions. That is, they would have the property $M = \varnothing$ (2-safe transactions). As discussed in the introduction, they would use a two-phase commit protocol to achieve this property. With the weaker constraints we have defined in this section, it is possible to have a more efficient algorithm. In essence, transactions commit at the primary (using a local two-phase commit that involves primary hosts and stores), release their locks, and only then are propagated to the remote backup. At the backup there will be a second local commit protocol to ensure that actions are properly installed there. The details will be given in the rest of the paper.

Our last observation deals with the semantics of missing transactions. In particular, is it "valid" to lose transactions that already committed at the primary? Is it "reasonable" to throw away transactions that were propagated but violate the local consistency requirement? As discussed in the introduction, real applications do allow missing transactions and can cope with the consequences. A simple example may illustrate what these consequences are.

Consider a banking application where a transaction T_x deposits 1,000 dollars into an account that initially has no funds, and a transaction T_y that withdraws 400 dollars. Both of these transactions run at the primary just before a disaster. Transaction T_x is lost, but T_y arrives at the backup. (This can happen if the log records for T_x are propagated via a different communication line than that for T_y.) To satisfy the local consistency constraint, T_y is not installed at the backup. The database will be consistent at the backup but not consistent with the real world. The inconsistencies with the real world will be detected and corrected manually. For example, when the customer checks the balance of the account (at the backup) and sees zero instead of 600, he will go to the bank with his deposit receipt and ask for a correction. The bank, knowing that a disaster occurred, will be willing to make amends for missing transactions. The bank might lose some money (especially if withdrawal transactions are lost), but this cost can be much smaller than the cost of providing 2-safe transactions. Furthermore, not all transactions have to be 1-safe. Operations that involve large sums of money can be performed with 2-safe protocols. (In this paper we do not discuss the algorithms for such hybrid 1-2-safe processing. The extensions to our algorithm for this are straightforward.)

Finally, there is the issue of whether T_y (the withdrawal transaction that depends on the lost T_x) should be discarded even if it arrived at the backup. Installing T_y makes the database inconsistent (e.g., there are not enough funds in the account to cover this withdrawal). On the other hand, discarding T_y creates an inconsistency with the real world. We believe it is more important to maintain database consistency. Without it, it would be very hard to continue processing new transactions. Furthermore, T_y can be saved for special manual processing. A bank employee can look at the T_y record and decide on the best compensating transaction to run. Since disasters occur rarely, the price to pay for this special processing of the relatively few transactions that are discarded is probably acceptable.

5. Generating the Logs

The basic idea in our solution is to reproduce the actions of each primary store at a corresponding backup store. For this, a log of the executed actions must be forwarded to the backup store. The log could be at the action or the transaction level. We have chosen the former. The log could have undo/redo information, or just redo. Our method does *not* send undo information; this reduces communication traffic. Each host forwards the log entries for its own transactions. We made this set of choices for concreteness and for compatibility with our architecture; other alternatives may also be possible.

Given these design choices, we now describe in more detail how the primary site generates the logs. The stores at the primary site keep track in a redo log of the actions they are performing on behalf of a transaction. As will be explained later, even read operations have to be recorded. Each redo log entry should contain enough information to enable the corresponding stores at the backup site to repeat the same operations on their copy. This information should contain the following data:

S_i:	the store performing the operation
H_j:	the host controlling the transaction
T_x:	the transaction id
act:	action descriptor (e.g., update)
tbl:	the name of the table involved
key:	the record id of the record being updated
val:	the after image of the record being updated

Note that not all entries contain all of the above data. For example, the log entry for a delete need not contain an after image, and for create-table no key nor value is necessary.

When a host receives a transaction, it issues a begin-transaction request to the stores (at the primary site) that will participate in the processing of the transaction. Processing proceeds as usual, with local locks being used for concurrency control and with the stores constructing the redo log. Note that in many cases stores keep a log anyway (for local crash recovery), so the log for the remote backup represents no extra overhead. However, other alternatives are also possible. For example, a host could construct its own redo log or get it from the stores as actions are acknowledged.

When a transaction completes, the host initiates a (local) two-phase commit protocol. Upon receipt of a *commit* message, a store produces a local ticket number by atomically incrementing a counter and getting its new value. Intuitively, if a transaction gets a ticket t at store S_i, the transaction ''saw'' state $t-1$ of the store. If the transaction has only read (not written) data at that store, the ticket of the transaction becomes the value of the counter plus one, but the counter is *not* incremented (since the state of the store did not change). For example, if the counter has the value 25, the committing transaction gets ticket 26. If the transaction wrote at this store, the counter becomes 26; otherwise, it stays at 25.

The store creates a commit entry in its redo log, with the following data:

S_i:	identifies the store involved
H_j:	identifies the host controlling this transaction
T_x:	transaction id
act:	action descriptor, in this case ''commit''
ticket:	the local ticket number

The stores return the portion of the redo log for T_x (including the ticket number) to the host, as an acknowledgement to the commit message they received and release locks held. The host controlling T_x makes sure that T_x has committed at all of the primary stores; it then batches together all of the redo log entries for T_x it received from the stores, and sends them across the network to a remote host. The choice of the remote host may be static (i.e., host i at the primary site always sends to host j at the backup site) or it may be determined dynamically, perhaps on the basis of load information.

Each transaction id may be associated with several ticket numbers, one for each store in which actions of this transaction were executed. As will be explained in the next section, these ticket numbers are used to ensure that the changes are applied to the backup copy in the same order as in the primary. After the processing of the transaction at the remote site has finished, the host that controlled the transaction at the primary site is informed, which in turn informs the stores that they may safely erase the relevant portion of the redo log from their storage. Note that so far transaction processing is similar to what it would be without a backup copy. The only difference is the *local* generation of a ticket number and the forwarding of log records to a remote host. We believe these activities do not impose a significant overhead on normal transaction processing, which is consistent with our goals.

There are two alternatives with respect to when the user submitting a transaction gets a response. If the response is sent to the user after the transaction has committed at the primary site, then, in case of disaster, the transaction may be lost if it does not commit at the backup. If the user gets a response after the transaction has committed at the backup, then it is guaranteed that the effects of a transaction will not be lost in case of disaster. Note that transactions are 1-safe in both cases; only the user's information about the fate of a transaction is different.

6. Installing Actions at the Backup — Why the Simple Approach Does Not Work

The next problem we address is installing the actions at the backup site. At this point, it is tempting to propose a very simple solution: As each backup store receives the redo log entries, it simply installs them in ticket order, without regard to what other backup stores may be doing. This simple solution could be viewed as the generalization of the approach used by commercial systems for the case where there is no master log.

In this section we will show that there are problems with this approach. But before doing this, we point out that at least the simple approach does guarantee mutual consistency (Requirement 2). To see this, suppose we have two transactions T_x and T_y, such that $T_x \rightarrow T_y$ at the backup site. Let z be a data item that caused this dependency. If actions are installed at the backup in local ticket order, the ticket number of T_x is smaller than that of T_y at the particular store. This implies that at the primary T_x got its ticket before T_y. When T_x got its ticket, it held a lock on z, which was not released until T_x committed. The lock was incompatible with the lock on z requested by T_y. Thus, $T_x \rightarrow T_y$ at the primary.

One problem with the simple approach is that it does not ensure that transactions are installed atomically at the backup (Requirement 1). Thus, in addition to executing writes in ticket order at each store, it is necessary to execute a *local* two-phase commit protocol among the stores that participated in a transaction.

Even with a two-phase commit protocol, local consistency (Requirement 3) may be violated. To illustrate this, we need an example with three transactions and three stores (Figure 2). At the primary site, the following events occur: T_a writes data at store S_1 getting ticket number 8 (S_1). (We follow the ticket number by the store id to prevent confusion.) Transaction T_b then reads this data, getting ticket 9 (S_1) and writes at store S_2, receiving ticket 3 (S_2). Later on, T_c reads this data and writes at both S_2 (ticket 4) and S_3 (ticket 7).

A disaster hits and T_a is lost. Transactions T_b and T_c do make it to the backup. At backup stores BS_2 and BS_3 all writes are received so there are no gaps in the sequence numbers. Thus, it would appear that a transaction like T_c could commit. All of the data it wrote is available at BS_2 and BS_3. Furthermore, all data written at those stores before T_c is also available. Unfortunately, since T_a is lost, at BS_1 there is a gap: sequence number 8 (S_1) is missing. Thus, T_b with sequence number 9 (S_1) must be aborted. If this happens, then there will be a gap at BS_2: sequence number 3 (for T_b) is not really available. Hence, writes with higher sequence numbers cannot be installed, so T_c must also be aborted.

The difficulty caused by this type of cascading aborts should be apparent by now: before a transaction can be installed at the backup, we must make sure that the transactions it depends on have committed at the backup. This involves synchronization with other stores. Thus, updates cannot be installed blindly by each backup store, even if they are in correct sequence[1].

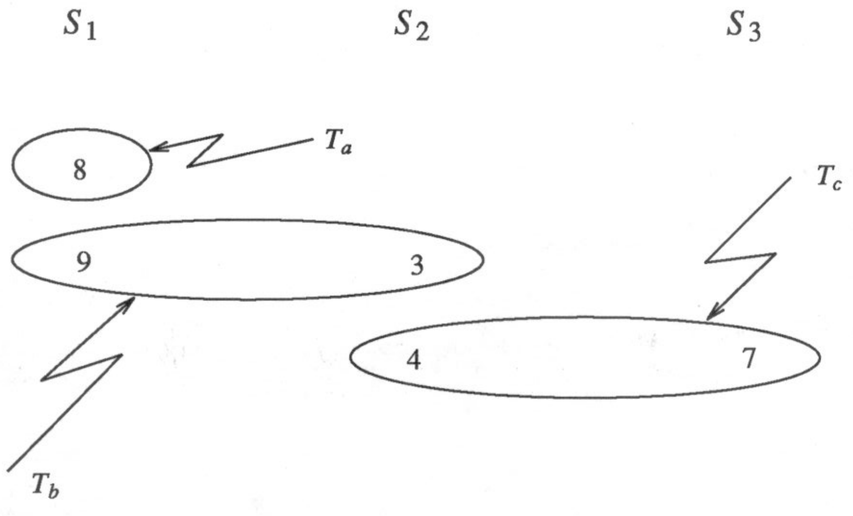

Figure 2. Cascading Aborts

[1]Even if we send the redo actions batched by store rather than by transaction, the problem is not fixed. In this case, all actions with smaller tickets are always received, but the streams of redo logs may not be synchronized. Using the above example, the actions of T_a may be received at BS_1 but not at some other store, say BS_5. Transaction T_a is not able to commit, so it is as if its actions at BS_1 were missing. The cascading aborts are again the same.

As we have seen, installing actions in ticket order at each store guarantees mutual consistency. Unfortunately, there is potentially a major drawback in terms of efficiency. Let us illustrate with an example. Suppose that T_1 and T_2 access disjoint data sets at some primary store and $ticket(T_1) = ticket(T_2) - 1$. At the backup, the writes for T_2 cannot be installed until those for T_1 are installed (this is what ticket order means). Thus, T_2 must wait until T_1 *commits* (which as we have seen involves waiting for other stores to tell us what they have done), and then wait further until the writes of T_1 are actually executed. This is clearly inefficient, especially if stores have a capacity for executing writes in parallel e.g., have multiple disks. Even with one disk, efficient disk scheduling of T_1's and T_2's writes is not possible. Remember that T_1 and T_2 do not depend on each other, so their commits are independent and their writes could proceed in parallel.

To avoid these delays, it is necessary for each store to determine if transactions depend on each other or not. If T_1 and T_2 write a common item, then clearly they depend and T_2 must wait for T_1 to commit and write. However, if T_1 and T_2 write disjoint sets there may still be a dependency! These dependencies can only be detected if read sets of transactions are also propagated to the backup. To illustrate, suppose that at a primary store T_1 wrote item y and that T_2 read y and wrote z. There is a dependency $T_1 \rightarrow T_2$, so T_2 cannot commit at the backup unless T_1 does (local consistency, Requirement 3). The corresponding backup store will not be able to detect the dependency from the write sets ($\{y\}$ and $\{z\}$). If the read sets are sent, the dependency will be seen, and T_2 can be delayed until T_1 finishes.

In summary, we have seen that actions cannot simply be installed in ticket order at the backup stores. The system must guarantee that a transaction only commits when all the transactions that it depends on have committed at the backup. In addition, a transaction should not be delayed waiting for transactions it does *not* depend on, something that would happen if actions were done in *strict* ticket order. The mechanism we describe in the following section achieves these goals.

Finally, note that sending undo/redo logs (as opposed to simple redo logs as we are assuming) does not really eliminate the problems we have sketched here. If undo logs are included, it is possible for backup stores to install updates in local ticket order disregarding the state of other stores. If later it turns out that some transaction does not commit, its updates can be undone. However, the commit decisions must still be made at some point, and still involve making sure that all transactions that a transaction depends on have committed. It is not a good idea to delay all commit decisions until a disaster hits because

[1] the undo logs would become too large, and

[2] processing of new transactions at the backup after the disaster would be delayed until the commits complete. This may take a long time, since the cascading aborts mentioned above will lead to extensive searches in the logs.

Thus, we would still need a commit protocol to run as transactions are received, and it would be similar to the one we will describe for redo logging only. As we have stated, redo logging sends less data to the backup, so in the rest of the paper we will only deal with it.

7. Installing Actions at the Backup — Our Approach

In this section we present our method for installing the redo logs at the backup, prove its correctness and discuss its performance. In what follows, the notations T_x and $T(s_x)$ are equivalent and are both used to denote the transaction with ticket number s_x. The store is usually implied by the context.

No real processing of transactions takes place at the backup site, in the sense that no computation is performed for a transaction. The backup site is given the actions that were performed on the primary copy (in the form of a redo log) and has only to install the changes in a manner consistent with our correctness criteria. The backup host does not have much work to do. It gets a redo log from a primary host and then partitions the log by store and sends each part to the appropriate store. This can be viewed as the first phase of a (local) two-phase commit protocol. The stores install the changes (in the proper order, see below) and, when they are finished, they send an acknowledgement to the host. When the host receives acknowledgements from all of the stores, it executes the second phase and also sends an acknowledgement to the primary host (as mentioned in section 5). The two-phase commit protocol is used to make sure that the changes of a transaction are installed atomically, so that the backup copy is never left in an inconsistent state.

In section 6 we saw that updates in strict sequence order reduce parallelism. The intuition behind our solution is to detect exactly those cases where waiting is necessary and to let all other cases take advantage of parallelism. This is achieved through locks on the data items accessed by the transactions, which are *granted to the transactions in ticket number order*. Write (exclusive) locks are requested for items that are to be updated. For other items in the read set, read (shared) locks are requested. Additionally, a read lock on every table "name" accessed is requested, in order to ensure that the table is not deleted while accessing one of its records; if the table was created or destroyed, this lock must be exclusive.

For a concrete example, suppose that T_x has a smaller ticket number than T_y at one of the stores. If they access a data item in conflicting modes, our mechanism ensures that the lock is granted to T_x, the transaction with the smaller ticket number. Transaction T_y cannot get the lock until T_x releases it, i.e., until T_x commits. If, on the other hand, there is no dependency between the two transactions, then they will not ask for conflicting locks, so they will be able to proceed in parallel.

We now describe the locking mechanism at the backup in detail. When a redo log for transaction T_x with ticket number s_x arrives at backup store BS_j, it is placed in a queue of transactions ordered by ticket number. In this queue, T_x can be in a number of states:

LOCKING :	the transaction arrives and requests locks for all of the records and tables it accesses. Then it waits until all transactions with smaller ticket numbers have entered (or gone past) the SUBSCRIBED state (so that conflicts can be detected). Only then does it enter the SUBSCRIBED state.
SUBSCRIBED :	the transaction is waiting until it is granted the locks it requested. The transaction may then proceed to the PREPARED state.
PREPARED :	an acknowledgement for the first phase of the two-phase commit protocol has been sent to the backup host.
COMMITTED :	the message for the second phase has been received for this transaction. All updates have been made public and all locks have been released.

When T_x arrives, BS_j sets the state of T_x to LOCKING and starts requesting the locks required. Transaction T_x asks for a lock on data item z by inserting itself in a list of transactions asking for a lock on z; the list is sorted by ticket number. Each store has a counter that keeps track of the local ticket sequence, showing the ticket of the last transaction that entered the SUBSCRIBED state at this store.

The locking procedure is summarized in Figure 3. After all of the locks for T_x have been requested, T_x waits for the counter to reach s_x-1, if it has not already done so. This recursively implies that T_x will wait for all transactions with smaller ticket numbers to enter the SUBSCRIBED state. (Note that some or all of these transactions may be further ahead, in the PREPARED or COMMITTED state. The important thing is that they must have reached at least the SUBSCRIBED state before T_x can do so.) When this happens, T_x enters the SUBSCRIBED state itself. If T_x writes data at this store, then it increments the counter by 1. The increment in the counter may in turn trigger $T(s_x + 1)$ to enter the SUBSCRIBED state and so on. For example, if the current value of the counter is 25, then if transaction T_x with ticket 26 is in the LOCKING state, it enters the SUBSCRIBED state and increments the counter to 26, which may cause the transaction with ticket 27 to enter the SUBSCRIBED state and so on. If T_x were read only for this store, it would proceed to the subscribed state without incrementing the counter to 26.

In the SUBSCRIBED state T_x waits until all of its lock requests reach the head of their corresponding lists. (A read request is also considered at the head of a list if all requests with smaller ticket numbers are for read locks.) When this condition is met, T_x enters the PREPARED state and informs its host. After commit, all of T_x's requests are removed from the corresponding lists.

When a failure occurs at the primary site, the backup is informed that primary processing will be switched to it. The (former) backup tries to commit all of the transactions that can commit and aborts the

```
state($T_x$) = locking ;
t = ticket($T_x$) at this store ;
FOR each object z accessed by $T_x$
   add lock request (includes $T_x$'s ticket) to z list ;
WAIT UNTIL (counter >= t - 1);
IF $T_x$ writes at this store THEN
   counter = counter + 1 ;
state($T_x$) = subscribed;
FOR each object z accessed by $T_x$
   WAIT UNTIL $T_x$ request is at head of z list;
state($T_x$) = prepared;
send acknowledgement to coordinating host;
WAIT UNTIL commit message for $T_x$ is received;
FOR each object z accessed by $T_x$
   BEGIN
      IF $T_x$ wrote z THEN
           install new z value;
      remove $T_x$ lock request from z list;
   END ;
state($T_x$) = committed
```

Figure 3. Store pseudo-code for backup transaction processing

ones that cannot. It then enters the primary mode and takes over processing.

Observation: the atomicity constraint holds.

Argument: Atomicity is enforced by the local two-phase commit protocol at the backup. □

Observation: the mutual consistency constraint holds.

Argument: Since locks are granted in ticket order at the stores, updates inducing dependencies are installed in ticket order. As discussed at the beginning of section 6, mutual consistency is observed. □

Observation: the local consistency constraint holds.

Argument: Suppose $T_a \rightarrow T_b \rightarrow T_c \rightarrow \cdots \rightarrow T_n$ and T_a has not arrived yet at the backup site (we remind the reader that the dependencies are non R-W). Further, assume that $T_a \rightarrow T_b$ occurs at store S_1, $T_b \rightarrow T_c$ at S_2, and so on (note that these stores are not necessarily different). Since the dependency is non R-W, the ticket number of T_b is *higher* than that of T_a. According to our processing rules, T_b cannot enter the SUBSCRIBED state at BS_1 and will be unable to commit. At BS_2, T_b will lock the objects that caused $T_b \rightarrow T_c$ and will not release them until it commits, thus preventing T_c from entering the PREPARED state. Transaction T_c in turn prevents T_d from committing and so on. Thus, local consistency holds. □

Unfortunately, the mechanism that enforces the mutual consistency and local consistency constraints also causes the minimum divergence constraint to be violated. If T_x is missing at a store, then all transactions with higher sequence numbers will not commit, regardless of whether they conflict with T_x or not. In case of failure, a few transactions may be unnecessarily aborted. However, one can argue that this disadvantage will not be serious in practice, because the number of transactions unnecessarily aborted will be rather small and limited to transactions that executed just before the catastrophic failure. Given a maximum transmission delay $T_{\max}$ for the transactions to reach the backup site and a maximum processing rate $R_{\max}$, we can bound the number of transactions missed by the backup in case of failure by $T_{\max} * R_{\max}$.

It is possible to devise a mechanism that strictly enforces the minimum divergence constraint, but we believe that its overhead would be too high. To enforce the constraint, each transaction T_i would have to arrive at the backup site with a list of transactions it depends on. Transaction T_i would only be delayed at the backup if the transactions on its list were missing. To construct these lists, the primary site would have to keep track of the last transaction that accessed every lockable database object. This storage and processing cost would be incurred during normal operation. In contrast, our solution pays the price of a few aborted transactions only when a disaster hits. (The minimum divergence constraint could also be satisfied by a mechanism that uses two-phase commit between the two sites.)

There is not much we can formally prove regarding parallelism, but we can convince ourselves that the proposed mechanism is not a processing bottleneck at the backup site. Transactions are received by the backup hosts in parallel. At each store, locks are requested concurrently by multiple transactions. If the transactions with consecutive ticket numbers at one store access disjoint data sets, then they can acquire their locks at that store in parallel. The only part that is executed serially is the increment of the counter, which is relatively fast. Note that this increment is *not* a global bottleneck, since the increment at one store is independent from the increment at another store, even for the same transaction. On the other hand, if the transactions access common data items at a store, a certain amount of parallelism is lost, because they will acquire their locks sequentially. However, at the primary site, these conflicting transactions also executed sequentially, so the backup is not introducing new delays.

8. Initialization of the Backup Database

After considering operation under normal conditions, with one site in primary and one site in backup mode, we must now consider the system with one site in the primary mode and one in the recovering mode. This will be the case at system initialization and when a previously failed site recovers. The basic idea we will use is similar to a fuzzy dump [11], but there are some differences. In a conventional fuzzy dump one gets a dump of the database and a log of the actions performed while the dump was in progress. In order to restore the database, the dump is installed and the log is replayed against it. No transaction processing takes place while the restoration is performed. The correctness criterion for the restoration process is defined clearly: the database must be brought to the consistent state existing at the time the last transaction recorded on the log committed. Our application is complicated because there are multiple logs and because transaction processing cannot be suspended during recovery. This makes both the implementation and the correctness proof trickier.

The basic idea is for the primary site to scan the entire database and transmit it to the recovering site, along with the changes that occur while this scan is taking place and may therefore not be reflected in the scan. These changes are essentially a redo log and are transmitted over the communication lines used for normal operation. The scan data will probably take too long to be transmitted over the communication lines, so that an alternate path between primary and recovering site may be established. This path is usually a tape that is written at the primary site and carried to the backup. The order in which scan messages are received at the recovering site is irrelevant for our method, so that some scan messages may be sent over the communication lines while others are written on tape. Our scheme allows the use of multiple tapes to expedite the process.

For each primary store S_i, a primary host H_i and a backup host BH_i are selected to copy the S_i database. Host H_i will scan the database at S_i, passing it to BH_i, which in turn installs that data at the corresponding backup store BS_i. First H_i informs S_i that initialization is starting. Then S_i records the current ticket number s_x and returns it to H_i, which in turn sends it (through BH_i) to BS_i. Backup store BS_i sets its ticket number to s_x, creates a legal but empty database and starts accepting redo logs with ticket numbers higher than s_x; redo logs with lower ticket numbers are simply ignored because their effects will be reflected in the scan copy. Store BS_i remains in the recovery mode until initialization completes. If the primary fails while the backup is in recovery mode, nothing can be done: the backup database is still useless.

Under the control of H_i, store S_i starts scanning the portion of the database residing in it. At the same time, normal processing continues and redo logs are sent to the recovering site. The scanning of the database is like a long lived transaction which reads the entire database. For each object scanned (table or record), a scan message is sent to the backup with enough information for it to create the object. The scan process is described in detail in Figure 4.

Note that the locks are held only while an object is scanned. Records are locked one at a time; groups of records do not have to be locked together. (When the scan process tries to lock an object that is already write locked by a transaction, the scan process does not get blocked: it reads the before image of the record). Tables or records created by transactions with ticket numbers greater than s_x do not have to be

```
FOR each table DO
 BEGIN
    get a table read lock ;
    send a scan message for table creation ;
    FOR each record DO
      BEGIN
         get a read lock on the record ;
         send a scan message with the image of the record ;
         release the lock held on the record ;
      END
    release the lock held on the table ;
  END
```

Figure 4. The scan process

scanned, since they will be transmitted anyway in the redo log of the transaction that created them. However it will not be harmful if they are transmitted by the scan process too. The same holds for objects that have been updated after time s_x; it does not matter whether we send the object value that existed at time s_x or at some later time.

The backup store will receive data of two types: messages from the scan process and normal redo logs from transactions. The backup store processes both types of messages, using the simple rule of always trying to have the most recent copy for every object. In particular, when a redo log arrives (ticket number greater than s_x), it is processed as usual, except for the following:

[1] If a record update action is to be performed but the record does not exist yet, the update is treated as an insert. (This assumes that the update log entries contain the full after image of the modified record.)

[2] If a record (or table) insert arrives but the record (or table) already exists, the values in the insert replace existing ones. That is, the end result should be as if the record (or table) did not exist and the insert were normally executed.

[3] Deletes do not actually delete the record (or table). They simply mark it as deleted, but it is still reachable through the primary key index. If a delete arrives, and the object does not exist, a dummy record is inserted with the record key and a flag indicating it is deleted.

When scan messages arrive at BS_i, they are treated as insertions of the record or table. However, if the object already exists (even in deleted form), the scan message is ignored. This is because the scan message may contain obsolete data (i.e., data which have been superseded by a later version). Note that even if the scan message contains a later version than the one already existing no problem arises, because this later version will be installed when the redo log for the transaction that wrote this later version arrives.

The motivation for the above rules is as follows: during the initialization phase, there will be two types of objects, those accessed by transactions and those not accessed at all. If the object is not accessed, then we want the scan to give us the image of this object as it exists in the primary store. However, if the object is modified by a transaction, then we really did not need the scan for the object since the normal processing will deliver the new image. Our rules ensure that a useless scan does not get in our way. For example, consider a record z that exists at ''time'' s_x in S_i. The scan starts and sends the image of z. Some time later a transaction deletes z. If the delete arrives at BS_i before the scan, we could end up with a copy of z that should not exist. But since the delete creates a dummy record, the scan message will find it, and the scan copy will be discarded. After initialization, the deleted objects can be removed, so they do not represent a serious storage problem.

There is a subtle point regarding the time at which we may resume normal processing at the backup site, i.e., exit from the recovery mode. It is not sufficient to wait until the scan process finishes, say at ''time'' s_y. Let us illustrate this with an example. Suppose that T_i writes data items a and b, which satisfy some consistency constraint (e.g., $a + b = const$). First the scan process holds a read lock on a and transmits its before image (with respect to T_i). Then T_i gets write locks on both a and b, updates them, commits and releases the locks. The scanner gets a read lock on b, transmits its after image and then finishes (say b was the last unscanned object). Suppose that scan messages for a and b arrive at the backup ahead of the redo log message for T_i. Assume further that the T_i message never makes it because of a failure. If we let the backup site resume normal processing at this point, we end up with an inconsistent copy (we are left with the before image for a and the after image for b).

To avoid this type of problem, we use the following rule. Suppose that the scan process finishes at the primary store S_i when the ticket number is s_y. Then it is safe to resume normal processing at the corresponding backup store BS_i after

(a) all transactions through $T(s_y)$ have committed at the backup store, and

(b) all of the scan messages for this store have been received and processed.

The entire backup site can resume normal processing once all of its stores are ready for normal processing.

Observation: the above two conditions are sufficient for correctness.

Argument: Suppose the scan starts at primary store S_i at "time" s_x (i.e., after transaction $T(s_x)$ ran), and ends at time s_y. Let Z_0 be the state of S_i at time s_x and let T_{scan} be the set of transactions that committed at the primary during the scan.

At the backup store BS_i we start with an empty database state and install a set of transactions T_{back} that includes *all* transactions in T_{scan}. The resulting schedule is SCH_{back}. Concurrently, we process all scan messages and arrive at a database state Z_α. Note that T_{back} may contain more transactions than those in T_{scan}. This is because transactions that committed after time s_y at S_i can arrive before $T(s_y)$.

We want to show that Z_α is identical to Z_β, the state resulting from running SCH_{back} on Z_0. This can be done by considering each object z in the database.

- Case I. Object z was not modified by any transaction in T_{back}. Since $T_{scan} \subseteq T_{back}$ (property (a) above), z was not modified by any transaction at the primary site during the scan. Hence, the scan message for z will contain the value of z in Z_0, i.e., $Z_0(z)$. This value will be installed, so $Z_\alpha(z) = Z_0(z)$. This is exactly the value of $Z_\beta(z)$.
- Case II. Object z was modified by some transaction in T_{back}. Let T_j be the last transaction in SCH_{back} to have modified z, writing value z_j. That is, $Z_\beta(z) = z_j$. Next, let us look at the moment when z_j is installed by T_j at BS_i. If the scan message for z arrives after this time, it is ignored. If it arrives before this time, T_j overwrites z. In either case, $Z_\alpha(z) = z_j$. Thus, $Z_\alpha(z) = Z_\beta(z)$.

We have shown that after SCH_{back} is run, the state of BS_i is as if we had started with the initial state Z_0. Given the correctness of normal processing, the subsequent states of BS_i will also have this property. This means that after conditions (a) and (b) hold, our implicit assumption about initial states holds (see Section 4). □

9. Coping with Single Host or Store Failures

We give a brief discussion of what can be done in case a host fails (not a disaster). If a primary host H_i fails, a replacement host RH_i is selected at the primary site. Host RH_i is in charge of committing or aborting pending H_i transactions. In addition, RH_i is responsible for sending missing redo logs to BH_i. In order to achieve this, RH_i contacts all of the stores at the primary site and asks for transactions that were controlled by H_i, have committed, but are still in the redo logs. If BH_i does not have the redo logs for some of these transactions, RH_i sends them to BH_i. As in normal processing, when an acknowledgement is received that a transaction has successfully committed at the backup site, RH_i informs the primary stores to purge the corresponding log entries.

If a backup host BH_i fails, a replacement host RBH_i is selected. Host RBH_i contacts all of the backup stores and obtains the status of all transactions in progress that were originated by BH_i; RBH_i becomes the coordinator for these transactions and tries to commit them. Host RBH_i is also responsible for the transactions that ran while BH_i was down and before RBH_i took over. Host RBH_i contacts H_i and gets a list of pending redo logs that have not been acknowledged by the backup. H_i retransmits all pending logs until they are acknowledged.

Stores can also fail, but we have assumed that they have reliable storage and their own recovery method. If a store fails in a way that cannot be handled by its recovery mechanism, it is considered a disaster, and the general recovery scheme described in the previous section is used.

10. Extensions

It is possible to break up the data residing in a store into *chunks* and apply our proposed solution, generating ticket numbers per chunk instead of per store. For example, each relation could be a chunk. This has the advantage of gaining parallelism, by reducing the critical sections: there is less contention for getting the ticket numbers at the primary stores, less contention for the state flipping (from LOCKING to SUBSCRIBED) at the backup stores, etc. In addition to this, the impact of a missing transaction in case of failure is reduced. Let us illustrate with an example. Suppose disaster strikes and transaction T_x does not make it to the backup site. All transactions accessing data items in the same chunks as T_x that have been received properly at the backup but have higher ticket numbers than T_x have to be aborted. The less data that the chunks accessed by T_x contain, the fewer aborted transactions. On the other hand, more ticket numbers have to be processed both at the primary and the backup site, which will add some overhead.

Another optimization can be made in the scanning process. As we saw in section 8, some of the messages sent by the scanning process are ignored at the backup site. We can decrease the number of such messages if we keep track of what has been scanned at S_i. For simplicity assume that each record has a scan bit that indicates if it has been scanned. (This might actually be implemented with a cursor showing how far the scan process has gone.) As each record is scanned, the bit is set. If a transaction modifies a record that has not been scanned, the after image for that record need not be sent, since the scan process will send the after image of the record later. However, the redo log entry containing the ticket must be sent, so that the ticket sequence will not be broken.

Yet another interesting problem is the partition of the data among the stores. In this paper we took the partition to be identical at both sites. This is the most natural case (since the backup system will probably be a replica of the primary). However, it turns out that arbitrary partitions are not possible, at least within our framework. Consider the following case: backup store BS_{ij} contains data from primary stores S_i and S_j. Suppose that $\mathit{ticket}(T_x) < \mathit{ticket}(T_y)$ at S_i and $\mathit{ticket}(T_x) > \mathit{ticket}(T_y)$ at S_j (at one of the primary stores there is no dependency between the two transactions). It is possible that all of the data accessed by T_x and T_y resides on BS_{ij}. Which ticket number will be used to determine the order in which the transactions will enter the SUBSCRIBED state at BS_{ij}? This leaves only the possibility of a *finer* partition at the backup site. But it turns out that even this is not practical. Suppose the data on primary store S_i is split into backup stores BS_{i1} and BS_{i2} at the backup. Transaction T_x accesses data only on BS_{i1} and has $\mathit{ticket}(T_x) = k$. Transaction T_y accesses data only on BS_{i2} and has $\mathit{ticket}(T_y) = k + 1$. Transaction T_x will never appear in store BS_{i2}. This means that transaction T_y will never execute, because it cannot enter the SUBSCRIBED state unless T_x has previously done so! Thus, the partitions of data at the two sites have to be identical.

The method for recovery we suggested in section 8 assumes that the primary database is lost during a disaster. If this is not the case, it may be possible to bring the failed primary up-to-date by sending it the redo logs for the transactions it missed, instead of a copy of the entire database. The recovering primary also has to undo transactions that did not commit at the backup before redoing the missed transactions. The use of redo logs for recovery may or may not be feasible, depending on the time it takes for the failed site to be repaired. If this time is long, the logs will grow too big. The relative performance of the two methods also depends on the size of the logs. Our method is general and works even when the primary loses its copy entirely, without excluding the use of the other method. For example, one could combine the two strategies as follows: when a failure of the primary is detected, the backup takes over transaction processing and tries to keep the logs for as long as it can, in case the primary recovers soon and still has its copy. When its capacity overflows, it starts discarding the logs; when the failed site recovers, the method proposed in section 8 will be used.

11. Conclusions

We presented a method for keeping a remote backup database up-to-date for disaster recovery. The method ensures that the backup copy will be consistent with the primary and that in case of failure the backup copy will be (at most) a few transactions behind the primary. The method is relatively straightforward and and can be implemented using well known concepts and techniques, such as locking and logging. The overhead imposed at the primary site is relatively small, and there is *no central processing* in our mechanism, i.e., no component that must ''see'' all transactions. This means that the system can scale upwards: more communication lines, hosts and stores can be added without having backup management interfere.

ACKNOWLEDGEMENTS

The authors would like to thank Jim Gray for his comments and the references he provided.

REFERENCES

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

[2] W. Finkelstein and M. Cappi, ''Experiences with Large Networks of Computers,'' Presentation at the *International Workshop on High Performance Transaction Systems*, Pacific Grove, CA, September 1985.

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

[4] J. N. Gray and M. Anderton, ''Distributed computer systems: four case studies,'' *Proc. of the IEEE, Special Issue on Distributed Database Systems,* pp. 719-726, May 1987.

[5] J. N. Gray, ''Why do computers stop and what can be done about it ?,'' Presentation at the *Fifth Symposium on Reliability in Distributed Software and Database Systems,* Los Angeles, CA, January 1986.

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

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

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

[9] IBM, *IMS/VS Extended Recovery Facility (XRF): General Information*, Document Number GG24-3150, March 1987.

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

[11] D. J. Rosenkrantz, ''Dynamic Database Dumping,'' *Proc. SIGMOD Int'l Conf. on Management of Data*, pp. 3-8, ACM, 1978.

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

[13] R. D. Schlichting and F. D. Schneider, ''Fail-stop processors: an approach to designing fault-tolerant computing systems,'' *ACM, Transactions on Computer Systems,* Vol. 1, pp. 222-238, August 1983.

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

[15] A. S. Tanenbaum, *Computer Networks.* Englewood Cliffs, NJ: Prentice Hall, 1988.